Alberto Pettorossi

Semantics of Programming Languages

ARACNE

Contents

Preface

In these lecture notes we present a few basic approaches to the definition of the semantics of programming languages. In particular, we present: (i) the operational semantics and the axiomatic semantics for a simple imperative language, and (ii) the operational semantics and the denotational semantics for some first order and higher order, typed functional languages. We then present some basic techniques for proving properties of imperative, functional, and concurrent programs. We closely follow the presentation done in the book by Glynn Winskel [**19**].

I express my gratitude to my colleagues at the Department of Informatics, Systems, and Production of the University of Roma Tor Vergata, and to my students and my co-workers Fabio Fioravanti, Fulvio Forni, Maurizio Proietti, and Valerio Senni for their support and encouragement. Thanks to Michele Ranieri and Massimiliano Macchia for pointing out some imprecisions in a preliminary version of these lecture notes.

Many thanks also to the Aracne Publishing Company for its helpful cooperation.

Roma, December 2010

Alberto Pettorossi
Department of Informatics, Systems, and Production
University of Roma Tor Vergata
Via del Politecnico 1,
I-00133 Roma, Italy
email: pettorossi@info.uniroma2.it
URL: http://www.iasi.rm.cnr.it/~adp

CHAPTER 1

Propositional Calculus, Predicate Calculus, and Peano Arithmetics

In this chapter we briefly recall the main concepts of Propositional Calculus (see Section 1), Predicate Calculus (see Section 2 on page 14), and Peano Arithmetics (see Section 3 on page 19). For more details the reader may refer to [**12**] or other classical books on Mathematical Logic.

1. Propositional Calculus

Let us introduce the syntax of Propositional Calculus by defining: (i) the set of variables, (ii) the set of formulas, (iii) the axioms, and (iv) the deduction rule.

1.1. Syntax of Propositional Calculus.

Let N denote the set of natural numbers. The set *Vars* of *variables* of the Propositional Calculus is defined as follows:

$$Vars = \{P_i \mid i \in N\}$$

The set of *formulas* is defined as follows:

$$\varphi ::= P \mid \neg\varphi \mid \varphi_1 \to \varphi_2$$

where $P \in Vars$ and φ, φ_1, and φ_2 are formulas. The connectives $\wedge$ (and), $\vee$ (or), and $\leftrightarrow$ (if and only if) can be expressed in terms of $\neg$ and $\to$, as usual. We assume the following decreasing order of precedence among connectives: $\neg$ (strongest precedence), $\wedge$, $\vee$, $\to$, $\leftrightarrow$ (weakest precedence).

Parentheses can be used for overriding precedence. If the precedence among operators is the same, then we assume *left associativity*.

For instance, (i) $\neg P_1 \wedge P_2$ stands for $(\neg P_1) \wedge P_2$, and (ii) $P_1 \to P_2 \to P_1$ stands for $(P_1 \to P_2) \to P_1$.

Axiom Schemata. We have the following three axiom schemata for all formulas φ, ψ, and χ:

1. $\varphi \to (\psi \to \varphi)$
2. $(\varphi \to (\psi \to \chi)) \to ((\varphi \to \psi) \to (\varphi \to \chi))$
3. $(\neg\varphi \to \neg\psi) \to (\psi \to \varphi)$

Deduction Rule. Given the formulas φ and $\varphi \to \psi$ we get the new formula ψ via the following deduction rule, called *Modus Ponens*. It is usually represented as follows:

$$\frac{\varphi \quad \varphi\to\psi}{\psi} \qquad \text{(Modus Ponens)}$$

A formula which is above the horizontal line of the deduction rule is said to be a *premise* of the rule. A formula which is below the horizontal line of the deduction

rule is said to be a *conclusion* of the rule. A conclusion of a rule is said to be a *direct consequence* of the premises of the rule.

We have the following definition.

DEFINITION 1.1. [**Theorem**] We say that a formula φ is a *theorem*, and we write $\vdash \varphi$, iff there exists a sequence $\varphi_1, \ldots, \varphi_n$ of formulas such that: (i) $\varphi_n = \varphi$, and (ii) each formula in the sequence is *either* an instance of an axiom schema *or* it is derived by Modus Ponens from two preceding formulas of the sequence.

In order to define the semantics of the Propositional Calculus we need to introduce the notions of: (i) the set of variables in a formula, and (ii) the variable assignment.

Given a formula φ, the set $vars(\varphi)$ of variables in φ is defined by structural induction as follows: for every variable P and every formula φ, φ_1, φ_2,

$vars(P) = \{P\}$
$vars(\neg\varphi) = vars(\varphi)$
$vars(\varphi_1 \rightarrow \varphi_2) = vars(\varphi_1) \cup vars(\varphi_2)$

A *variable assignment* σ is a function from $V \subseteq Vars$ to $\{true, false\}$. Thus, a variable assignment with domain V is a set of pairs, each of which is of the form $\langle P, b\rangle$, with $P \in V$ and $b \in \{true, false\}$. A pair of the form $\langle P, b\rangle$ is called an *assignment to* P.

Let $Assignments$ denote the set of all variable assignments.

There are other presentations of the Propositional Calculus where, instead of a deduction rule only (that is, the Modus Ponens), one introduces more inference rules. These rules allow us to derive new formulas from old ones and provide a natural understanding of how the logical connectives of the Propositional Calculus behave. Now we present ten of these inference rules. Among them there is the Modus Ponens rule and, thus, it is not difficult to show that these ten rules are complete (see Theorem 1.2 on the next page).

Let us first introduce the *entailment relation*.

Given a set Γ of formulas and a formula φ, $\Gamma \vdash \varphi$ denotes the existence of a sequence of formulas such that each of them is in Γ or it is derived from previous formulas in the sequence by a rule listed below (see Rules (1)–(10)). Note the overloaded use of the operator $\vdash$ which is now used as a binary operator.

When $\Gamma \vdash \varphi$ we say that Γ *entails* φ. We also say that φ *is inferred from* Γ.

As for any formal system, we have the following properties: for any set of formulas Γ and Δ, for any formula φ,

(A.1) if $\varphi \in \Gamma$ then $\Gamma \vdash \varphi$,
(A.2) if $\Gamma \subseteq \Delta$ and $\Gamma \vdash \varphi$ then $\Delta \vdash \varphi$,
(A.3) $\Gamma \vdash \varphi$ iff there exists a *finite* subset Φ of Γ such that $\Phi \vdash \varphi$, and
(A.4) if $\Delta \vdash \varphi$ and for all $\psi \in \Delta$, $\Gamma \vdash \psi$, then $\Gamma \vdash \varphi$.

Here are the inference rules for the entailment relation which hold for all formulas φ, ψ, and ρ.

(1) *True axiom*
$\{\} \vdash \varphi \vee \neg\varphi$

(2) *False axiom*
$\{\} \vdash \neg(\varphi \wedge \neg\varphi)$

(3) *Reductio ad absurdum* (*negation introduction*)
$\{\varphi,\ \ \psi \to \neg\varphi\} \ \vdash\ \neg\psi$

(4) *Double negation elimination*
$\{\neg\neg\varphi\} \ \vdash\ \varphi$

(5) *Conjunction introduction*
$\{\varphi, \psi\} \ \vdash\ \varphi \wedge \psi \qquad \{\varphi, \psi\} \ \vdash\ \psi \wedge \varphi$

(6) *Conjunction elimination*
$\{\varphi \wedge q\} \ \vdash\ \varphi \qquad \{\varphi \wedge q\} \ \vdash\ q$

(7) *Disjunction introduction*
$\{\varphi\} \ \vdash\ \varphi \vee \psi \qquad \{\psi\} \ \vdash\ \varphi \vee \psi$

(8) *Disjunction elimination*
$\{\varphi \vee \psi,\ \ \varphi \to \rho,\ \ \psi \to \rho\} \ \vdash\ \rho$

(9) *Modus Ponens* (*conditional elimination*)
$\{\varphi,\ \ \varphi \to \psi\} \ \vdash\ \psi$

(10) *Conditional proof* (*conditional introduction*)
If $\{\varphi\} \vdash \psi$ then $\{\} \vdash \varphi \to \psi$

Rule 10 is a conditional rule: it asserts the existence of a new pair of the entailment relation from the existence of an old pair. Note that the other direction of the implication of Rule (10) can be proved as follows.

(i) $\{\} \vdash \varphi \to \psi$ (given)
(ii) $\{\varphi\} \vdash \varphi$ (by $(A.1)$ above)
(iii) $\{\varphi\} \vdash \varphi \to \psi$ (by $(A.2)$ above)
(iv) $\{\varphi\} \vdash \psi$ (by (ii), (iii), $(A.4)$, and Modus Ponens)

We have the following fact.

THEOREM 1.2. A formula φ is a theorem of the Propositional Calculus iff $\{\} \vdash \varphi$.

Thus, given a formula φ of the Propositional Calculus we may test whether or not φ is a theorem by using the above Theorem 1.2. However, the presence of many inference rules make the test a bit difficult. In order to overcome this difficulty, in Section 10 on page 54 we will present a method based on rewriting rules.

1.2. Semantics of Propositional Calculus via the Semantic Function.

The semantic function $[\![\_]\!]$ of the Propositional Calculus takes a formula φ and a variable assignment σ with finite domain V such that $vars(\varphi) \subseteq V$, and returns an element in $\{true, false\}$.

The function $[\![\_]\!]$ is defined by structural induction as follows: for every variable P and every formula φ, φ_1, φ_2,

$[\![P]\!]\,\sigma = \sigma(P)$
$[\![\neg\varphi]\!]\,\sigma = not\ [\![\varphi]\!]\,\sigma$
$[\![\varphi_1 \to \varphi_2]\!]\,\sigma = [\![\varphi_1]\!]\,\sigma\ implies\ [\![\varphi_2]\!]\,\sigma$

EXAMPLE 1.3. $[\![P_1 \to P_2]\!]\,\{\langle P_1,\, \mathit{false}\rangle,\, \langle P_2,\, \mathit{true}\rangle\} = \mathit{true}$.

EXAMPLE 1.4. $[\![P_1 \to P_2]\!]\,\{\langle P_1,\, \mathit{false}\rangle,\, \langle P_2,\, \mathit{true}\rangle,\, \langle P_4,\, \mathit{true}\rangle\} = \mathit{true}$.

DEFINITION 1.5. [**Tautology**] We say that a formula φ is a *tautology*, and we write $\models \varphi$, iff for every variable assignment σ with domain $\mathit{vars}(\varphi)$, we have that $[\![\varphi]\!]\,\sigma = \mathit{true}$.

A tautology is also called a *valid* formula. The negation of a valid formula is said to be an *unsatisfiable* formula. A formula that is not unsatisfiable is said to be a *satisfiable* formula.

We have the following important result which gives us an algorithm for checking (in exponential time) whether or not a formula φ is a theorem by checking whether or not φ is a tautology. Indeed, we can check whether or not a formula φ is a tautology by checking whether or not for every variable assignment σ with domain $\mathit{vars}(\varphi)$, we have that $[\![\varphi]\!]\,\sigma = \mathit{true}$.

Note that if there are n distinct variables in a formula φ, then there are 2^n distinct variable assignments with domain $\mathit{vars}(\varphi)$. Thus, the checking algorithm has an exponential time bound.

THEOREM 1.6. [**Completeness Theorem for the Propositional Calculus**] For all formulas φ, $\vdash \varphi$ iff $\models \varphi$.

1.3. Semantics of Propositional Calculus via the Satisfaction Relation.

In this section we present the semantics of the Propositional Calculus via a binary relation $\models \subseteq \mathit{Assignments} \times \mathit{Formulas}$, called the *satisfaction relation.*

A pair $\langle \sigma, \varphi\rangle$ in the relation $\models$ will be denoted by $\sigma \models \varphi$, and when $\sigma \models \varphi$ holds we say that the variable assignment σ satisfies φ.

For every variable assignment σ, every variable P, every formula φ, φ_1, φ_2, we define $\sigma \models \varphi$ by structural induction as follows:

$\sigma \models P$ iff $\sigma(P) = \mathit{true}$

$\sigma \models \neg\varphi$ iff $\mathit{not}\ \sigma \models \varphi$

$\sigma \models \varphi_1 \to \varphi_2$ iff $\sigma \models \varphi_1\ \mathit{implies}\ \sigma \models \varphi_2$ (that is, $(\mathit{not}\ \sigma \models \varphi_1)\ \mathit{or}\ \sigma \models \varphi_2$)

The following fact establishes the equivalence between this semantics and the semantics of the Propositional Calculus we have defined in Section 1.2.

FACT 1.7. For every formula φ, for every variable assignment σ with domain $\mathit{vars}(\varphi)$, we have that: $[\![\varphi]\!]\,\sigma = \mathit{true}$ iff $\sigma \models \varphi$.

As a consequence, we get the following equivalent definition of a tautology:
a formula φ is a *tautology*, and we write $\models \varphi$, iff for every variable assignment σ with domain $\mathit{vars}(\varphi)$, we have that $\sigma \models \varphi$.

2. Predicate Calculus

This section is devoted to the syntax and the semantics of the First Order Predicate Calculus (Predicate Calculus, for short) [**12**].

2.1. Syntax of Predicate Calculus.

The set *Vars* of *variables* of the Predicate Calculus is a denumerable set defined as follows:

$$Vars = \{x, y, z, \dots\}$$

The set of the *function symbols* (with arity $r \geq 0$) is a finite, or denumerable, or empty set of symbols, defined as follows:

$$\{f, s, \dots\}$$

Function symbols of arity 0 are called *constants.*

The set of the *predicate symbols* (with arity $r \geq 0$) is a finite or denumerable, non-empty set of symbols, defined as follows:

$\{p, q, \dots\}$ (finite or denumerably many predicate symbols).

The symbols *true* and *false* are two predicate symbols of arity 0.

The set of *terms* is constructed as usual from variables and function symbols of arity $r\,(\geq 0)$ applied to $r\,(\geq 0)$ terms.

The set of *atoms* (or *atomic formulas*) are constructed from predicate symbols of arity $r\,(\geq 0)$ applied to $r\,(\geq 0)$ terms.

The set of *formulas* of the Predicate Calculus is defined as follows, where $x \in$ *Vars*, $A \in$ *atomic formulas*, and $\varphi, \varphi_1, \varphi_2 \in$ *formulas:*

$$\varphi ::= A \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \rightarrow \varphi_2 \mid \varphi_1 \leftrightarrow \varphi_2 \mid \exists x.\, \varphi \mid \forall x.\, \varphi$$

We will feel free to write $\exists x\, \varphi$, or $\exists x,\, \varphi$, instead of $\exists x.\, \varphi$. The *scope* of the quantifier $\exists x$ in the formula $\exists x\, \varphi$ is φ. Analogously for the quantifier $\forall x$, instead of $\exists x$.

We assume the following decreasing order of precedence among connectives and the quantifiers: $\neg$ (strongest precedence), $\wedge$, $\vee$, $\forall x$, $\exists x$, $\rightarrow$, $\leftrightarrow$ (weakest precedence). Parentheses can be used for overriding precedence. If the precedence among operators is the same, then we assume *left associativity.*

In order to define the semantics of the Predicate Calculus now we introduce the notions of: (i) the set of variables in a formula φ, denoted $vars(\varphi)$, and (ii) the set of free variables in a formula φ, denoted $vars(\varphi)$.

For any variable x, any function symbol f, any predicate symbol p, any term $t_1, \dots, t_n$, any formula $\varphi, \varphi_1, \varphi_2$, we have that:

$$\begin{aligned}
&vars(x) = \{x\} \\
&vars(f(t_1, \dots, t_n)) = vars(t_1) \cup \dots \cup vars(t_n) && \text{(for terms)} \\
&vars(p(t_1, \dots, t_n)) = vars(t_1) \cup \dots \cup vars(t_n) && \text{(for atomic formulas)} \\
&vars(\neg\varphi) = vars(\varphi) \\
&vars(\varphi_1 \vee \varphi_2) = vars(\varphi_1) \cup vars(\varphi_2) \\
&vars(\varphi_1 \wedge \varphi_2) = vars(\varphi_1) \cup vars(\varphi_2) \\
&vars(\varphi_1 \rightarrow \varphi_2) = vars(\varphi_1) \cup vars(\varphi_2) \\
&vars(\varphi_1 \leftrightarrow \varphi_2) = vars(\varphi_1) \cup vars(\varphi_2) \\
&vars(\exists x.\, \varphi) = vars(\varphi) \cup \{x\} \\
&vars(\forall x.\, \varphi) = vars(\varphi) \cup \{x\}
\end{aligned}$$

For any variable x, any function symbol f, any predicate symbol p, any term $t_1, \ldots, t_n$, any formula $\varphi, \varphi_1, \varphi_2$, we have that:

$$\begin{array}{ll}
\mathit{freevars}(x) = \{x\} & \\
\mathit{freevars}(f(t_1, \ldots, t_n)) = \mathit{freevars}(t_1) \cup \ldots \cup \mathit{freevars}(t_n) & \text{(for terms)} \\
\mathit{freevars}(p(t_1, \ldots, t_n)) = \mathit{freevars}(t_1) \cup \ldots \cup \mathit{freevars}(t_n) & \text{(for atomic formulas)} \\
\mathit{freevars}(\neg\varphi) = \mathit{freevars}(\varphi) & \\
\mathit{freevars}(\varphi_1 \vee \varphi_2) = \mathit{freevars}(\varphi_1) \cup \mathit{freevars}(\varphi_2) & \\
\mathit{freevars}(\varphi_1 \wedge \varphi_2) = \mathit{freevars}(\varphi_1) \cup \mathit{freevars}(\varphi_2) & \\
\mathit{freevars}(\varphi_1 \rightarrow \varphi_2) = \mathit{freevars}(\varphi_1) \cup \mathit{freevars}(\varphi_2) & \\
\mathit{freevars}(\varphi_1 \leftrightarrow \varphi_2) = \mathit{freevars}(\varphi_1) \cup \mathit{freevars}(\varphi_2) & \\
\mathit{freevars}(\exists x.\, \varphi) = \mathit{freevars}(\varphi) - \{x\} & \\
\mathit{freevars}(\forall x.\, \varphi) = \mathit{freevars}(\varphi) - \{x\} &
\end{array}$$

A formula φ is said to be *closed* if $\mathit{freevars}(\varphi) = \emptyset$.

An occurrence of a variable in a formula φ is said to be a *bound occurrence* in φ iff it is (i) either the occurrence x of the quantifier $\forall x$ in φ, (ii) or the occurrence x of the quantifier $\exists x$ in φ, (iii) or it is an occurrence of the variable x in the scope of a quantifier $\forall x$ or $\exists x$.

An occurrence of a variable in a formula φ is said to be a *free occurrence* if it is not bound.

A variable which has a free occurrence in a formula φ is said to be a *free variable* of φ.

A variable which has a bound occurrence in a formula φ is said to be a *bound variable* of φ.

When we write $\varphi(x_1, \ldots, x_n)$ we mean that *some* (maybe *none*) of the free variables of the formula φ are in the set $\{x_1, \ldots, x_n\}$.

Note that it may be the case that: (i) some of the variables in $\{x_1, \ldots, x_n\}$ *do not* occur free in φ, and (ii) some of the free variables of φ are not in $\{x_1, \ldots, x_n\}$.

Given the terms $t_1, \ldots, t_n$, by $\varphi(t_1, \ldots, t_n)$ we denote the formula $\varphi(x_1, \ldots, x_n)$ where *all free* occurrences, if any, of the variables $x_1, \ldots, x_n$ have been replaced by $t_1, \ldots, t_n$, respectively.

A term t *is free for* x *in* $\varphi(x)$ if (i) no free occurrence of the variable x in $\varphi(x)$ occurs in the scope of a quantifier $\forall x$ or $\exists x$, and (ii) x is a variable of t. This means that if t is substituted for all free occurrences, if any, of the variable x in $\varphi(x)$, then no occurrence of a variable in t becomes a bound occurrence in the resulting formula $\varphi(t)$.

In the Predicate Calculus we have the following axiom schemata and deduction rules.

Axiom Schemata. We have the following five axiom schemata for all formulas φ, ψ, and χ and for all terms t:

1. $\varphi \rightarrow (\psi \rightarrow \varphi)$
2. $(\varphi \rightarrow (\psi \rightarrow \chi)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi))$
3. $(\neg\varphi \rightarrow \neg\psi) \rightarrow (\psi \rightarrow \varphi)$
4. $(\forall x\, \varphi(x)) \rightarrow \varphi(t)$ — if the term t is free for x in $\varphi(x)$
5. $(\forall x\, (\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow (\forall x\, \psi))$ — if x is not a free variable in φ

*Deduction Rule*s. We have the following two deduction rules:

$$\frac{\varphi \quad \varphi \to \psi}{\psi} \quad \text{(Modus Ponens)} \qquad\qquad \frac{\varphi}{\forall x\, \varphi} \quad \text{(Generalization)}$$

As in the Propositional Calculus, a conclusion of a rule is said to be a *direct consequence* of the premises of the rule.

Let us also introduce the following definition (see also the entailment relation in the Propositional Calculus on page 12).

DEFINITION 2.1. [**Derivation (or Proof**)] Given a set Γ of formulas and a formula φ, a *derivation* (or a *proof*) of φ from Γ, denoted $\Gamma \vdash \varphi$, is a sequence of formulas ending with φ, such that each of formula in the sequence is either
(i) a formula in Γ, or
(ii) an axiom, that is, an instance of an axiom schema, or
(iii) it can be obtained by using the Modus Ponens rule from two preceding formulas of the sequence, or
(iv) it can be obtained by using the Generalization rule from a preceding formula of the sequence.

We write $\Gamma, \varphi \vdash \psi$ to denote $\Gamma \cup \{\varphi\} \vdash \psi$.

DEFINITION 2.2. [**Theorem**] We say that a formula φ is a *theorem* iff $\emptyset \vdash \varphi$.

When $\Gamma = \emptyset$, $\Gamma \vdash \varphi$ is also written as $\vdash \varphi$.

Given two formulas φ and ψ occurring in a derivation from Γ, we say that ψ *depends on* φ iff either ψ is φ or there exists a formula σ such that ψ is a direct consequence of σ and σ depends on φ.

THEOREM 2.3. [**Deduction Theorem**] (i) If $\Gamma \vdash \varphi \to \psi$ then $\Gamma, \varphi \vdash \psi$. (ii) Let us assume that in a derivation of ψ from $\Gamma \cup \{\varphi\}$, whenever we apply the Generalization rule to a formula, say χ, whereby deriving $\forall x\, \chi$, either (ii.1) χ does not depend on φ or (ii.2) x does not belong to *freevars*(φ). If $\Gamma, \varphi \vdash \psi$ then $\Gamma \vdash \varphi \to \psi$.

2.2. Semantics of Predicate Calculus.

In order to define the semantics of the Predicate Calculus we start off by introducing the notion of an interpretation.

An *interpretation* I is defined as follows.
(1) We take a *non-empty* set D, called the *domain* of the interpretation.
(2) To each function symbol of arity $r\,(\geq 0)$ we assign a function from D^r to D. (The elements of D^r are r-tuples of elements in D.) To each constant symbol we assign an element of D.
(3) To each predicate symbol of arity $r\,(\geq 0)$ we assign an r-ary relation, that is, a subset of D^r. To *true* we assign the subset of D^0, which is D^0 itself, that is, the set $\{\langle\rangle\}$ whose only element is the tuple $\langle\rangle$ with 0 components. To *false* we assign the subset of D^0 which is the empty set, denoted $\emptyset$, as usual.

A *variable assignment* σ is a function from *Vars* to D. Given an interpretation I and a variable assignment σ, we assign an element d of D to every term t as follows:
(i) if t is a variable, say x, then $d = \sigma(x)$, and

(ii) if t is $f(t_1, \ldots, t_r)$ then $d = f_I(d_1, \ldots, d_r)$, where f_I is the function from D^r to D assigned to f by I, and $\langle d_1, \ldots, d_r \rangle$ is the r-tuple of D^rassigned to $\langle t_1, \ldots, t_r \rangle$ by I.

In this case we say that d is the element of D assigned to t by I and σ.

Given an interpretation I, a variable assignment σ, and a formula φ, we define the *satisfaction relation*, denoted $I, \sigma \models \varphi$, by structural induction as follows. (In this definition, by $\sigma[d/x]$ we denote the variable assignment which is equal to σ except that $\sigma(x) = d$.) For all interpretation I, all variable assignment σ, all formulas φ and ψ, we have that:

$I, \sigma \models \mathit{true}$		
$I, \sigma \models p(t_1, \ldots, t_r)$		for every atom $p(t_1, \ldots, t_r)$ iff the r-tuple $\langle v_1, \ldots, v_r \rangle$ belongs to the relation assigned to p by I, where for $i = 1, \ldots, r$, v_i is the element of D assigned to t_i by I and σ
$I, \sigma \models \neg\varphi$	iff	*not* $(I, \sigma \models \varphi)$
$I, \sigma \models \varphi \wedge \psi$	iff	$I, \sigma \models \varphi$ *and* $I, \sigma \models \psi$
$I, \sigma \models \varphi \vee \psi$	iff	$I, \sigma \models \varphi$ *or* $I, \sigma \models \psi$
$I, \sigma \models \varphi \rightarrow \psi$	iff	$I, \sigma \models \varphi$ *implies* $I, \sigma \models \psi$ (that is, (*not* $I, \sigma \models \varphi$) *or* $I, \sigma \models \psi$)
$I, \sigma \models \exists x\, \varphi$	iff	*there exists* d in D such that $I, \sigma[d/x] \models \varphi$
$I, \sigma \models \forall x\, \varphi$	iff	*for all* d in D we have that $I, \sigma[d/x] \models \varphi$

If $I, \sigma \models \varphi$ holds, we say that the interpretation I for the variable assignment σ *satisfies* φ, or φ *is true in* the interpretation I for the variable assignment σ.

Note that for every I and σ we have that $I, \sigma \models \mathit{true}$, because $\langle\rangle$ belongs to $\{\langle\rangle\}$, and it is not the case that $I, \sigma \models \mathit{false}$, because $\langle\rangle$ does *not* belong to the empty set $\emptyset$.

We say that a formula φ is *satisfiable* iff there exist I and σ such that $I, \sigma \models \varphi$. A formula φ is *unsatisfiable* iff it is not satisfiable.

We say that φ *is true in an interpretation* I or I *is a model of* φ, and we write $I \models \varphi$, iff for all σ we have that $I, \sigma \models \varphi$.

DEFINITION 2.4. [**Logically Valid Formulas**] We say that a formula φ is *logically valid* (or *valid*, for short) and we write $\models \varphi$, iff for every interpretation I and every variable assignment σ we have that $I, \sigma \models \varphi$.

THEOREM 2.5. For every interpretation I and for every formula φ we have that $I \models \varphi$ iff $I \models \forall x\, \varphi$. (Note that it does not matter whether or not x occurs in φ.)

The following theorem establishes the correspondence between the relation $\vdash$ and the relation $\models$.

THEOREM 2.6. [**Gödel Completeness Theorem**] For every set Γ of *closed* formulas and for every formula φ, we have that $\Gamma \vdash \varphi$ iff $\Gamma \models \varphi$. In particular, $\vdash \varphi$ iff $\models \varphi$, that is, the theorems of the predicate calculus are precisely the logically valid formulas.

3. Peano Arithmetics

In this section we present the syntax of Peano Arithmetics. Peano Arithmetics is a First Order Predicate Calculus with: (i) the extra axiom schemata E1 and E2 (these axiom schemata make the Predicate Calculus to be a Predicate Calculus with equality), and (ii) the extra axiom schemata PA1 through PA9 listed below [**12**].

Here are the axioms E1 and E2.

E1. $\forall x\,(x{=}x)$

E2. for every formula $\varphi(x,x)$, $\forall x\,\forall y\,(x{=}y \rightarrow (\varphi(x,x) \rightarrow \varphi(x,y)))$

The formula $\varphi(x,y)$ denotes the formula $\varphi(x,x)$ where *some* free occurrences of the variable x have been replaced by y and y is free for x in $\varphi(x,x)$. Thus, $\varphi(x,y)$ may or may not have a free occurrence of x. (Recall that by our convention, when we write the formula $\psi(x)$ we do not mean that x actually occurs free in $\psi(x)$.

As a consequence of axioms E1 and E2, one can show that the equality predicate $=$ enjoys: (i) reflexivity, (ii) symmetry, (iii) transitivity, and (iv) substitutivity, that is, $\forall x\,\forall y\,(x{=}y \rightarrow (\varphi(x,x) \leftrightarrow \varphi(x,y)))$.

Let us consider the following extra function symbols: (i) the nullary constant 0, called *zero*, (ii) the unary function s, called *successor*, (iii) the binary function $+$, called *plus*, and (iv) the binary function $\times$, called *times*.

Here are the axioms PA1–PA9.

PA1. $\forall x\,\forall y\,\forall z\ (x{=}y \rightarrow (x{=}z \rightarrow y{=}z))$

PA2. $\forall x\,\forall y\ (x{=}y \rightarrow (s(x){=}s(y)))$

PA3. $\forall x\ 0 \neq s(x)$

PA4. $\forall x\,\forall y\ (s(x){=}s(y) \rightarrow x{=}y)$

PA5. $\forall x\ \ x + 0 = x$

PA6. $\forall x\,\forall y\ \ x + s(y) = s(x + y)$

PA7. $\forall x\ \ x \times 0 = 0$

PA8. $\forall x\,\forall y\ \ x \times s(y) = (x \times y) + x$

PA9. For any formula $\varphi(x)$, $\big(\varphi(0) \ \wedge\ \forall x\,(\varphi(x) \rightarrow \varphi(s(x)))\big) \rightarrow \forall x\,\varphi(x)$

Note that: (i) PA1 is a consequence of E1, and (ii) PA2 is a consequence of E1 and E2. PA9 is called the *principle of mathematical induction* (see also Section 1.1 on page 59). It stands for an infinite number of axioms, one axiom for each formula $\varphi(x)$. (We leave to the reader to prove that the formulas of Peano Arithmetics are as many as the natural numbers.)

A set T of formulas is said to be a *consistent* (or a *consistent theory*) iff it does not exist any formula φ such that both $T \vdash \varphi$ and $T \vdash \neg\varphi$ hold. A set T of formulas is said to be an *inconsistent* (or an *inconsistent theory*) iff it is not consistent. We have that a set T of formulas is inconsistent iff $T \vdash \mathit{false}$.

A formula φ is said to be *undecidable* in a set T of formulas iff neither $T \vdash \varphi$ nor $T \vdash \neg\varphi$ holds.

A set T of formulas is said to be *complete* (or a *complete theory*) iff for any closed formula φ, either $T \vdash \varphi$ or $T \vdash \neg\varphi$ holds or both. Thus, given a set T of formulas, if T is inconsistent then T is complete.

The following theorem establishes the incompleteness of the set of theorems of Peano Arithmetics [**12**].

THEOREM 3.1. [**Gödel-Rosser Incompleteness Theorem**] If the set *PA* of all theorems of Peano Arithmetics is consistent, then there exists a closed formula which is undecidable in *PA*.

NOTATION 3.2. When writing quantified formulas we will feel free to use familiar abbreviations. In particular, given the variables x and y ranging over the natural numbers, we will write $x \leq y$, instead of $\exists z.\, x+z=y$, and we will write $\forall x,\ 0 \leq x < k$, $p(x,k)$, instead of $\forall x.\, (0 \leq x \ \wedge\ x < k) \rightarrow p(x,k)$. □

4. Some Mathematical Notations

In the sequel we will consider the following sets and notations.

Natural numbers and integer numbers. The set of natural numbers is $\{0,1,2,\ldots\}$ and is denoted by ω. We will feel free to denote the set of the natural numbers also as $N^{\geq 0}$. By N we denote, unless otherwise specified, the set $\{\ldots,-2,-1,0,1,2,\ldots\}$ of the integer numbers.

Function notation. A function which, given the two arguments x and y, returns the value of the expression e, is denoted by $\lambda x.\lambda y.\, e$, or $\lambda x, y.\, e$, or $\lambda(x,y).\, e$. In this case we say that that function is denoted by using *the lambda notation*.

The application of the function f to the argument x is denoted by $(f\, x)$ or $f(x)$ or simply $f\, x$, when no confusion arises.

Rules for the lambda notation. When using the lambda notation for denoting functions, we consider the following rules.

(i) *α-rule* (change of bound variables). The expression $\lambda x.e[x]$, where $e[x]$ denotes an expression with zero or more occurrences of the variable x (those occurrences are free in $e[x]$ and bound in $\lambda x.e[x]$), is the same as $\lambda y.e[y]$, where: (i) y is a variable not occurring in $e[x]$, and (ii) $e[y]$ denotes the expression $e[x]$ where all the free occurrences of x in $e[x]$ have been replaced by y. For instance, $\lambda x.x+1$ and $\lambda y.y+1$ both denote the familiar successor function on natural numbers.

(ii) *β-rule*. The expression $(\lambda x.e)\, t$ is the same as the expression $e[t/x]$, that is, the expression e where the free occurrences of x have all been replaced by the expression t. For instance, $(\lambda x.x+1)\, 0 = 0+1$.

(iii) *η-rule*. The expression $\lambda x.(e\, x)$, where x does not occur free in the expression e, is the same as e. For instance, $\lambda x.(f\, x) = f$, that is, the function which given x as input, returns $f\, x$ as output, is the same as the function f.

Scope and variables in lambda terms. In the lambda term $\lambda x.t$ the subterm t is said to be the *scope* of λx. We say that the *binder* λ in $\lambda x.t$ binds the variable x. In what follows we will feel free to simply say 'term', instead of 'lambda term'.

An *occurrence* of a variable x is said to be *bound* in a term t iff either it is the occurrence of x in $\lambda x.t$ or it occurs in the scope of an occurrence of λx in t. An occurrence of a variable is said to be *free* iff it is not bound. A variable x is said to be *bound* (or *free*) in a term t iff there is an occurrence of x which is *bound* (or *free*) in t.

Note that in the term $t =_{def} f(x, \lambda x.g(x))$ the variable x is both bound and free, but obviously, each occurrence of x is either bound or free. The only free occurrence of x in t is the first argument of f.

Given the terms $t_1, \ldots, t_n$, for $n \geq 1$, by $vars(t_1, \ldots, t_n)$ we denote the set of variables occurring in those terms. Given term t, (i) by $freevars(t)$, or $FV(t)$, we denote the set of variables which are free in t, and (ii) by $boundvars(t)$ we denote the set of variables which are bound in t. A term t is said to be *closed* iff $freevars(t) = \emptyset$, otherwise the term is said to be *open*.

Here is the definition of the set $FV(t)$ of the free variables of a term t.

$$\begin{array}{lll} FV(x) & = \{x\} & \text{for all variables } x \\ FV(a) & = \{\} & \text{for all constants } a \\ FV(t_1 t_2) & = FV(t_1) \cup FV(t_2) & \\ FV(\lambda x.t) & = FV(t) - \{x\} & \end{array}$$

A term is said to be *ground* iff that term does not have any occurrence of a variable.

Given a term t, where $x_1, \ldots, x_m$ are the distinct variables occurring free in t, the *closure* of t is the term $\lambda x_1, \ldots, x_m.t$.

Substitution in lambda terms. The substitution of the lambda term u for the variable x into the lambda term t is denoted by $t\,[x/u]$, and it is defined as follows (in [**5**] the authors write $[u/x]\,t$, instead of $t\,[x/u]$).

We assume that there exists a countable set V of variables. Given two variables x and y in V, by $x \not\equiv y$ we mean that x is syntactically different from y.

For all variables x, y, for all terms t, t_1, t_2, and u, we have that:

$$\begin{array}{lll} x\,[x/t] & = t & \\ a\,[x/t] & = a & \text{for all constants or variables } a \text{ different from } x \\ (t_1 t_2)[x/u] & = ((t_1[x/u])\,(t_2[x/u])) & \\ (\lambda x.t)\,[x/u] & = \lambda x.t & \\ (\lambda y.t)\,[x/u] & = \lambda y.(t\,[x/u]) & \text{if } x \not\equiv y \text{ and } (y \notin FV(u) \text{ or } x \notin FV(t)) \\ (\lambda y.t)\,[x/u] & = \lambda z.((t[y/z/])[x/u]) & \text{if } x \not\equiv y \text{ and } (y \in FV(u) \text{ and } x \in FV(t)), \quad (\dagger) \\ & & \text{where } z \text{ is a variable in } V \text{ which is not in } (t\,u) \end{array}$$

The basic ideas behind this definition of a substitution, are the following ones:
(i) a free variable in the term u should be free also in the term $t[x/u]$,
(ii) the name of a bound variable is insignificant, so that the term $\lambda x.t$ should be equal to the term $\lambda y.(t\,[x/y])$, and
(iii) the substitution only affect the free occurrences of a variable. In particular, if $x \not\equiv y$ and $x \notin FV(t)$ then $(\lambda y.t)\,[x/u] = \lambda y.(t\,[x/u]) = \lambda y.t$.

In some books, in order to make the substitution to be a deterministic operation, it is assumed that the countable set V of variables is totally ordered and, in Condition (†) above, the sentence: 'z is the *first* variable in V which is not in $(t\,u)$' is used, instead of: 'z is a variable in V which is not in $(t\,u)$'.

This notion of substitution is used throughout the book and, in particular, in the Local Model Checker of Section 5 on page 295 (see the clauses for the predicate `subst(vax(X),V,T,NT)` which denotes the substitution of the value (or term) `V` for the variable `X` into the term `T`, thereby getting the new term `NT`).

Function composition and iterated composition. Given two functions f and g, their *composition*, denoted $f \circ g$, is the function, if any, such that for all x, $(f \circ g)(x) = f(g(x))$. Obviously, for the composition of f and g to be defined we need the range of g to be included in the domain of f.

Given a function $f : D \to D$, we define for any $n \in \omega$, the function f^n, called the *n-iterated composition* of the function f, as follows: for all $d \in D$,

$f^0(d) = d$

$f^{n+1}(d) = f(f^n(d))$

Thus, f^0 is the identity function and $f^1 = f$. Obviously, we have that for all $d \in D$, $f(f^n(d)) = f^n(f(d))$.

Function updating. By $f[y_0/x_0]$ we mean the function which is equal to f, except that in x_0 where it takes the value y_0, that is:

$$f[y_0/x_0]\,(x) =_{def} \begin{cases} y_0 & \text{if } x = x_0 \\ f(x) & \text{if } x \neq x_0 \end{cases}$$

Binary relations. Given a set A, the identity binary relation on A, denoted I_A, is the following set of pairs: $\{\langle a, a\rangle \mid a \in A\}$.

Given the sets A, B, and C, and the binary relations $\rho \subseteq A \times B$ and $\sigma \subseteq B \times C$, the relational composition of ρ and σ, denoted $\rho\,;\sigma$, is the set

$\{\langle a, c\rangle \mid \exists b \in B.\, \langle a, b\rangle \in \rho \wedge \langle b, c\rangle \in \sigma\}$

which is a subset of $A \times C$.

Given a binary relation ρ, ρ^+ denotes its transitive closure and ρ^* denotes its reflexive, transitive closure.

Recursive sets and recursive enumerable sets. Let us consider a finite or denumerable alphabet Σ. Given a set $A \subseteq \Sigma^*$, we say that A is *recursive* iff there exists a Turing Machine M such that: for each word $w \in \Sigma^*$, (i) M terminates, and (ii) M accepts w iff $w \in A$.

We say that $A \subseteq \Sigma^*$ is *recursive enumerable* (r.e., for short) iff there exists a Turing Machine M such that for each word $w \in \Sigma^*$, M accepts w iff $w \in A$.

We say that A *can be enumerated* (possibly with repetition) iff there exists a Turing Machine M such that: (i) for each natural number n, returns an element of A, and (ii) for each element $a \in A$, there exists a natural number n such that M on input n returns a. In this case we say that the Turing Machine M provides an *enumeration* of the set A.

One can show that a set $A \subseteq \Sigma^*$ is recursive enumerable iff either A is empty or A can be enumerated.

CHAPTER 2

Introduction to Operational and Denotational Semantics

In this chapter we present a few examples of the operational semantics and the denotational semantics of simple languages. In particular, we present: (i) the structural operational semantics for arithmetic expressions (Section 1), (ii) the operational semantics based on transition systems for an imperative language (Section 2 on page 25) and a functional language (Section 3 on page 27), and (iii) the denotational semantics for binary strings (Section 4 on page 30).

1. Structural Operational Semantics

One can specify the operational semantics of arithmetic expressions by providing rules for their evaluation. Following the style of the so called *structured operational semantics rules* à la Plotkin [**17**], the evaluation rules can take the form of deduction rules as we now specify.

Let us assume the following syntax of the set **AddAexp** of the *additive arithmetic expressions*.

$$n \in N^{\geq 0} \qquad N^{\geq 0} ::= \{0, 1, \ldots\} \qquad \text{(natural numbers)}$$

$$e \in \mathbf{AddAexp} \qquad e \quad ::= n \mid (e + e)$$

We introduce the following three deduction rules (actually, rule $R3$ is an axiom because it has no premises) which allow us to establish whether or not the relation $\rightarrow \subseteq \mathbf{AddAexp} \times \mathbf{AddAexp}$ holds between any two arithmetic expressions.

$R1.$ $$\frac{e \rightarrow e_1}{(e + e_2) \rightarrow (e_1 + e_2)}$$

$R2.$ $$\frac{e \rightarrow e_1}{(n + e) \rightarrow (n + e_1)}$$

$R3.$ $(n_1 + n_2) \rightarrow n$ where n is the sum of n_1 and n_2

We say that the arithmetic expression e *evaluates to* n (or the semantics value of e is n), and we write $e \rightarrow^* n$, where $\rightarrow^*$ is the reflexive, transitive closure of $\rightarrow$, iff $e \rightarrow^* n$ may be deduced using the rules $R1$, $R2$, and $R3$. Note that according to those rules, given any arithmetic expression, in any of its subexpressions, we first evaluate the left summand and then the right summand.

For instance, the evaluation relation $((1 + (2 + 3)) + (4 + 5)) \rightarrow^* 15$ holds because we have the following deductions (α), (β), (γ), and (δ) (over the right arrow $\rightarrow$ we indicate the deduction rule which has been applied):

$$(\alpha)\qquad \frac{\dfrac{(2+3) \overset{R3}{\to} 5}{(1+(2+3)) \overset{R2}{\to} (1+5)}}{((1+(2+3))+(4+5)) \overset{R1}{\to} ((1+5)+(4+5))}$$

$$(\beta)\qquad \frac{(1+5) \overset{R3}{\to} 6}{((1+5)+(4+5)) \overset{R2}{\to} (6+(4+5))}$$

$$(\gamma)\qquad \frac{(4+5) \overset{R3}{\to} 9}{(6+(4+5)) \overset{R2}{\to} (6+9)}$$

$$(\delta)\qquad (6+9) \overset{R3}{\to} 15$$

Here is a different set of deduction rules for establishing the relation $\to \subseteq \mathbf{AddAexp} \times \mathbf{AddAexp}$:

$$R1.\qquad \frac{e \to e_1}{(e + e_2) \to (e_1 + e_2)}$$

$$R2.1\qquad \frac{e \to e_2}{(e_1 + e) \to (e_1 + e_2)}$$

$$R3.\qquad (n_1 + n_2) \to n \quad \text{where } n \text{ is the sum of } n_1 \text{ and } n_2$$

Note that for any arithmetic expression e, rules $R1$, $R2$, and $R3$ allow one proof only of the relation $e \to n$ for some n, while the rules $R1$, $R2.1$, and $R3$ allow more than one proof, in general. In that sense we may say that rule $R1$, $R2$, and $R3$ force a *deterministic evaluation* of any given expression e, while rules $R1$, $R2.1$, and $R3$ do not force a deterministic evaluation. Indeed, for instance, we have both $(1+2)+(3+4) \to 3+(3+4)$ and $(1+2)+(3+4) \to (1+2)+7$.

When the evaluation is not deterministic, we have the problem of establishing whether or not the evaluation enjoys the confluence property, which we now define for an arbitrary binary relation ρ on a given set A.

DEFINITION 1.1. [**Confluence**] We say that a binary relation $\rho \subseteq A \times A$ for some set A, is *confluent* iff for every x, ℓ, $r \in A$, if $x \,\rho^*\, \ell$ and $x \,\rho^*\, r$ then there exists $z \in A$ such that $\ell \,\rho^*\, z$ and $r \,\rho^*\, z$.

Let us also consider the following properties which are necessary to establish Proposition 1.5 below. These properties will be considered again in the context of the rewriting systems in Section 9 on page 39.

DEFINITION 1.2. [**Termination**] We say that a binary relation $\rho \subseteq A \times A$ for some set A, *terminates* iff there is no infinite sequence $a_0, a_1, a_2, \ldots$ of elements in A such that for all $i \geq 0$, $a_i \, \rho \, a_{i+1}$.

If ρ terminates we also say that ρ is *terminating*, or *strongly terminating*, or *noetherian.*

DEFINITION 1.3. [**Normal Form**] Given a binary relation $\rho \subseteq A \times A$ for some set A, we say that v is a *normal form* of $a \in A$ iff $a \, \rho^* \, v$ and there is no element $v_1 \in A$ such that $v \, \rho \, v_1$.

THEOREM 1.4. [**Uniqueness of Normal Form**] Consider a binary relation $\rho \subseteq A \times A$ for some set A, such that ρ is confluent and terminating. For all a, ℓ, $r \in A$, if (i) $a \, \rho^* \, \ell$, (ii) $a \, \rho^* \, r$, (iii) it does not exist $\ell_1 \in A$ such that $\ell \, \rho \, \ell_1$, and (iv) it does not exist $r_1 \in A$ such that $r \, \rho \, r_1$, then $\ell = r$.

PROOF. By the confluence property, if (i) and (ii) hold then there exists $z \in A$ such that $\ell \, \rho^* \, z$ and $r \, \rho^* \, z$. By (iii) and (iv) we have that $\ell = z$ and $r = z$. Thus, $\ell = r$. □

We have the following proposition which states that the structural operational semantics defines a function, that is, for all additive arithmetic expression e, for all natural numbers n_1 and n_2, *if* $e \rightarrow^* n_1$ and $e \rightarrow^* n_2$ *then* $n_1 = n_2$.

PROPOSITION 1.5. [**Functionality of the Structural Operational Semantics**] The evaluation relation $\rightarrow$ defined by the deduction rules $R1$, $R2$, and $R3$, is confluent, terminating, and enjoys the uniqueness of normal form property. The same properties hold for the evaluation relation, also denoted $\rightarrow$, defined by the deduction rules $R1$, $R2.1$, and $R3$.

2. Operational Semantics: The SMC Machine

In this section we specify the operational semantics of a simple imperative language using a transition system following the approach suggested by Peter Landin [**8**].

In Section 3 on page 27 we will provide the operational semantics of a more complex programming language where recursive function calls are allowed.

Here are the syntactic domains of the simple imperative language we now consider.

$n \in N$ (Integer Numbers)	$N = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
$X \in \mathbf{Loc}$ (Locations or Memory Addresses)	$\mathbf{Loc} = \{X_0, X_1, \ldots, X_i, \ldots\}$
$\mathbf{op} \in \mathbf{Aop}$ (Arithmetic Operators)	$\mathbf{Aop} = \{+, -, \times\}$
$\mathbf{rop} \in \mathbf{Rop}$ (Relational Operators)	$\mathbf{Rop} = \{<, \leq, =, \geq, >\}$
$\mathbf{bop} \in \mathbf{Bop}$ (Binary Boolean Operators)	$\mathbf{Bop} = \{\wedge, \vee, \Rightarrow\}$

Let **Operators** denote the set $\{+, -, \times\} \cup \{<, \leq, =, \geq, >\} \cup \{\neg, \wedge, \vee, \Rightarrow\}$ of the arithmetic, relational, and boolean operators. All operators have arity 2, except the negation operator $\neg$ which has arity 1.

We also have the following derived syntactic domains.

$$
\begin{array}{lll}
a, a_1, a_2 \in \mathbf{Aexp} & a ::= n \mid X \mid a_1 \,\mathbf{op}\, a_2 & \text{(Arithmetic expressions)} \\
b, b_1, b_2 \in \mathbf{Bexp} & b ::= \mathbf{true} \mid \mathbf{false} \mid a_1 \,\mathbf{rop}\, a_2 & \text{(Boolean expressions)} \\
 & \quad \mid \neg b \mid b_1 \,\mathbf{bop}\, b_2 & \\
p, p_0, p_1, p_2 \in \mathbf{Com} & p ::= \mathbf{skip} \mid X := a \mid p_1; p_2 & \text{(Commands)} \\
 & \quad \mid \mathbf{if}\ b\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2 & \\
 & \quad \mid \mathbf{while}\ b\ \mathbf{do}\ p_0 &
\end{array}
$$

Let **Constants** denote the set $N \cup \{\mathbf{true}, \mathbf{false}\}$, and let **Phrases** denote the set $\mathbf{Aexp} \cup \mathbf{Bexp} \cup \mathbf{Com}$.

The operational semantics is specified as a *transition relation*, denoted $\longrightarrow$, subset of $\mathit{State} \times \mathit{State}$, where an element of the set *State* is a triple of the form $\langle S, M, C\rangle$, where:

(i) S, called *value stack*, is a stack whose elements are taken from the set **Phrases**,

(ii) M, called *memory*, is a function from the natural numbers $N^{\geq 0}$ to **Constants**, and

(iii) C, called *control stack*, is a stack whose elements are taken from the set **Phrases** $\cup$ **Operators** $\cup \{\neg, \mathbf{assign}, \mathbf{if}, \mathbf{while}\}$. (Note that $\mathbf{skip} \in \mathbf{Phrases}$.)

Both the value stack and the control stack are assumed to grow to the left and, thus, $a \,.\, s$ denote the stack s after pushing the element a on top of it. By $M[n/i]$ we denote the memory M whose location i contains the value n. By $M(i)$ we denote the value stored in the location X_i in the memory M. Thus, $M[n/i](i) = n$.

Here are the rules for the transition relation $\longrightarrow$. They are said to specify the *SMC machine*.

*R*1. $\langle S, M, c \,.\, C\rangle \longrightarrow \langle c \,.\, S, M, C\rangle$ for each c in **Constants**

*R*2. $\langle S, M, X_i \,.\, C\rangle \longrightarrow \langle M(i) \,.\, S, M, C\rangle$ for each location $X_i \in \mathbf{Loc}$

*R*3. $\langle S, M, \mathbf{skip} \,.\, C\rangle \longrightarrow \langle S, M, C\rangle$

*R*4. $\langle S, M, (X_i := a) \,.\, C\rangle \longrightarrow \langle X_i \,.\, S, M, a \,.\, \mathbf{assign} \,.\, C\rangle$

*R*5. $\langle S, M, (c_1; c_2) \,.\, C\rangle \longrightarrow \langle S, M, c_1 \,.\, c_2 \,.\, C\rangle$

*R*6. $\langle S, M, (\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) \,.\, C\rangle \longrightarrow \langle c_2 \,.\, c_1 \,.\, S, M, b \,.\, \mathbf{if} \,.\, C\rangle$

*R*7. $\langle S, M, (\mathbf{while}\ b\ \mathbf{do}\ c_0) \,.\, C\rangle \longrightarrow \langle c_0 \,.\, b \,.\, S, M, b \,.\, \mathbf{while} \,.\, C\rangle$

*R*8. $\langle S, M, \neg\, b \,.\, C\rangle \longrightarrow \langle S, M, b \,.\, \neg \,.\, C\rangle$

*R*9. $\langle b \,.\, S, M, \neg \,.\, C\rangle \longrightarrow \langle b' \,.\, S, M, C\rangle$

where b' is the semantic value which is the negation of b, that is, if b is **true** (or **false**) then b' is **false** (or **true**, respectively).

*R*10. $\langle S, M, (a_1 \,\mathbf{op}\, a_2) \,.\, C\rangle \longrightarrow \langle S, M, a_1 \,.\, a_2 \,.\, \mathbf{op} \,.\, C\rangle$

and, analogously, for each relational operator **rop** and binary boolean operator **bop**.

$R11.$ $\langle m \centerdot n \centerdot S, M, \mathbf{op} \centerdot C\rangle \longrightarrow \langle (n\ op\ m) \centerdot S, M, C\rangle$

where *op* is the semantic operator corresponding to the arithmetic operator **op**, and, analogously, for each relational operator **rop** and binary boolean operator **bop**.

$R12.$ $\langle n \centerdot X_i \centerdot S, M, \mathbf{assign} \centerdot C\rangle \longrightarrow \langle S, M[n/i], C\rangle$

$R13.$ $\langle \mathbf{true} \centerdot c_2 \centerdot c_1 \centerdot S, M, \mathbf{if} \centerdot C\rangle \longrightarrow \langle S, M, c_1 \centerdot C\rangle$

$R14.$ $\langle \mathbf{false} \centerdot c_2 \centerdot c_1 \centerdot S, M, \mathbf{if} \centerdot C\rangle \longrightarrow \langle S, M, c_2 \centerdot C\rangle$

$R15.$ $\langle \mathbf{true} \centerdot c_0 \centerdot b \centerdot S, M, \mathbf{while} \centerdot C\rangle \longrightarrow \langle S, M, (c_0; \mathbf{while}\ b\ \mathbf{do}\ c_0) \centerdot C\rangle$

$R16.$ $\langle \mathbf{false} \centerdot c_0 \centerdot b \centerdot S, M, \mathbf{while} \centerdot C\rangle \longrightarrow \langle S, M, \mathbf{skip} \centerdot C\rangle$

Note that the relation $\longrightarrow$ is deterministic, that is, for all triples t_1, t_2, t_3 in *State*, if $t_1 \longrightarrow t_2$ and $t_1 \longrightarrow t_3$ then $t_2 = t_3$.

We may also define an evaluation function, named *eval*, from **Com** $\times$ *Memory* to *Memory*, where *Memory* is the set of all functions from $N^{\geq 0}$ to **Constants**, as follows: for all commands c and memories M and M',

$eval(c, M) = M'$ iff $\langle nil, M, c \centerdot nil\rangle \longrightarrow^* \langle nil, M', nil\rangle$

where *nil* denotes the empty value stack and the empty control stack and $\longrightarrow^*$ denotes the reflexive, transitive closure of $\longrightarrow$. Obviously, the function *eval* is a partial function because (the rewriting induced by) the relation $\longrightarrow$ may not terminate (see Definition 1.2 on page 25).

Note that the syntax of our simple imperative language can be expressed via a context-free grammar while, in general, in order to express the semantics of our language, we need a type 0 grammar. The need for a type 0 grammar is due to the fact that in our language we can denote any Turing computable function from $N^{\geq 0}$ to $N^{\geq 0}$.

3. Operational Semantics: The SECD Machine

In this section we will give the operational semantics of a recursive Algol-like language following Peter Landin's approach [**8**]. As in Section 2 on page 25, the operational semantics is based on a transition system from old tuples to new tuples.

Let us consider a language with the following basic sets.

$n \in N$ (Integer Numbers)	$N = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
$x \in \mathbf{Var}$ (Variables)	$\mathbf{Var} = \{x_0, x_1, \ldots, x_i, \ldots\}$
$f \in \mathbf{Fvar}$ (Function Variables)	$\mathbf{Fvar} = \{f_0, f_1, \ldots, f_i, \ldots\}$
$\mathbf{op} \in \mathbf{Aop}$ (Arithmetic Operators)	$\mathbf{Aop} = \{+, -, \times\}$

We have the following derived set of *terms*.

$$t, t_0, t_1, t_2, t_{a_i} \in \mathbf{Term} \qquad t ::= n \mid x \mid t_1\, \mathbf{op}\, t_2 \mid \mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \mid f_i(t_1, \ldots, t_{a_i})$$

where when evaluating the term **if** t_0 **then** t_1 **else** t_2, we evaluate the left arm if $t_0 = 0$ and the right arm if $t_0 \neq 0$. This convention avoids the introduction of the boolean values **true** and **false** and we can do with integer numbers only.

We also have a set of declarations of the form

$$\begin{cases} f_1(x_1, \ldots, x_{a_1}) & = & d_1 \\ \qquad \vdots & & \\ f_k(x_1, \ldots, x_{a_k}) & = & d_k \end{cases}$$

where the d_i's are terms. Each declaration is related to a function variable f_i in **Fvar**. We assume that all the declarations are well-formed. In particular, we assume that, for $i = 1, \ldots, k$, all variables occurring in d_i belong to the set $\{x_1, \ldots, x_{a_i}\}$.

For instance, for the familiar Fibonacci function we have the following declaration:

$$\begin{aligned} \mathit{fib}(x) = {} & \textbf{if } x \textbf{ then } 1 \textbf{ else} \\ & \textbf{if } x-1 \textbf{ then } 1 \textbf{ else } \mathit{fib}(x-1) + \mathit{fib}(x-2) \end{aligned}$$

The operational semantics is specified as a *transition relation*, denoted $\longrightarrow$, subset of $\mathit{State} \times \mathit{State}$, where an element of the set *State* is a 4-tuple of the form $\langle S, E, C, D \rangle$, where:

(i) S, called *value stack*, is a stack whose elements are taken from the set **Term**,

(ii) E, called *environment*, is a list of *bindings*, that is, a list of pairs and each pair is of the form $[x, n]$,

(iii) C, called *control stack*, is a stack whose elements are taken from the set **Term** $\cup$ **Fvar** $\cup$ **Aop** $\cup$ {**if**}, and

(iv) D, called *dump*, is a stack whose elements are, recursively, 4-tuples of the form $\langle S, E, C, D \rangle$.

For all variables $x \in \mathbf{Var}$, for all integers $n \in N$, the pair $[x, n]$ is said to bind x to n.

All stacks are assumed to grow to the left and, thus, $a \centerdot s$ denote the stack s after pushing the element a on top of it.

In the definition of the operational semantics we use the following auxiliary constant *nil* and functions *lookup* and *decl*:

(i) *nil* denotes either the empty stack S, or the empty environment E, or the empty control C, or the empty dump D (the context will tell the reader which of those empty structures is denoted by any given occurrence of *nil*);

(ii) $\mathit{lookup}(x, E)$, which given a variable $x \in \mathbf{Var}$ and an environment E, returns the integer in N to which x is bound;

(iii) $\mathit{decl}(f_i)$, which given a function variable $f_i \in \mathbf{Fvar}$, returns the term which occurs on the right hand side of the declaration of f_i.

Here are the rules for the transition relation $\longrightarrow$. They are said to specify the *SECD machine*.

These rules define a *call by-value* semantics in the sense that, before applying a function, we evaluate all its arguments. As already mentioned, when evaluating the term **if** t_0 **then** t_1 **else** t_2 we first evaluate the subterm t_0 and then the left arm t_1 if $t_0 = 0$, and the right arm t_2 if $t_0 \neq 0$.

For reasons of simplicity, we have assumed that the declarations of the function variables in **Fvar** are kept outside the 4-tuple $\langle S, E, C, D\rangle$. Indeed, those declarations are accessed via the *decl* function, so that, for instance, if we have the declaration $f_i(x_1, \ldots, x_{a_i}) = d_i$ then $decl(f_i) = d_i$. (Alternatively, one could have stored all the declarations in the environment E at the expense of replicating those fixed declarations any time a new dump D is constructed.)

$R1.\ \langle S, E, n \centerdot C, D\rangle \longrightarrow \langle n \centerdot S, E, C, D\rangle \quad$ for each $n \in N$

$R2.\ \langle S, E, x \centerdot C, D\rangle \longrightarrow \langle lookup(x, E) \centerdot S, E, C, D\rangle \quad$ for each variable $x \in \mathbf{Var}$

$R3.\ \langle S, E, (t_1\ \mathbf{op}\ t_2) \centerdot C, D\rangle \longrightarrow \langle t_2 \centerdot t_1 \centerdot S, E, \mathbf{op} \centerdot C, D\rangle$

$R4.\ \langle m \centerdot n \centerdot S, E, \mathbf{op} \centerdot C, D\rangle \longrightarrow \langle (n\ op\ m) \centerdot S, E, C, D\rangle$

where *op* is the semantic operator corresponding to the arithmetic operator **op**

$R5.\ \langle S, E, (\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2) \centerdot C, D\rangle \longrightarrow \langle t_2 \centerdot t_1 \centerdot S, E, t_0 \centerdot \mathbf{if} \centerdot C, D\rangle$

$R6.\ \langle 0 \centerdot t_2 \centerdot t_1 \centerdot S, E, \mathbf{if} \centerdot C, D\rangle \longrightarrow \langle S, E, t_1 \centerdot C, D\rangle$

$R7.\ \langle n \centerdot t_2 \centerdot t_1 \centerdot S, E, \mathbf{if} \centerdot C, D\rangle \longrightarrow \langle S, E, t_2 \centerdot C, D\rangle \quad$ for $n \neq 0$

$R8.\ \langle S, E, f_i(t_1, \ldots, t_{a_i}) \centerdot C, D\rangle \longrightarrow \langle S,\ E,\ t_1 \centerdot \ldots \centerdot t_{a_i} \centerdot f_i \centerdot C,\ D\rangle \quad$ for each $f_i \in \mathbf{Fvar}$

$R9.\ \langle n_{a_i} \centerdot \ldots \centerdot n_1 \centerdot S,\ E,\ f_i \centerdot C,\ D\rangle$
$\longrightarrow \langle nil,\ [x_1, n_1] \centerdot \ldots \centerdot [x_{a_i}, n_{a_i}] \centerdot nil,\ decl(f_i) \centerdot nil,\ \langle S, E, C, D\rangle\rangle$

$R10.\ \langle n \centerdot S,\ E,\ nil,\ \langle S', E', C', D'\rangle\rangle \longrightarrow \langle n \centerdot S', E', C', D'\rangle$

As for the SMC machine, we have that the relation $\longrightarrow$ is deterministic, that is, for all 4-tuples t_1, t_2, t_3 in *State*, if $t_1 \longrightarrow t_2$ and $t_1 \longrightarrow t_3$ then $t_2 = t_3$.

We may also define for any given set of declarations, an evaluation function, named *eval*, from **Term** to N as follows: for all terms t without occurrences of variables, for all $n \in N$,

$eval(t) = n$ iff there exists an environment E such that
$\langle nil, nil, t \centerdot nil, nil\rangle \longrightarrow^* \langle n, E, nil, nil\rangle$

where $\longrightarrow^*$ denotes the reflexive, transitive closure of $\longrightarrow$. Obviously, the function *eval* is a partial function because (the rewriting induced by) the relation $\longrightarrow$ may not terminate (see Definition 1.2 on page 25).

Exercise 3.1. Provide the transition relation of the SECD machine for the call-by-name regime in which the arguments of a function are *not* evaluated before the application of the function itself. The reader may want to look first at the operational semantics that we will give on Section 4 on page 173.

Exercise 3.2. Write a Prolog program which implements the SECD machine.

4. Denotational Semantics for the Evaluation of Binary Numerals

In this section we give a simple example of how to define the denotational semantics of binary strings as natural numbers. We follow the technique proposed by Scott and Strachey in 1971 (see, for instance, [**18**]).

We have the syntactic domain **Bin** of the binary strings, where $b \in \mathbf{Bin}$:

$$b ::= 0 \mid 1 \mid b\,0 \mid b\,1$$

with the concatenation operation which we have denoted by juxtaposition. We choose the semantic domain to be the set of the natural numbers $N^{\geq 0}$ with the nullary constructor *zero*, denoted 0, the unary constructor *successor*, denoted s, and the multiplication operation, denoted $\times$. Thus, the natural numbers will be denoted by 0, $s(0)$, $s(s(0))$, ...

The denotational semantics is given by the semantic function $[\![\_]\!]$ (pronounced fat square brackets) from **Bin** to $N^{\geq 0}$ defined as follows, where $b \in \mathbf{Bin}$:

$$\begin{aligned}
[\![0]\!] &= 0 \\
[\![1]\!] &= s(0) \\
[\![b\,0]\!] &= s(s(0)) \times [\![b]\!] && \text{(recall that } [\![b\,0]\!] \text{ is } 2\times[\![b]\!]) \\
[\![b\,1]\!] &= s\big(s(s(0)) \times [\![b]\!]\big) && \text{(recall that } [\![b\,1]\!] \text{ is } (2\times[\![b]\!])+1)
\end{aligned}$$

Note that in the equation $[\![0]\!] = 0$, the 0 on the left is a string belonging to the syntactic domain **Bin**, while the 0 on the right is a natural number belonging to the semantic domain $N^{\geq 0}$.

The definition of the function $[\![\_]\!]$ provided by the above four equations specifies a single mathematical function from **Bin** to $N^{\geq 0}$ because that definition is given by induction on the structure of the elements of **Bin**. (Actually, $[\![\_]\!]$ is a *homomorphism* from **Bin** to $N^{\geq 0}$.) This fact is a consequence of the Recursion Theorem (see Section 2 on page 74).

CHAPTER 3

Introduction to Sorted Algebras and Rewriting Systems

In this chapter we introduce the notion of a *sorted algebra* and we study some theories based on equations between terms. We closely follow the presentation given in [**7**]. We also present the basic concepts on *rewriting systems* and termination proofs.

1. Syntax of Sorted Algebras

First, we define the syntax of a sorted algebra, that is, an algebra with so called typed operators.

We consider a finite set $\mathbf{S}$ of *sorts*. They are a finite collection of identifiers. For instance, $\mathbf{S} = \{\mathbf{integer}, \mathbf{boolean}\}$.

From a given set of sorts we get a set of *types*, whose generic element $\mathbf{t}$ is of the form:

$\mathbf{t} ::= \mathbf{s} \mid \mathbf{s}_1 \times \ldots \times \mathbf{s}_n \to \mathbf{s}$

where $\mathbf{s}, \mathbf{s}_1, \ldots, \mathbf{s}_n \in \mathbf{S}$. Thus, a type is a sequence of one or more sorts. The rightmost sort $\mathbf{s}$ of a type $\mathbf{t}$ of the form either $\mathbf{s}$ or $\mathbf{s}_1 \times \ldots \times \mathbf{s}_n \to \mathbf{s}$ is said to be the sort of any element generated by any operator of type $\mathbf{t}$.

A *typed signature* (or *signature*, for short) is a set Σ of *typed operators*, that is, operators with types. A type signature defines the syntax of a *sorted algebra*, or *algebra*, for short.

We will assume that every given signature is *sensible*, that is, every sort occurring in the type of an operator in Σ, is generated by an operator in Σ.

For instance, given the set of sorts $\mathbf{S} = \{\mathbf{integer}, \mathbf{boolean}\}$ and the typed signature $\Sigma = \{0, s, +, \text{True}, \text{False}, eq\}$ with the following typed operators:

$$\begin{array}{lcl} 0 & : & \mathbf{integer} \\ s & : & \mathbf{integer} \to \mathbf{integer} \\ + & : & \mathbf{integer} \times \mathbf{integer} \to \mathbf{integer} \\ \text{True} & : & \mathbf{boolean} \\ \text{False} & : & \mathbf{boolean} \\ eq & : & \mathbf{integer} \times \mathbf{integer} \to \mathbf{boolean} \end{array}$$

we have that Σ is a sensible signature.

Given the set of sorts $\mathbf{S} = \{\mathbf{integer}, \mathbf{boolean}\}$ and the signature $\Sigma = \{\text{True}, \text{False}, eq\}$ with the following typed operators:

$$\begin{array}{lcl} \text{True} & : & \mathbf{boolean} \\ \text{False} & : & \mathbf{boolean} \\ eq & : & \mathbf{integer} \times \mathbf{integer} \to \mathbf{boolean} \end{array}$$

we have that Σ is *not* a sensible signature, because no element of sort **integer** can be generated by an operator in Σ.

2. Semantics of Sorted Algebras

Now we consider the semantics of a sorted algebra by introducing the notion of a Σ-algebra.

A Σ-*algebra* is a pair $\langle \mathcal{A}, \mathcal{F} \rangle$, where: (i) $\mathcal{A}$ is a family of sets such that for each sort $\mathbf{s} \in \mathbf{S}$, there exists a set $\mathcal{A}_{\mathbf{s}}$, called the *carrier* of sort $\mathbf{s}$, and (ii) $\mathcal{F}$ is a family of functions such that for each $F \in \Sigma$,
(ii.1) if the type of F is $\mathbf{s}$ then $\mathcal{F}_F$ is an element of $\mathcal{A}_{\mathbf{s}}$, and
(ii.2) if the type of F is $\mathbf{s}_1 \times \ldots \times \mathbf{s}_n \rightarrow \mathbf{s}$ then $\mathcal{F}_F$ is a *total* function in $(\mathcal{A}_{\mathbf{s}_1} \times \ldots \times \mathcal{A}_{\mathbf{s}_n}) \rightarrow \mathcal{A}_{\mathbf{s}}$, where, as usual, $S_1 \times S_2$ denotes the cartesian product of the sets S_1 and S_2, and $S_1 \rightarrow S_2$ denotes the set of all functions from S_1 to S_2.

For instance, given the set $\{\mathbf{integer}, \mathbf{boolean}\}$ of sorts and the signature $\Sigma = \{0, s, +, \text{True}, \text{False}, eq\}$, we have the following Σ-algebra called *Arith*.

The Σ-algebra *Arith*

$\mathcal{A}_{\mathbf{integer}}$ is N (that is, the set of the natural numbers),
$\mathcal{A}_{\mathbf{boolean}}$ is $\{true, false\}$,
0 is the natural number $0 \in N$,
s is the usual successor function *succ* from N to N,
$+$ is the usual function *sum* from $N \times N$ to N,
True is *true*,
False is *false*, and
$eq(m, n)$ is the function such that if $m = n$ then *true* else *false*.

3. Initial (or Free) Algebras

Among all possible Σ-algebras, there is a Σ-algebra, called the *free* Σ-*algebra*, or the *initial* Σ-*algebra*. In this algebra, (i) for each sort $\mathbf{s}$, the carrier of sort $\mathbf{s}$ is the set of all *finite ordered trees* with nodes labeled by elements in Σ, and (ii) the functions associated with the operators are defined as follows: for each operator $F \in \Sigma$,
(ii.1) if the type of the operator F is $\mathbf{s}$ then $\mathcal{F}_F$ is F, that is, a leaf with label F, and
(ii.2) if the type of operator F is $\mathbf{s}_1 \times \ldots \times \mathbf{s}_n \rightarrow \mathbf{s}$ then $\mathcal{F}_F$ is $\lambda x_1, \ldots, x_n.\, F(x_1, \ldots, x_n)$, that is, a node with label F with n child nodes that are recursively defined from the n arguments $x_1, \ldots, x_n$, respectively.

The union of the carriers of the free Σ-algebra, denoted $\mathcal{T}(\Sigma)$, is called the set of *ground terms* (or *words*, or *abstract syntax trees*) over Σ.

For instance, given the set $\{\mathbf{integer}, \mathbf{boolean}\}$ of sorts and the signature $\Sigma = \{0, s, +, \text{True}, \text{False}, eq\}$, the free Σ-algebra is defined as follows:

(i) $\mathcal{A}_{\mathbf{integer}}$ is the set *IntegerTerms*, whose generic element t is of the form:

$t ::= 0 \mid s(t) \mid t_1 + t_2$

where $t, t_1, t_2 \in$ *IntegerTerms*,

(ii) $\mathcal{A}_{\mathbf{boolean}}$ is the set *BooleanTerms*, whose generic element b is of the form:

$b ::= \text{True} \mid \text{False} \mid eq(t_1, t_2)$

where $t_1, t_2 \in$ *IntegerTerms*, and the functions associated with the operators in Σ are the following ones.

- The function associated with the nullary operator 0 returns a tree which is a leaf with label 0.
- The function associated with the unary operator s returns, for every argument t, a root node with label s and a single child node which is the tree returned by the function associated with the top operator of t.
- The function associated with the binary operator + returns, for all arguments t_1 and t_2, a root node labeled by + with two child nodes which are returned by the function associated with the top operators of t_1 and t_2, respectively.

Analogous definitions specify the functions associated with the operators True, False, and *eq*.

The expert reader will note that the union of the carriers of the free Σ-algebra is the so called Herbrand Universe generated from the symbols in Σ.

Given the signature $\Sigma = \{0, s, +, \text{True}, \text{False}, eq\}$, the set $\mathcal{T}(\Sigma)$ is *IntegerTerms* $\cup$ *BooleanTerms*. For instance, the following ground terms, viewed as trees, belong to $\mathcal{T}(\Sigma)$: (i) $0{+}s(0)$, (ii) $s(0{+}s(s(0)))$, and (iii) $eq(s(0){+}0,\ 0)$.

4. Variables

We may have variables occurring in terms as we now specify. We consider for each sort $\mathbf{s} \in \mathbf{S}$, a countable set $\mathcal{V}_\mathbf{s}$ of variables of type $\mathbf{s}$. We then consider the set $\mathcal{T}(\Sigma \cup \mathcal{V})$, where $\mathcal{V} = \bigcup_{\mathbf{s} \in \mathbf{S}} \mathcal{V}_\mathbf{s}$, made out of the terms which can be constructed by considering the typed variables as typed operators of arity 0.

For instance, $0 + s(x)$ is a term in $\mathcal{T}(\Sigma \cup \mathcal{V})$, where $\Sigma = \{0, s, +, \text{True}, \text{False}, eq\}$ and $x \in \mathcal{V}_{\mathbf{integer}}$.

A term without variables is said to be a *ground term*.

5. Morphisms

Given two Σ-algebras $\langle \mathcal{A}_1, \mathcal{F}_1 \rangle$ and $\langle \mathcal{A}_2, \mathcal{F}_2 \rangle$, a Σ-*morphism* (or a *morphism*, for short) is a family of total functions, one total function $h_\mathbf{s}$ for each sort $\mathbf{s} \in \mathbf{S}$, such that:
(i) $h_\mathbf{s} : \mathcal{A}_{1\mathbf{s}} \to \mathcal{A}_{2\mathbf{s}}$, and
(ii) for each operator $F \in \Sigma$ with type $\mathbf{s}_1 \times \ldots \times \mathbf{s}_n \to \mathbf{s}$, we have that:

for all $a_1, \ldots, a_n \in \mathcal{A}_{1\mathbf{s}} \times \ldots \times \mathcal{A}_{1\mathbf{s}}$, $h_\mathbf{s}(\mathcal{F}_{1\,F}(a_1, \ldots, a_n)) = \mathcal{F}_{2\,F}(h_{\mathbf{s}_1}(a_1), \ldots, h_{\mathbf{s}_n}(a_n))$.

We have the following theorems.

THEOREM 5.1. **[Initiality of the Free Algebra]** Let $\langle \mathcal{A}, \mathcal{F} \rangle$ be the initial Σ-algebra. For any other Σ-algebra $\langle \mathcal{A}', \mathcal{F}' \rangle$ there exists a unique Σ-morphism from $\langle \mathcal{A}, \mathcal{F} \rangle$ to $\langle \mathcal{A}', \mathcal{F}' \rangle$.

THEOREM 5.2. **[Universality of the Free Algebra]** Let $\langle \mathcal{A}, \mathcal{F} \rangle$ be a Σ-algebra. Every variable assignment $\nu : \mathcal{V} \to \mathcal{A}$ can be extended in a unique way to a Σ-morphism, also named ν, from $\mathcal{T}(\Sigma \cup \mathcal{V})$ to $\langle \mathcal{A}, \mathcal{F} \rangle$.

For instance, if $\nu(X) = s(0)$ then $\nu(X{+}0) = s(0){+}0$.

A Σ-morphism from $\mathcal{T}(\Sigma \cup \mathcal{V})$ to $\mathcal{T}(\Sigma \cup \mathcal{V})$ which extends a variable assignment, is called a *substitution*. An equivalent definition of a substitution will be given in Section 9 (see Definition 9.1 on page 40). In that section (i) the set of operators is denoted by F, instead of Σ, (ii) the set of variables is denoted by *Vars*, instead of $\mathcal{V}$, and (iii) the set of terms is denoted by $T(F \cup \mathit{Vars})$, instead of $\mathcal{T}(\Sigma \cup \mathcal{V})$.

Given a Σ-algebra $\langle \mathcal{A}, \mathcal{F} \rangle$, a binary relation $\sim \subseteq \mathcal{A} \times \mathcal{A}$ is said to be a Σ-*congruence over* $\mathcal{A}$ iff (i) $\forall a, b \in \mathcal{A}$ if $a \sim b$ then a and b have the same sort, and (ii) for each operator $F \in \Sigma$ with type $\mathbf{s}_1 \times \ldots \times \mathbf{s}_n \to \mathbf{s}$, we have that:

$\forall a_1, b_1, \ldots, a_n, b_n \in \mathcal{A}$, $(a_1 \sim b_1, \ldots, a_n \sim b_n) \Rightarrow \mathcal{F}_F(a_1, \ldots, a_n) \sim \mathcal{F}_F(b_1, \ldots, b_n)$.

6. Equations, Validity, Satisfiability, and Variety

A Σ-*equation* (or *equation*, for short) is a pair $\langle t_1, t_2 \rangle$ of terms in $\mathcal{T}(\Sigma \cup \mathcal{V})$ such that t_1 and t_2 have the same sort. Usually, the Σ-equation $\langle t_1, t_2 \rangle$ is written as $t_1 = t_2$.

Definition 6.1. [**Validity of Σ-equations**] Given a Σ-algebra $\langle \mathcal{A}, \mathcal{F} \rangle$ and a Σ-equation $t_1 = t_2$, we say that $t_1 = t_2$ *is valid in* $\langle \mathcal{A}, \mathcal{F} \rangle$, or $t_1 = t_2$ *holds in* $\langle \mathcal{A}, \mathcal{F} \rangle$, or $\langle \mathcal{A}, \mathcal{F} \rangle$ is a *model of* $t_1 = t_2$, and we write $\langle \mathcal{A}, \mathcal{F} \rangle \models t_1 = t_2$, iff *for every* variable assignment $\nu : \mathcal{V} \to \mathcal{A}$, we have that $\nu(t_1) = \nu(t_2)$.

A set $\mathcal{E}$ of Σ-equations holds in a Σ-algebra $\langle \mathcal{A}, \mathcal{F} \rangle$ iff every equation in $\mathcal{E}$ holds in $\langle \mathcal{A}, \mathcal{F} \rangle$.

Thus, when equations involve variables, those variables should be understood as universally quantified in front.

For instance, for the Σ-algebra *Arith* defined on page 32 we have that $Arith \models x + y = y + x$ holds, that is, the operator $+$ is commutative, because the function *sum* on the natural numbers is commutative (see Theorem 1.3 on page 60).

Definition 6.2. [**Satisfiability of Σ-equations**] Given a Σ-algebra $\langle \mathcal{A}, \mathcal{F} \rangle$ and a Σ-equation $t_1 = t_2$, we say that $t_1 = t_2$ is *satisfiable* in $\langle \mathcal{A}, \mathcal{F} \rangle$ iff *there exists* a variable assignment ν such that $\nu(t_1) = \nu(t_2)$.

We say that a Σ-equation $t_1 = t_2$ is *satisfiable in a class* $\mathcal{C}$ of Σ-algebras iff there exists a Σ-algebra $\langle \mathcal{A}, \mathcal{F} \rangle \in \mathcal{C}$ such that $t_1 = t_2$ is satisfiable in $\langle \mathcal{A}, \mathcal{F} \rangle$.

Let us consider a signature Σ and a Σ-algebra $\mathcal{A}$. For reasons of simplicity, here and in what follows the Σ-algebra $\langle \mathcal{A}, \mathcal{F} \rangle$ is also denoted by $\mathcal{A}$ only.

Let us consider the binary relation $=_{\mathcal{A}}$ defined as follows: for all $t_1, t_2 \in \mathcal{T}(\Sigma \cup \mathcal{V})$, $t_1 =_{\mathcal{A}} t_2$ iff $\mathcal{A} \models t_1 = t_2$. The relation $=_{\mathcal{A}}$ is a Σ-congruence over $\mathcal{T}(\Sigma \cup \mathcal{V})$.

Definition 6.3. [**Variety**] Given a set $\mathcal{E}$ of Σ-equations, the *variety* of $\mathcal{E}$ is the class of Σ-algebras, denoted $\mathcal{M}(\Sigma, \mathcal{E})$, or simply $\mathcal{M}(\mathcal{E})$, which are models of $\mathcal{E}$. That is, for each equation $t_1 = t_2 \in \mathcal{E}$, we have that $\mathcal{M}(\mathcal{E}) \models t_1 = t_2$ iff for each Σ-algebra $\mathcal{A}$ in $\mathcal{M}(\mathcal{E})$, $\mathcal{A} \models t_1 = t_2$ holds.

$\mathcal{M}(\mathcal{E})$ is never empty because, as we will see below, there is a Σ-algebra, called *Sort*, which is a model of every set $\mathcal{E}$ of equations. The Σ-algebra *Sort* is defined as follows: for each sort $\mathbf{s}$, the carrier of sort $\mathbf{s}$ is the singleton $\{\mathbf{s}\}$, and for each operator $F \in \Sigma$,

(1) if the type of the operator F is $\mathbf{s}$ then $\mathcal{F}_F$ is $\mathbf{s}$, and

(2) if the type of operator F is $\mathbf{s}_1 \times \ldots \times \mathbf{s}_n \to \mathbf{s}$ then $\mathcal{F}_F$ is $\lambda x_1, \ldots, x_n.\, \mathbf{s}$.

Every variable x of sort $\mathbf{s}$, that is, every $x \in \mathcal{V}_{\mathbf{s}}$, is assigned the value $\mathbf{s}$. Thus, every term in $\mathcal{T}(\Sigma \cup \mathcal{V})$ is given a value in the set $\mathbf{S}$ of sorts.

Now the Σ-algebra *Sort* is a model of every given set $\mathcal{E}$ of equations, because we have assumed that in every Σ-equation $t_1 = t_2$ the terms t_1 and t_2 should have the same sort.

Let us consider the following set $\mathcal{E}^+$ of equations for the function symbol $+$:

$\{\, x+0=x, \qquad x+s(y)=s(x+y)\,\}$ $\qquad (\mathcal{E}^+)$

Let us also consider the set $\{\mathbf{integer}, \mathbf{boolean}\}$ of sorts and the signature $\Sigma = \{0, s, +, \text{True}, \text{False}, eq\}$. We have the following Σ-algebra called $Arith1$.

The Σ-algebra $Arith1$

$\mathcal{A}_{\mathbf{integer}}$ is $N_{red} \cup N_{blue} = \{0_{red}, 1_{red}, 2_{red}, \ldots, 0_{blue}, 1_{blue}, 2_{blue}, \ldots\}$ (that is, two colored copies of the set of the natural numbers),
$\mathcal{A}_{\mathbf{boolean}}$ is $\{true, false\}$,
0 is 0_{red},
s is the usual successor function $succ$ from $N_{red} \cup N_{blue}$ to $N_{red} \cup N_{blue}$ such that color is preserved (for instance, $succ(1_{red}) = 2_{red}$ and $succ(0_{blue}) = 1_{blue}$),
$+$ is the usual function sum from $(N_{red} \cup N_{blue}) \times (N_{red} \cup N_{blue})$ to $N_{red} \cup N_{blue}$ such that the color of the *left* operand is preserved (for instance, $sum(2_{red}, 1_{blue}) = 3_{red}$),
True is $true$,
False is $false$, and
$eq(m_a, n_b)$ is the function such that if $m = n$ then $true$ else $false$ (for instance, $eq(4_{red}, 4_{blue}) = true$ and $eq(3_{red}, 4_{red}) = false$).

Both $Arith$ and $Arith1$ are models of the set $\mathcal{E}^+$ of equations.

However, we have that: $Arith1 \not\models x+y = y+x$. Indeed, for instance, we have that $sum(1_{red}, 0_{blue}) = 1_{red}$ and $sum(0_{blue}, 1_{red}) = 1_{blue}$. Thus, $\mathcal{M}(\mathcal{E}^+) \not\models x + y = y + x$.

We have that: $\mathcal{M}(\mathcal{E}^+) \models 0 + s^n(0) = s^n(0)$, for all $n \geq 0$. The easy proof by induction on n, using the Equations $\mathcal{E}^+$, is left to the reader. Note that in $Arith1$ we have that, for every $n \in N$, $s^n(0)$ is n_{red}.

However, $\mathcal{M}(\mathcal{E}^+) \not\models 0+y = y$, because we have that $Arith1 \not\models 0+y = y$. Indeed, $sum(0_{red}, 0_{blue}) = 0_{red}$.

Exercise 6.4. Show by induction that for all $m, n \geq 0$, we have that

$\mathcal{M}(\mathcal{E}^+) \models s^m(0) + s^n(0) = s^n(0) + s^m(0)$

that is, commutativity of $+$ holds for ground terms.

7. Validity Problem and Word Problem

In what follows we will consider the so called *validity problem* which consists in deciding whether or not, given a class $\mathcal{C}$ of Σ-algebras and a Σ-equation $t_1 = t_2$, for some terms t_1 and t_2 of the same sort in the free Σ-algebra $\mathcal{T}(\Sigma \cup \mathcal{V})$, we have that $\mathcal{A} \models t_1 = t_2$, for all Σ-algebras $\mathcal{A} \in \mathcal{C}$.

We will also consider the *word problem* which consists in deciding whether or not, given a Σ-algebra $\mathcal{A}$ and a Σ-equation $t_1 = t_2$, for some terms t_1 and t_2 of the same sort in the free Σ-algebra $\mathcal{T}(\Sigma)$, we have that $\mathcal{A} \models t_1 = t_2$.

Note that the terms t_1 and t_2 are *ground* terms in the word problem, while they may contain variables in the validity problem.

Obviously, the word problem is an instance of the validity problem.

DEFINITION 7.1. [**Congruence $=_{\mathcal{E}}$ Associated with a Set $\mathcal{E}$ of Equations**] Given a set $\mathcal{E}$ of Σ-equations, the *equational theory* $=_{\mathcal{E}}$ is the finest Σ-congruence over $\mathcal{T}(\Sigma \cup \mathcal{V})$ which contains, for all equations $t_1 = t_2$ in $\mathcal{E}$, for all substitutions σ, every pair $\langle \sigma(t_1), \sigma(t_2) \rangle$, which is also written as $\sigma(t_1) =_{\mathcal{E}} \sigma(t_2)$.

Recall that, given any Σ-algebra, a Σ-*congruence* is an equivalence relation on $\mathcal{T}(\Sigma \cup \mathcal{V})$ which is preserved by every operator in Σ.

Thus, an equational theory is a set of theorems each of which is an equality between two (not necessarily ground) terms.

The equations in $\mathcal{E}$ are also called the *axioms* of the equational theory $=_{\mathcal{E}}$.

For any given term t_1 and t_2 of the free Σ-algebra $\mathcal{T}(\Sigma \cup \mathcal{V})$, we have that $t_1 =_{\mathcal{E}} t_2$ holds iff it can be derived by the following proof system.

The equational theory $=_{\mathcal{E}}$

$R1.$	$\dfrac{}{t_1 =_{\mathcal{E}} t_2}$	if $t_1 = t_2 \in \mathcal{E}$
$R2.$	$\dfrac{t_1 =_{\mathcal{E}} t_2}{\sigma(t_1) =_{\mathcal{E}} \sigma(t_2)}$	for any substitution σ
$R3.$	$\dfrac{t_1 =_{\mathcal{E}} t_2}{t[t_1/p] =_{\mathcal{E}} t[t_2/p]}$	for any term $t \in \mathcal{T}(\Sigma \cup \mathcal{V})$ and any position p in t

In rule R3 of that proof system, for all terms $t, t_1 \in \mathcal{T}(\Sigma \cup \mathcal{V})$, by $t[t_1/p]$ we denote the term t whose subterm at *position* p has been replaced by the subterm t_1. We do not formally define the notion of a position here. The following properties and the following Example 7.2 will suffice.

Given a term t, the set $P(t)$ of the positions of t is a set of words, each of which is of the form, where $n \in N - \{0\}$:

$p ::= \varepsilon \mid p.n$

The position ε is the position of the top operator of the given term. Thus, for all terms $t, t_1 \in \mathcal{T}(\Sigma \cup \mathcal{V})$, we have that $t[t_1/\varepsilon]$ is t_1.

Obviously, if $p.n \in P(t)$ then: (i) $p \in P(t)$, and (ii) for all $m \in N - \{0\}$ such that $m < n$, we have that $p.m \in P(t)$.

EXAMPLE 7.2. In Figure 1 on the facing page we have depicted the term $0+s(x)$, its positions, and the term $(0+s(x))\,[(s(0)+y)\,/\,2]$, which is $0+(s(0)+y)$. □

We have the following theorem.

THEOREM 7.3. [**Birkhoff Theorem**] Given a set $\mathcal{E}$ of Σ-equations and two terms t_1 and t_2 of the free Σ-algebra $\mathcal{T}(\Sigma \cup \mathcal{V})$, we have that $\mathcal{M}(\mathcal{E}) \models t_1 = t_2$ iff $t_1 =_{\mathcal{E}} t_2$.

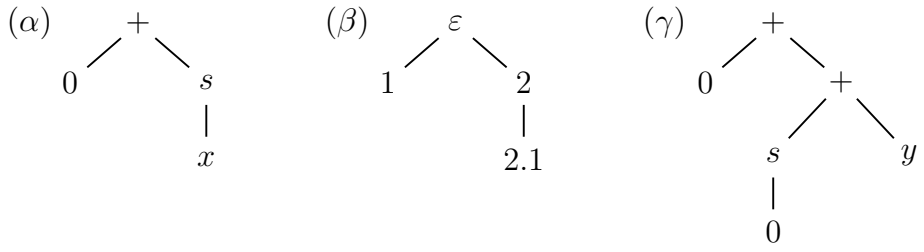


FIGURE 1. (α) The term $0+s(x)$. (β) The set $\{\varepsilon, 1, 2, 2.1\}$, depicted as a tree, of the four positions of the term $0+s(x)$. (γ) The term $(0+s(x))\,[(s(0)+y)/2]$ which denotes the term $0+s(x)$ whose subterm at position 2 has been replaced by $s(0)+y$.

Thus, given a set $\mathcal{E}$ of Σ-equations, an equation holds in the variety of $\mathcal{E}$ iff it can be derived from $\mathcal{E}$ by *substitution* (see rule $R2$ above) and *replacement* of equals by equals (see rule $R3$ above).

PROPOSITION 7.4. **[Semidecidability of the Validity Problem for a Recursive Set of Equations]** If the given set $\mathcal{E}$ of Σ-equations is recursive, then there exists a recursively enumerable procedure to decide the validity problem in the variety of $\mathcal{E}$, that is, the problem to establish whether or not, given any two terms t_1 and t_2, $\mathcal{M}(\mathcal{E}) \models t_1 = t_2$ holds is semidecidable.

PROOF. It is based on: (i) Theorem 7.3 on the preceding page, and (ii) the proof system for the equational theory $=_{\mathcal{E}}$ with the proof rules $R1$, $R2$, and $R3$ (see page 36). First, note that the sets of (i) axioms, (ii) terms, (iii) substitutions, and (iv) positions can all be enumerated (enumerable sets are defined on page 22). Also the sets of all sequences of expressions of the form (i)–(iv) can be enumerated. Now, since: (1) a substitution of many variables can be realized by a sequence of substitutions of single variables, and (2) all sequences of $\langle$*term*, *position*$\rangle$ pairs can be enumerated (using a dove-tailing technique), we have that all sequences of instances of Rule $R2$ or Rule $R3$, can be enumerated. Then, all trees whose internal nodes are instances of Rule $R2$ or Rule $R3$ and whose leaves are instances of Rule $R1$, can be enumerated. These trees denote proofs of equalities of the form $t_1 =_{\mathcal{E}} t_2$ when using the proof system for $=_{\mathcal{E}}$. As a consequence, the set of all valid equalities $t_1 =_{\mathcal{E}} t_2$ can be enumerated, and semidecidability follows. □

DEFINITION 7.5. **[Initial Σ-Algebra Generated by a Set of Equations]** Given a set $\mathcal{E}$ of Σ-equations, *the initial Σ-algebra generated by $\mathcal{E}$*, denoted $\mathcal{I}(\Sigma, \mathcal{E})$, or simply $\mathcal{I}(\mathcal{E})$, is the quotient of the initial algebra $\mathcal{T}(\Sigma)$ by the Σ-congruence $=_{\mathcal{E}} \subseteq \mathcal{T}(\Sigma \cup \mathcal{V}) \times \mathcal{T}(\Sigma \cup \mathcal{V})$, when $=_{\mathcal{E}}$ is restricted to the ground terms, that is, when$=_{\mathcal{E}}$ is restricted to $\mathcal{T}(\Sigma) \times \mathcal{T}(\Sigma)$.

There exists a unique Σ-morphism from $\mathcal{I}(\mathcal{E})$ to any Σ-algebra in $\mathcal{M}(\mathcal{E})$.

A substitution $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma)$ is said to be a *ground substitution* (see also page 40).

THEOREM 7.6. $\mathcal{I}(\mathcal{E}) \models t_1 = t_2$ iff for every *ground substitution* $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma)$ we have that $\sigma(t_1) =_{\mathcal{E}} \sigma(t_2)$.

Thus, given a set $\mathcal{E}$ of Σ-equations, the problem of checking whether or not a given Σ-equation $t_1 = t_2$ holds in $\mathcal{I}(\mathcal{E})$ is equivalent to the problem of checking whether or not all the *ground instances* of $t_1 = t_2$ hold in $\mathcal{M}(\mathcal{E})$, that is, $\mathcal{I}(\mathcal{E}) \models t_1 = t_2$ holds iff $\mathcal{M}(\mathcal{E}) \models \sigma(t_1) = \sigma(t_2)$ holds for all ground substitutions σ.

While $\mathcal{M}(\mathcal{E}) \models t_1 = t_2$ can be established by the proof system of page 36, unfortunately, there is no a similarly simple proof system for establishing whether or not $\mathcal{I}(\mathcal{E}) \models t_1 = t_2$ holds.

We have that equational reasoning is complete for solving the word problem in the initial Σ-algebra, that is, by using: (i) substitution, and (ii) replacement of equals by equals (see rules $R2$ and $R3$ on page 36) we can solve the problem of checking whether or not, given a set $\mathcal{E}$ of equations and two *ground terms* t_1 and t_2, $\mathcal{I}(\mathcal{E}) \models t_1 = t_2$ holds.

If t_1 and t_2 are not ground, we may be able to check whether or not $\mathcal{I}(\mathcal{E}) \models t_1 = t_2$ holds by using induction on the structure of $\mathcal{T}(\Sigma)$. However, there is no general induction schema which is powerful enough to establish equalities between non-ground terms.

Let us consider the signature $\Sigma = \{0, s, +, \text{True}, \text{False}, eq\}$ and the Σ-equations $\mathcal{E}^+$ (see page 35). The $\mathcal{E}^+$-equivalence classes, which are the elements of $\mathcal{I}(\mathcal{E}^+)$ of sort **integer** are the natural numbers N. Indeed, we have that:

$$
\begin{array}{lcl}
[0] & = & \{0,\ 0{+}0,\ 0{+}(0{+}0), \ldots\} \\
[s(0)] & = & \{s(0),\ 0{+}s(0),\ s(0){+}0,\ 0{+}(0{+}s(0)), \ldots\} \\
[s(s(0))] & = & \{s(s(0)),\ 0{+}s(s(0)),\ 0{+}(0{+}s(s(0))),\ s(s(0)){+}0,\ s(0){+}s(0), \ldots\} \\
\ldots &&
\end{array}
$$

where the equivalence class of the element x is denoted by $[x]$.

However, if we consider all the elements of $\mathcal{I}(\mathcal{E}^+)$, both of sort **integer** and sort **boolean**, we have that there are $\mathcal{E}^+$-equivalence classes which are *not* elements of $\{[0], [s(0)], \ldots\} \cup \{[\text{True}], [\text{False}]\}$. Indeed, in particular, it is *not* the case that $eq(0,0) =_{\mathcal{E}^+} \text{True}$, and thus, $eq(0,0) \notin [\text{True}]$. Similarly, $eq(0,0) \notin [eq(s(0), s(0))]$, $eq(0, s(0)) \notin [\text{False}]$, while $eq(0, s(0)) \in [eq(0, s(0){+}0)]$.

Exercise 7.7. Show by induction that for all $x, y \in \mathcal{T}(\Sigma)$, $\mathcal{I}(\mathcal{E}^+) \models x{+}y = y{+}x$, that is, commutativity of $+$ holds for ground terms in the initial Σ-algebra generated by $\mathcal{E}^+$.

Remark 7.8. Σ-algebras provide a mathematical understanding of the notions of *classes* and *objects* encountered in object-oriented languages such as Java. A set $\mathcal{E}$ of Σ-equations provides the semantics of the terms by associating with each term t its equivalence class, that is, the set of all terms t' such that $t =_{\mathcal{E}} t'$ (see Definition 7.1 on page 36).

8. Unification and Matching Problems

In this section we introduce the notions of unification and matching which will be useful in the sequel.

Let us first introduce the following definition. Given a term t, by $vars(t)$ we denote the set of variables occurring in t.

DEFINITION 8.1. [**Unification Modulo Equations or $\mathcal{E}$-unification**] Given a signature Σ, two terms t_1 and t_2 in $\mathcal{T}(\Sigma \cup \mathcal{V})$, and a set $\mathcal{E}$ of Σ-equations, we say that t_1 and t_2 are *$\mathcal{E}$-unifiable* (or *unifiable modulo the set $\mathcal{E}$ of Σ-equations*) iff there exists a substitution $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma \cup \mathcal{V})$ such that $\sigma(t_1) =_{\mathcal{E}} \sigma(t_2)$.

A substitution σ such that $\sigma(t_1) =_{\mathcal{E}} \sigma(t_2)$ is called an *$\mathcal{E}$-unifier* of t_1 and t_2.

We say that an $\mathcal{E}$-unifier σ of t_1 and t_2 is *away from the set W of variables* such that $vars(t_1) \cup vars(t_2) \subseteq W$ iff $V_\sigma \cap W = \emptyset$, where V_σ is the set $vars(\sigma(t_1)) \cup vars(\sigma(t_2))$.

For instance, given the equations $\mathcal{E}^+$ on page 35, the terms $x+0$ and y have the $\mathcal{E}^+$-unifier σ such that $\sigma(x)=y$ (because $y+0=y$). They also have the $\mathcal{E}^+$-unifier σ such that $\sigma(x)=\sigma(y)=0$. An $\mathcal{E}^+$-unifier away from $\{x, y\}$ is the substitution σ such that $\sigma(x)=\sigma(y)=z$.

The *unification problem* consists in deciding whether or not, given two terms t_1 and t_2 in $\mathcal{T}(\Sigma \cup \mathcal{V})$ and a set $\mathcal{E}$ of Σ-equations, there exists a substitution $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma \cup \mathcal{V})$ such that $\sigma(t_1) =_{\mathcal{E}} \sigma(t_2)$.

In particular, if $\mathcal{E}$ is the set of Peano axioms, the unification problem is equivalent to the Hilbert tenth problem and, thus, it is undecidable (take t_1 to be a polynomial and t_2 to be 0). If from Peano axioms we exclude the two axioms for multiplication and we keep the two axioms for addition, we get a theory called Presburger Arithmetics, which is known to be a decidable theory. Thus, the unification problem in the Presburger Arithmetics is decidable.

Unification has been studied also in the case of higher order languages, but we will not discuss this issue here. For the expert reader we only recall that *monadic* second-order $\mathcal{E}$-unification is a decidable problem, while second-order $\mathcal{E}$-unification is, in general, an undecidable problem [**7**, page 12].

We introduce also the following notion.

DEFINITION 8.2. [**Matching Modulo Equations or $\mathcal{E}$-matching**] Given a signature Σ, two terms t_1 and t_2 in $\mathcal{T}(\Sigma \cup \mathcal{V})$, and a set $\mathcal{E}$ of Σ-equations, we say that t_1 *$\mathcal{E}$-matches* (or *matches modulo the equations $\mathcal{E}$*) t_2 iff there exists a substitution $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma \cup \mathcal{V})$ such that $\sigma(t_1) =_{\mathcal{E}} t_2$ [**7**, page 12].

A substitution σ such that $\sigma(t_1) =_{\mathcal{E}} t_2$ is called an *$\mathcal{E}$-matcher* of t_1 and t_2.

For instance, an $\mathcal{E}^+$-matcher of $x+0$ and y is a substitution σ such that $\sigma(x)=y$.

9. Rewriting Systems

In this section we introduce the notion of a rewriting system which will be useful in the sequel (see, for instance, Example 6.13 on page 186 and Section 7 on page 192).

Let F be a finite set of symbols with arity $n\,(\geq 0)$. Let *Vars* be a countable set of variables (with arity 0). Let $T(F)$ be the set of terms constructed from symbols in F. The elements of $T(F)$ are said to be *ground terms*. Let $T(F \cup Vars)$ be the set of terms constructed from symbols in F and variables in *Vars*. The elements of $T(F \cup Vars)$ with at least one occurrence of a variable are said to be *non-ground terms*.

Obviously, $T(F) \subset T(F \cup Vars)$.

Given an expression e, the set of variables occurring in e is denoted by $vars(e)$. Likewise, given a set E of expressions, the set of variables occurring in E is denoted by $vars(E)$.

A *rewriting system* (or a *term rewriting system*) Σ *over* $T(F \cup Vars)$ is a finite set $\{\langle \ell_k, r_k \rangle \mid 1 \leq k \leq K\}$ of pairs of terms in $T(F \cup Vars)$, whose variables range over $T(F)$. A pair $\langle \ell_k, r_k \rangle$, also called a *rule* or a *rewriting rule*, is more often written as $\ell_k \longrightarrow r_k$. For any k, with $1 \leq k \leq K$, we assume that $vars(r_k) \subseteq vars(\ell_k)$.

DEFINITION 9.1. [**Substitution**] A *substitution* ϑ is a total function from the finite set $V \subseteq Vars$ of variables to $T(F \cup Vars)$. The set V is said to be the *domain* of the substitution ϑ and it is denoted by $dom(\vartheta)$. The *range* of ϑ, denoted $rng(\vartheta)$, is the set $\{\vartheta(x) \mid x \in dom(\vartheta)\}$.

Given a substitution ϑ, for any $x \in dom(\vartheta)$, a pair $\langle x, \vartheta(x) \rangle$, also denoted $x/\vartheta(x)$, is said to be a *binding* of the substitution ϑ. An *identity binding* is a binding of the form x/x, for some $x \in Vars$. We assume that the substitutions do *not* include identity bindings, that is, for any given substitution ϑ, for all $x \in dom(\vartheta)$, $\vartheta(x) \neq x$.

A substitution ϑ is said to be *ground* iff for all $x \in dom(\vartheta)$, $\vartheta(x)$ is a term in $T(F)$.

Unless otherwise specified, we will assume that every substitution ϑ is *in solved form*, that is, $dom(\vartheta) \cap vars(rng(\vartheta)) = \emptyset$.

Given a term t and a substitution ϑ such that $vars(t)$ is included in the domain of ϑ, the application of the substitution ϑ to the term t returns the term, denoted $t\,\vartheta$ or $\vartheta(t)$, which is obtained from t by replacing every variable $x \in vars(t)$, by the term $\vartheta(x)$. The term $t\,\vartheta$ is said to be an *instance* of t or the ϑ-instance of the term t.

Given the substitutions $\vartheta = \{x_1/t_1, \ldots, x_n/t_n\}$ and $\sigma = \{y_1/s_1, \ldots, y_m/s_m\}$, their *composition*, denoted $\vartheta\,\sigma$, is the substitution obtained from $\{x_1/t_1\sigma, \ldots, x_n/t_n\sigma$, $y_1/s_1, \ldots, y_m/s_m\}$ by deleting: (i) the identity bindings, and (ii) every binding y_i/s_i, for $i = 1, \ldots, m$, such that $y_i \in \{x_1, \ldots, x_n\}$.

For instance, given the substitutions $\vartheta = \{x/v,\ y/w\}$ and $\sigma = \{v/y,\ w/y,\ x/a\}$, we have that $\vartheta\sigma = \{x/y,\ v/y,\ w/y\}$ which is obtained from the set $\{x/y,\ y/y,\ v/y,$ $w/y,\ x/a\}$ of bindings by deleting: (i) the identity binding y/y, and (ii) the binding x/a because $x \in dom(\vartheta)$.

Note that $dom(\vartheta\,\sigma)$ may be a proper superset of $dom(\vartheta)$ and it may be a proper subset of $dom(\vartheta) \cup dom(\sigma)$.

DEFINITION 9.2. [**Unifier and Most General Unifier**] Let us consider any two terms t_1 and t_2. Let W be $vars(t_1) \cup vars(t_2)$. A substitution ϑ with domain $V \subseteq W$ is said to be a *unifier* of t_1 and t_2 iff $t_1\,\vartheta = t_2\,\vartheta$. The substitution ϑ is said to be a *most general unifier* of t_1 and t_2 iff for any other unifier ρ of t_1 and t_2, there exists a substitution σ such that $\rho = \vartheta\,\sigma$.

Let us consider the set of symbols $F = \{a, b\}$. Let the arity of a be 0 and the arity of b be 2. Let x, y, z, and w be variables. Let us also consider the terms $t_1 =_{def} b(b(z, x), b(a, w))$ and $t_2 =_{def} b(b(z, a), y)$.

The substitutions $\rho = \{x/a,\ y/b(a, a),\ z/a,\ w/a\}$ and $\vartheta = \{x/a,\ y/b(a, w)\}$ are unifiers of t_1 and t_2. ϑ is a most general unifier and we have that $\rho = \vartheta\,\{z/a,\ w/a\}$. The substitution ρ is a ground substitution and ϑ is not.

DEFINITION 9.3. [**Matching Substitution**] A unifier ϑ of two terms t_1 and t_2 is said to be a *matching substitution* if either $t_1\,\vartheta = t_2$ or $t_2\,\vartheta = t_1$, that is, the variables of the domain of ϑ occur either in t_1 only or in t_2 only. If $t_1\,\vartheta = t_2$, we say that t_1 *matches* t_2 according to (or via) the substitution ϑ (or the matcher ϑ), and likewise, if $t_2\,\vartheta = t_1$, we say that t_2 *matches* t_1 according to the substitution ϑ.

The matching substitution $\vartheta = \{y/b(a,x)\}$ is such that $b(a,y)\,\vartheta = b(a,b(a,x))$. There is no matching substitution ϑ such that $b(x,a)\,\vartheta = b(a,y)$ or $b(x,a) = b(a,y)\,\vartheta$. There is no ground matching substitution ϑ such that $b(a,y)\,\vartheta = b(a,b(a,x))$.

DEFINITION 9.4. [**Associative-Commutative Unifier and Matcher**] An *associative-commutative unifier* (or *AC-unifier*, for short) ϑ of two terms t_1 and t_2 is a unifier of t_1 and t_2 in which we assume that some symbols in t_1 and t_2 are associative and commutative. An associative-commutative unifier ϑ of two terms t_1 and t_2 is said to be an *associative-commutative matcher* (or *AC-matcher*, for short) of t_1 and t_2 if $t_1\vartheta = t_2$.

For instance, let us consider the set of symbols $F = \{0, s, +\}$ with the arities 0, 1, and 2, respectively. Let us also assume that $+$ is associative and commutative. A most general AC-unifier of the term $t_1 =_{def} (0 + s(0)) + x$ and the term $t_2 =_{def} s(0) + (s(y) + 0)$ is the matching substitution $\{x/s(y)\}$. That substitution is an AC-matcher of t_1 and t_2.

Let us consider a rewriting system $\Sigma = \{\ell_k \longrightarrow r_k \mid 1 \leq k \leq K\}$ over $T(F \cup Vars)$. Given a term t_1 we can derive a term t_2 by applying once the k-th rewriting rule of Σ iff (i) ℓ_k *matches* t_1 (or a subterm s of t_1) according to a substitution ϑ, that is, $\ell_k\vartheta = t_1$ (or $\ell_k\vartheta = s$, respectively), and (ii) t_2 is obtained from t_1 by replacing t_1 (or by replacing s in t_1, respectively) by $r_k\,\vartheta$. In this case we write $t_1 \longrightarrow_k t_2$ and we say that the terms t_1 and t_2 are in the *rewriting relation* $\longrightarrow_k$, or t_1 can be rewritten into t_2 using the k-th rewriting rule.

When it is understood from the context or it is irrelevant, we will feel free to omit the subscript k and we will simply write $\longrightarrow$, instead of $\longrightarrow_k$.

DEFINITION 9.5. [**Rewriting Relation**] We associate with every rewriting system $\Sigma = \{\ell_k \longrightarrow r_k \mid 1 \leq k \leq K\}$ a rewriting relation $\longrightarrow \subseteq T(F \cup Vars) \times T(F \cup Vars)$ which is $\bigcup_{1 \leq k \leq K} \longrightarrow_k$.

As usual, (i) $\longrightarrow^+$ denotes the transitive closure of $\longrightarrow$, (ii) $\longrightarrow^*$ denotes the reflexive, transitive closure of $\longrightarrow$, and (iii) $\longleftrightarrow^*$ denotes the symmetric closure of $\longrightarrow^*$.

For any $j \geq 0$, we define by induction on j, the binary relation $\longrightarrow^j$, called the *j-fold composition of* $\longrightarrow$, as follows:

$$\begin{array}{lll} \longrightarrow^0 & =_{def} & I_{T(F \cup Vars)}, \quad \text{that is, the identity relation on } T(F \cup Vars), \text{ and} \\ \longrightarrow^{j+1} & =_{def} & \longrightarrow^j; \longrightarrow, \quad \text{for all } j \geq 0. \end{array}$$

It is easy to see that for all $j \geq 0$, $\longrightarrow^j; \longrightarrow$ is equal to $\longrightarrow; \longrightarrow^j$.

In order to state a few termination properties which we will be useful in the sequel (see, for instance, Example 6.13 on page 186), now we introduce the following notions and results [**2**].

DEFINITION 9.6. [**Well-founded Set**] An ordered set $(A, >)$ is said to be *well-founded* iff there is no infinite descending sequence of elements of A such that $a_1 > a_2 > \ldots > a_n > \ldots$

DEFINITION 9.7. [**Strong Termination of a Rewriting System**] A rewriting system Σ is said to be *strongly terminating* (or *terminating*, or *noetherian*) with respect to a set of terms $T(F)$ iff for all terms $t \in T(F)$ there is no infinite sequence of terms $t_1 \longrightarrow t_2 \longrightarrow \ldots \longrightarrow t_n \longrightarrow \ldots$ such that: (i) t_1 is t, and (ii) for all $p \geq 0$ there exists k such that $t_p \longrightarrow_k t_{p+1}$ (that is, the term t_p can be rewritten into the term t_{p+1} by using the k-th rewriting rule of Σ).

THEOREM 9.8. [**Undecidability of Strong Termination**] Given a rewriting system it is undecidable whether or not it is strongly terminating. If we assume that the rewriting system is ground (that is, no variables occur in its rules) then termination is decidable [**7**, page 16].

Given a set A, we denote by $\mathcal{M}(A)$ the set of the multisets built out of the elements of A. For instance, $\{1, 3, 2, 1, 3\}$ is a multiset of natural numbers, that is, $\{1, 3, 2, 1, 3\} \in \mathcal{M}(N)$, where N is the set of natural numbers.

A multiset in $\mathcal{M}(A)$ is also denoted as a set of pairs in $A \times N$. The second component of each pair is a natural number which specifies the so called *copy number* of the first component of the pair. For example, the multiset $\{1, 3, 2, 1, 3\}$ is also denoted by the set $\{\langle 1, 0\rangle, \langle 3, 0\rangle, \langle 2, 0\rangle, \langle 1, 1\rangle, \langle 3, 1\rangle\}$, where, in particular, $\langle 1, 0\rangle$ and $\langle 1, 1\rangle$ indicate that in the multiset $\{1, 3, 2, 1, 3\}$ there are two occurrences of the element 1, a first one with copy number 0 and a second one with copy number 1.

Copy numbers are assumed to satisfy the following condition:
for all multisets $M \in \mathcal{M}(A)$, for all $\langle a, n\rangle \in A \times N$, if there exists a pair $\langle a, n+1\rangle$ in M then in M there exists the pair $\langle a, n\rangle$.

In particular, for all multisets $M \in \mathcal{M}(A)$, for all $\langle a, n\rangle \in A \times N$, if $\langle a, n\rangle \in M$ then $\langle a, 0\rangle \in M$.

Let $\uplus$ denote the disjoint union of multisets. For instance, $\{1, 3, 2, 1, 3\} \uplus \{3, 2, 2\} = \{1, 3, 2, 1, 3, 3, 2, 2\}$.

Let $\equiv$ denote the syntactic identity of terms.

DEFINITION 9.9. [**Permutation Equivalence**] Let us consider a set F of symbols. Let $f \in F$ be a symbol of arity m. Given any two terms $s, t \in T(F)$, we say that $s \equiv f(s_1, \ldots, s_m)$ is *permutation equivalent* to $t \equiv f(t_1, \ldots, t_m)$, and we write $s \sim t$, iff there exists a permutation $\langle \pi(1), \pi(2), \ldots, \pi(m)\rangle$ of $\langle 1, 2, \ldots, m\rangle$ such that, for $i = 1, \ldots, m$, $s_i \sim t_{\pi(i)}$. (Obviously, for $i = 1, \ldots, m$, in order to establish $s_i \sim t_{\pi(i)}$ we may use a permutation different from π. Indeed, the arity of the top operators of s_i and $t_{\pi(i)}$ may be different from m.)

DEFINITION 9.10. [**Multiset Order**] Given an ordered set $(A, >)$, the *multiset extension* of the order $>$ is an order, denoted $\gg$, on the set $\mathcal{M}(A)$ defined as follows: $\forall M, N \in \mathcal{M}(A)$, $M_1 \gg M_2$ iff $\exists X, Y, Z \in \mathcal{M}(A)$, $M_1 = Z \uplus X$ and $M_2 = Z \uplus Y$ and $\forall y \in Y$, $\exists x \in X$, $x > y$.

For instance, $\{5, 5, 4, 1\} \gg \{5, 5, 3, 2, 3\}$ by taking $X = \{4, 1\}$, $Y = \{3, 2, 3\}$, and $Z = \{5, 5\}$.

In the following definition we generalize the notion of a multiset order.

DEFINITION 9.11. [**Multiset Order Based on an Equivalence**] Given an ordered set $(A, >)$ and an equivalence relation $\sim$ on A, the *multiset extension* $\gg\!\!\!\!\!\sim$ of the order $>$ *based on the equivalence* $\sim$ is a multiset extension of $>$ according to the above Definition 9.10 on the facing page, where: (i) the multiset M_2 is equal to $f(Z) \uplus Y$, and (ii) f is a bijection from $A \times N$ to $A \times N$ such that (here, we represent multisets as sets of pairs using the copy numbers):

$\forall \langle z, n \rangle \in Z$, $\exists z_1 \in A$, $z \sim z_1$ and $f(\langle z, n \rangle) = \langle z_1, n \rangle$.

For instance, let us consider:
(i) the set T of terms whose syntax is defined as follows, where s is the successor function on natural numbers:

$$t \ ::= \ 0 \ \mid \ s(t) \ \mid \ t + t$$

(for reasons of simplicity we will also write k, instead of $s^k(0)$),
(ii) the equivalence $\sim$ based on the commutativity of $+$, that is, $\forall x, y$, $x+y \sim y+x$, and
(iii) the order $>$ defined by the proper subterm relation, modulo $\sim$. For instance, $(2+0)+1 > 0+2$, because $2+0$ is a subterm of $(2+0)+1$ and $2+0 \sim 0+2$.

We have that: $\{0+2, 0+1\} \gg\!\!\!\!\!\sim \{2+0, 0, 1, 0\}$ because, with reference to Definition 9.11, we can take: $Z = \{0+2\}$, $f(\langle 0+2, 0 \rangle) = \langle 2+0, 0 \rangle$ (here we have indicated the elements $0+2$ and $2+0$ with their copy number 0), $X = \{0+1\}$, and $Y = \{0, 1, 0\}$.

THEOREM 9.12. Given an ordered set $(A, >)$, (i) if $>$ is irreflexive and transitive then $\gg$ is irreflexive and transitive, and (ii) $(A, >)$ is well-founded iff $(\mathcal{M}(A), \gg)$ is well-founded.

DEFINITION 9.13. [**Recursive Path Order**] Let us consider an irreflexive, transitive order $\succcurlyeq$ on a set F of symbols and the permutation equivalence $\sim$ over $T(F)$. The *recursive path order* $>$ (*rpo*, for short) over the set $T(F)$ *associated with* $\succcurlyeq$ *and* $\sim$, is recursively defined as follows: for any given term $s \equiv f(s_1, \ldots, s_m)$ and $t \equiv g(t_1, \ldots, t_n)$, we have that: $s > t$ iff $s \not\sim t$ and

either $f \succcurlyeq g$ and for $i = 1, \ldots, n$, $s > t_i$ (rpo 1)

or $f \equiv g$ and there exists a permutation π of $\langle 1, \ldots, m \rangle$ such that (rpo 2)
$\{s_1, \ldots, s_m\} \gg\!\!\!\!\!\sim \{t_{\pi(1)}, \ldots, t_{\pi(m)}\}$ (in this case $m = n > 0$)

or for some $i = 1, \ldots, m$, $(s_i > t$ or $s_i \sim t)$, (rpo 3)

where $\gg\!\!\!\!\!\sim$ is the multiset extension of the order $>$ based on the permutation equivalence $\sim$.

THEOREM 9.14. An ordered set $(F, \succcurlyeq)$ is well-founded iff $(T(F), >)$, where $>$ is the rpo over $T(F)$ associated with $\succcurlyeq$ and the permutation equivalence $\sim$, is well-founded.

DEFINITION 9.15. [**Simplification Order**] An irreflexive, transitive order $>$ over a set $T(F)$ of terms is said to be a *simplification order* iff for any term $t, t' \in T(F)$, for any $f \in F$,

(i) $t > t'$ implies $f(\ldots, t, \ldots) > f(\ldots, t', \ldots)$, and (monotonicity)

(ii) $f(\ldots, t, \ldots) > t$. (subterm)

The proof of the following theorem can be found in [**2**].

THEOREM 9.16. Given a set F of symbols with arity, the recursive path order $>$ over a set $T(F)$ of terms associated with an irreflexive, transitive order $\succ$ on F and the permutation equivalence $\sim$, is a simplification order.

THEOREM 9.17. [**Strong Termination (Dershowitz)**] Let F be a finite set of symbols with arity and *Vars* be a countable set of variables. Let us consider a rewriting system $\Sigma = \{\ell_k \longrightarrow r_k \mid 1 \leq k \leq K\}$ over $T(F \cup Vars)$. If there exists a simplification order $>$ such that for any ground substitution ϑ whose domain is the set of variables occurring in $\bigcup_{1 \leq k \leq K} \ell_k$, we have that $\vartheta(\ell_k) > \vartheta(r_k)$ for $k = 1, \ldots, K$, then Σ is strongly terminating with respect to $T(F)$.

Now we present a different sufficient condition for establishing strong termination of a rewriting system. First we need the following definitions.

DEFINITION 9.18. [**Lexicographic Order**] Given an ordered set $(A, >)$, a lexicographic order $>_{lex}$ on $A \times A$ associated with $>$, is defined as follows: for any $a_1, a_2, b_1, b_2 \in A$, $\langle a_1, a_2\rangle >_{lex} \langle b_1, b_2\rangle$ iff $a_1 > b_1$ or ($a_1 = b_1$ and $a_2 > b_2$).

The lexicographic order can be extended to tuples as follows: $\langle a_1, a_2, \ldots, a_k\rangle >_{lex} \langle b_1, b_2, \ldots, b_k\rangle$ iff there exists i, with $0 \leq i < k$, such that for $j = 1, \ldots, i$, $a_j = b_j$ and $a_{i+1} > b_{i+1}$.

For instance, $\langle 2, 0, 1, 3\rangle >_{lex} \langle 2, 0, 0, 1\rangle$ (in this case $k = 4$ and $i = 2$).

THEOREM 9.19. Given an ordered set $(A, >)$, if $>$ is well-founded then the lexicographic order $>_{lex}$ on $A \times A$ associated with $>$, is well-founded.

DEFINITION 9.20. [**Bounded Lexicographic Recursive Path Order**] Let us consider an irreflexive, transitive order $\succ$ on a set F of symbols. The *bounded lexicographic recursive path order* $>$ (*bl-rpo*, for short) over the set $T(F)$ associated with $\succ$, is recursively defined as follows: for any given term $s \equiv f(s_1, \ldots, s_m)$ and $t \equiv g(t_1, \ldots, t_n)$, we have that: $s > t$ iff $s \not\equiv t$ and

either $f \succ g$ and for $i = 1, \ldots, n$, $s > t_i$ (bl-rpo 1)

or $f \equiv g$ and $\langle s_1, \ldots, s_m\rangle >_{lex} \langle t_1, \ldots, t_n\rangle$ with $m = n > 0$ and (bl-rpo 2.1)
for $i = 1, \ldots, m$, $s > t_i$ (bl-rpo 2.2)

or for some $i = 1, \ldots, m$, $s_i > t$ or $s_i \equiv t$, (bl-rpo 3)

where $>_{lex}$ denotes the lexicographic order associated with $>$.

An alternative definition of a bounded lexicographic recursive path order can be obtained by replacing Condition (bl-rpo 2.2) by a semantic condition of the form:

for $i = 1, \ldots, m$, $[\![s]\!] \rhd [\![t_i]\!]$, (bl-rpo 2.2*)

where $[\![\_]\!]$ is a semantic function from $T(F,Vars)$ to a well-founded ordered set $(D, \rhd)$.

THEOREM 9.21. [**Strong Termination (Kamin-Lévy)**] Let F be a finite set of symbols with arity and *Vars* be a countable set of variables. Let us consider a

rewriting system $\Sigma = \{\ell_k \longrightarrow r_k \mid 1 \leq k \leq K\}$ over $T(F \cup Vars)$. If there exists a bounded lexicographic recursive path order $>$ such that for any ground substitution ϑ whose domain is the set of variables occurring in $\bigcup_{1 \leq k \leq K} \ell_k$, we have that $\vartheta(\ell_k) > \vartheta(r_k)$ for $k = 1, \ldots, K$, then Σ is strongly terminating with respect to $T(F)$.

PROOF. It is based on the fact that the bounded lexicographic recursive path order $>$ over a set $T(F)$ of terms associated with an irreflexive, transitive order $\succ$ on F, is a simplification order. Thus, by Theorem 9.17 on the facing page, we get the thesis. □

Now we will introduce some important concepts and results about term rewriting systems. Some of these concepts will be used in Section 10 where we will present an algorithm for checking whether or not a formula of the Propositional Calculus is a valid formula (that is, a tautology) [**6**]. In particular, we will extend to the rewriting relation associated with a rewriting system, the concepts presented at the end of Section 1 (see page 24).

Let F be a finite set of symbols with arity and *Vars* be a countable set of variables. Let $\Sigma = \{\ell_k \longrightarrow r_k \mid 1 \leq k \leq K\}$ be a rewriting system over $T(F \cup Vars)$ and $\longrightarrow$ the rewriting relation associated with Σ.

DEFINITION 9.22. [**Confluence**] We say that the rewriting relation $\longrightarrow$ is *confluent* iff for every p, ℓ, $r \in T(F \cup Vars)$, if $p \longrightarrow^* \ell$ and $p \longrightarrow^* r$ then there exists $q \in T(F \cup Vars)$ such that $\ell \longrightarrow^* q$ and $r \longrightarrow^* q$.

An equivalent definition is the following one.

DEFINITION 9.23. [**Church-Rosser Property**] We say that the rewriting relation $\longrightarrow$ is *Church-Rosser* iff for every ℓ, $r \in T(F \cup Vars)$, if $\ell \longleftrightarrow^* r$ then there exists $q \in T(F \cup Vars)$ such that $\ell \longrightarrow^* q$ and $r \longrightarrow^* q$.

DEFINITION 9.24. [**Strong Termination**] We say that the rewriting relation $\longrightarrow$ is *strongly terminating* (or *terminating*, or *noetherian*) iff there is no infinite sequence $p_0, p_1, p_2, \ldots$ of terms in $T(F \cup Vars)$ such that for all $i \geq 0$, $p_i \longrightarrow p_{i+1}$.

As an instance of the well-founded induction rule (see Section 1.6 on page 72), we have the following *noetherian induction rule*, which, given a set S, allows us to prove a property of all its elements.

Let us consider a set S and a binary, terminating relation $\longrightarrow \subseteq S \times S$. Thus, by definition, there is no infinite sequence $x_0, x_1, x_2, \ldots$ of elements in S such that for all $i \geq 0$, $x_i \longrightarrow x_{i+1}$. Let $P(x)$ be a predicate on S. Here is the noetherian induction rule where, as usual, $\longrightarrow^+$ denotes the transitive closure of $\longrightarrow$.

(Noetherian Induction)

$$\frac{\forall x \in S.\, (\forall y.\, (x \longrightarrow^+ y \Rightarrow P(y))) \Rightarrow P(x)}{\forall x \in S.\, P(x)}$$

We will use this induction rule in the proof of Theorem 9.28 on the following page.

DEFINITION 9.25. [**Normal Form and Reduction**] We say that a term v is *in normal form* (or *irreducible*) w.r.t. a rewriting system Σ iff there is no element v_1 such that $v \longrightarrow v_1$. We say that v is a *normal form* of u w.r.t. a rewriting system Σ iff $u \longrightarrow^* v$ and there is no element v_1 such that $v \longrightarrow v_1$. By $u\!\downarrow$ we denote a normal form of u. (Note that, in general, for any given term u there is more than one normal form of u.)

For any given term u, when its normal form is unique, the computation of the normal form $u\!\downarrow$ is called the *reduction* of u, and to *reduce* a term u means to compute its normal form.

THEOREM 9.26. [**Uniqueness of Normal Form**] If a rewriting relation is confluent and strongly terminating then for all $u \in T(F \cup Vars)$, then there exists a unique $v \in T(F \cup Vars)$ such that v is a normal form of u.

DEFINITION 9.27. [**Local Confluence**] We say that $\longrightarrow$ is *locally confluent* iff for all $p, \ell, r \in T(F \cup Vars)$, if $p \longrightarrow \ell$ and $p \longrightarrow r$ then there exists $q \in T(F \cup Vars)$ such that $\ell \longrightarrow^* q$ and $r \longrightarrow^* q$.

THEOREM 9.28. [**Newman Theorem**] Let us consider a rewriting system whose rewriting relation $\longrightarrow$ is strongly terminating. Then $\longrightarrow$ is confluent iff $\longrightarrow$ is locally confluent.

PROOF. ($\Rightarrow$) Obvious. ($\Leftarrow$) We assume local confluence, that is, we assume that $\forall x, y, z \in T(F \cup Vars)$, if $x \longrightarrow y$ and $x \longrightarrow z$ then there exists $t \in T(F \cup Vars)$ such that $y \longrightarrow^* t$ and $z \longrightarrow^* t$. In order to show confluence, we have to show that $\forall x \in T(F \cup Vars).\, P(x)$, where the predicate $P(x)$ is defined as follows:

$$P(x) =_{def} \forall y, z \in T(F \cup Vars).\, \exists t \in T(F \cup Vars).\\ \big((x \longrightarrow^* y \wedge x \longrightarrow^* z) \Rightarrow (y \longrightarrow^* t \wedge z \longrightarrow^* t)\big)$$

In order to prove $\forall x \in T(F \cup Vars).\, P(x)$, since $\longrightarrow$ is terminating, by noetherian induction it is enough to show $\forall x \in T(F \cup Vars).\, \big(\forall u.\, (x \longrightarrow^+ u \Rightarrow P(u)) \Rightarrow P(x)\big)$. Thus, (i) we take any $x \in T(F \cup Vars)$, (ii) we assume:

(H) $\quad \forall u.\, (x \longrightarrow^+ u \Rightarrow P(u))$

and (iii) we have to show $P(x)$.

In order to show $P(x)$, (i) we take any $y, z \in T(F \cup Vars)$, (ii) we assume that $x \longrightarrow^* y$ and $x \longrightarrow^* z$, that is, we assume that there exist $m \geq 0$ and $n \geq 0$ such that $x \longrightarrow^m y$ and $x \longrightarrow^n z$, and (iii) we have to show that there exists $t \in T(F \cup Vars)$ such that $y \longrightarrow^* t$ and $z \longrightarrow^* t$.

Now there are three cases.
Case (i): $m = 0$. In this case $y \equiv x$ and we take $t \equiv z$.
Case (ii): $n = 0$. In this case $z \equiv x$ and we take $t \equiv y$.
Case (iii): $m \neq 0$ and $n \neq 0$. In this case there exist y_1 and z_1 such that $x \longrightarrow y_1 \longrightarrow^* y$ and $x \longrightarrow z_1 \longrightarrow^* z$. By local confluence there exists u such that $y_1 \longrightarrow^* u$ and $z_1 \longrightarrow^* u$ (see Figure 2 on the next page).

By Hypothesis (H), we have that: (i) there exists v such that $y \longrightarrow^* v$ and $u \longrightarrow^* v$, and (ii) there exists w such that $u \longrightarrow^* w$ and $z \longrightarrow^* w$. Since $u \longrightarrow^* v$ and $u \longrightarrow^* w$, by Hypothesis (H), there exists t such that $v \longrightarrow^* t$ and $w \longrightarrow^* t$. Thus, from $y \longrightarrow^* v$ and $v \longrightarrow^* t$, by transitivity, we get:

(A) $\quad y \longrightarrow^* t$.

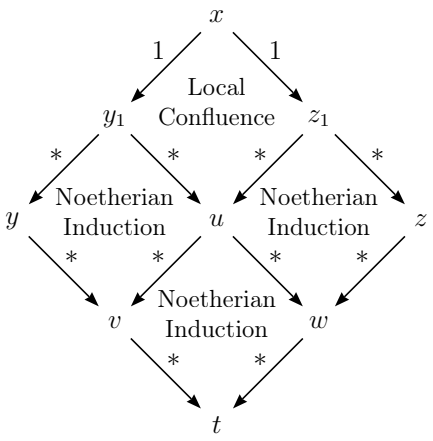


FIGURE 2. Graphical representation of the proof of Newman Theorem.

From $z \longrightarrow^* w$ and $w \longrightarrow^* t$, by transitivity, we get:

(B) $z \longrightarrow^* t$.

Having proved Properties (A) and (B), the proof of the theorem is completed. □

In Newman Theorem above the hypothesis that rewriting relation $\longrightarrow$ be strongly terminating is necessary. Indeed, the following example shows that if a rewriting system is *not* strongly terminating then local confluence does not imply confluence. Consider the four rewriting rules: (i) $a \longrightarrow b$, (ii) $a \longrightarrow c$, (iii) $b \longrightarrow a$, and (iv) $b \longrightarrow d$. We have that a has two normal forms c and d, because $a \longrightarrow^+ c$, $a \longrightarrow^+ d$, and c and d cannot be rewritten.

DEFINITION 9.29. **[Critical Pairs]** Let us consider a rewriting system $\Sigma = \{\ell_k \longrightarrow r_k \mid 1 \leq k \leq K\}$. Let us consider two (not necessarily distinct) rules $\ell_1 \longrightarrow r_1$ and $\ell_2 \longrightarrow r_2$ of Σ and let us assume, without loss of generality, that they do not have variables in common. This can be achieved by suitable renaming of variables. Let us also assume that a non-variable subterm u of ℓ_1 is unifiable with ℓ_2 with a most general unifier ϑ. Let $\ell_1[u \leftarrow r_2]$ denote the term ℓ_1 where the subterm u is replaced by r_2. Then we say that the pair $\langle r_1\,\vartheta,\ \ell_1[u \leftarrow r_2]\,\vartheta\rangle$ is a *critical pair* of Σ.

For instance, let us consider the following two rewriting rules, whose function symbols are f, g, h, k, a, m and whose variables are x, y, z:

1. $f(x, g(x, h(y))) \longrightarrow k(x, y)$
2. $g(a, z) \longrightarrow m(z)$

Since $g(x, h(y))$ and $g(a, z)$ are unifiable via the most general unifier $\{x/a,\ z/h(y)\}$, we get the following two rewritings:

$$f(a, g(a, h(y))) \begin{cases} \longrightarrow 1.\ k(a, y) \\ \longrightarrow 2.\ f(a, m(h(y))) \end{cases}$$

and the critical pair*:* $\langle k(a, y),\ f(a, m(h(y)))\rangle$.

A critical is unique up to a permutation. Indeed, for the results of our interest here, the order of the components of a critical pair is not significant and, thus, a critical pair $\langle t_1, t_2\rangle$ will also be denoted by $\langle t_2, t_1\rangle$.

THEOREM 9.30. [**Knuth-Bendix Theorem**] Given a rewriting system Σ, the associated rewriting relation $\longrightarrow$ is locally confluent iff for every critical pair $\langle p, q\rangle$ of Σ, we have that $p\downarrow \equiv q\downarrow$, that is, there exists a normal form of p which is syntactically identical to a normal form of q.

Thus, a rewriting relation $\longrightarrow$ is locally confluent if there are no critical pairs in Σ. Note that in this Theorem 9.30 strong termination of Σ is not required.

DEFINITION 9.31. [**Linear Term**] A term is said to be *linear* if every variable occurs in it at most once.

DEFINITION 9.32. [**Left Linear Rewriting System**] A rewriting system Σ is said to be *left linear* if for every rule $\ell \longrightarrow r$ in Σ, we have that ℓ is a linear term.

THEOREM 9.33. [**Huet Theorem**] Consider a left linear rewriting system Σ. If for every critical pair $\langle p, q\rangle$, we have that either $p \mathrel{\#\!\!\!\rightarrow} q$ or $q \mathrel{\#\!\!\!\rightarrow} p$, where $\mathrel{\#\!\!\!\rightarrow}$ denotes the parallel rewriting of disjoint subterms using rules in Σ, then Σ is confluent.

This theorem tells us that any left linear rewriting system without critical pairs is confluent (see, for instance, the Ackermann rewriting system of Definition 6.11 on page 184).

DEFINITION 9.34. [**Canonical or Complete Rewriting System**] A rewriting system whose rewriting relation $\longrightarrow$ is confluent and strongly terminating, is said to be *canonical* (or *complete*).

Theorem 9.28 on page 46 and Theorem 9.30 give us a method for deciding within the set $T(F\cup Vars)$ of terms, equality modulo a given set $\mathcal{E}$ of equations, denoted $=_{\mathcal{E}}$, as we now specify.

Consider a set $\mathcal{E}$ of equations over $T(F\cup Vars)$.

REMARK 9.35. Note that in every set of equations, we assume that, for any two terms t_1 and t_2 in $T(F\cup Vars)$,
(i) no equation of the form $t_1 = t_1$ exists, and
(ii) $t_1 = t_2$ and $t_2 = t_1$ are two ways of denoting the same equation.
Recall also that $\equiv$ denotes syntactic identity of terms. □

Consider any two terms t_1 and t_2 in $T(F\cup Vars)$. In order to decide whether or not $t_1 =_{\mathcal{E}} t_2$, we may construct a canonical rewriting system $\mathcal{R}$ using the Knuth-Bendix Completion algorithm which we now present (see Figure 3 on the next page).

In this algorithm and in what follows, for any term $t \in T(F\cup Vars)$, by $\mathcal{R}(t)$ we denote the *unique* normal form of t obtained by applying any number of times the rewriting rules of the current value of $\mathcal{R}$.

Knuth-Bendix Completion.

Input: a (possibly empty) finite set $\mathcal{E}$ of equations.

$\mathcal{R} := \emptyset$;
while $\mathcal{E} \neq \emptyset$ **do** select an equation $t_1 = t_2$ in $\mathcal{E}$;
  $n_1 := \mathcal{R}(t_1)$; $n_2 := \mathcal{R}(t_2)$;
  if $n_1 \equiv n_2$ **then** $\mathcal{E} := \mathcal{E} - \{t_1 = t_2\}$ **else**
    if α: $vars(n_2) \subseteq vars(n_1)$ **then** $\langle \ell := n_1,\ r := n_2 \rangle$ **or**
    if β: $vars(n_1) \subseteq vars(n_2)$ **then** $\langle \ell := n_2,\ r := n_1 \rangle$;
    if $\neg(\alpha \vee \beta)$ **then** stop with failure **else**
      let $\mathcal{R}_{\mathcal{E}} := \{\ell' \to r' \in \mathcal{R} \mid \ell'$ or r' contains an instance of ℓ as a subterm$\}$ **and**
        $\mathcal{R}_{\mathcal{R}} := (\mathcal{R} - \mathcal{R}_{\mathcal{E}}) \cup \{\ell \to r\}$ **in**
      if $\mathcal{R}_{\mathcal{R}}$ is not strongly terminating **then** stop with failure **else**
        $\mathcal{R} := \mathcal{R}_{\mathcal{R}}$;
        $\mathcal{E} := (\mathcal{E} - \{t_1 = t_2\}) \cup \{\ell' = r' \mid \ell' \to r' \in \mathcal{R}_{\mathcal{E}}\}$
          $\cup \{p = q \mid \langle p, q \rangle$ is a critical pair of $\mathcal{R}_{\mathcal{R}}\}$
od

Output: a canonical rewriting system $\mathcal{R}$ such that
  $\forall t_1, t_2 \in T(F \cup Vars),\ t_1 =_{\mathcal{E}} t_2$ iff $\mathcal{R}(t_1) \equiv \mathcal{R}(t_2)$.

FIGURE 3. Knuth-Bendix Completion algorithm.

The following points may help the reader to understand the Knuth-Bendix Completion algorithm.

(i) A new equation is generated by a critical pair: this situation occurs when an instance of a term of an equation that can be rewritten into two different ways (in a non-variable position) by the rewriting rules obtained so far (possibly the same rule).
(ii) Before processing an equation, we simplify it by applying the rewriting rules obtained so far, and if we get the trivial equation $t = t$ we discard it, and
(iii) We orient every equation so to guarantee strong termination.

In the Knuth-Bendix Completion algorithm we have that:

(i) the operator **or** denotes a nondeterministic choice and, thus, if we have that $vars(n_1) = vars(n_2)$, we can equivalently put either (i.1) $\langle \ell := n_1,\ r := n_2 \rangle$ or (i.2) $\langle \ell := n_2,\ r := n_1 \rangle$,

(ii) the choice between (i.1) and (i.2), that is, the choice of the orientation of the equation, is important because strong termination of the rewriting system $\mathcal{R}_{\mathcal{R}}$ may depend on that choice,

(iii) the ordering which allows us to show strong termination of the rewriting system $\mathcal{R}_{\mathcal{R}}$, is determined, in general, according to the rewriting system $\mathcal{R}_{\mathcal{R}}$ itself,

(iv) the instance of ℓ can be generated by the identity substitution (that is, can be ℓ itself) and the subterm relation need not be a proper subterm relation (that is, the instance of ℓ can be identical to ℓ' or r'),

(v) in the definition of $\mathcal{R}_{\mathcal{E}}$ it is not correct to replace: « ℓ' or r' contains an instance of ℓ as a subterm » by: « ℓ' contains an instance of ℓ as a subterm », because if $vars(n_1) = vars(n_2)$ we may introduce in the rewriting system $\mathcal{R}$ either the rewriting rule $n_1 \to n_2$ or the rewriting rule $n_2 \to n_1$, and

(vi) we can get rid of the set $\mathcal{R}_{\mathcal{R}}$ and we can replace $\mathcal{R}_{\mathcal{R}}$ by $\mathcal{R}$ (and delete the assignment $\mathcal{R} := \mathcal{R}_{\mathcal{R}}$). In our presentation of the algorithm we kept the set $\mathcal{R}_{\mathcal{R}}$ because it helps clarifying the examples below. (Note that the set $\mathcal{R}_{\mathcal{R}}$ also occurs in the presentation of [**7**], where it is named $\mathcal{R}''$.)

For the computations of the critical pairs of $\mathcal{R}_{\mathcal{R}}$, we note that:

(i) they can be computed incrementally, that is, we can take into account at each execution of the body of the while-do loop the new rewriting rule $\ell \longrightarrow r$ only, and

(ii) there is no need to consider any critical pair $\langle p, q \rangle$ such that for the current rewriting system $\mathcal{R}$, we have that $\mathcal{R}(p) \equiv \mathcal{R}(q)$, because the associated equation $p = q$ will be deleted during a subsequent execution of the body of the while-do loop.

If the algorithm stops with failure it means that, starting from $\mathcal{E}$, it does not construct a canonical rewriting system $\mathcal{R}$ such that $\forall t_1, t_2 \in T(F \cup Vars)$, $t_1 =_{\mathcal{E}} t_2$ iff $\mathcal{R}(t_1) \equiv \mathcal{R}(t_2)$.

Note that the algorithm may fail also because it keeps on generating new rewriting rules and, therefore, a canonical rewriting system $\mathcal{R}$ is never constructed.

Having constructed from $\mathcal{E}$ the canonical term rewriting system $\mathcal{R}$ by the Knuth-Bendix Completion algorithm, we have that, for any terms $t_1, t_2 \in T(F \cup Vars)$,

(i) if $t_1 = t_2 \in \mathcal{E}$ then $\mathcal{R}(t_1) \equiv \mathcal{R}(t_2)$

(ii) if $\ell \to r \in \mathcal{R}$ then $\ell =_{\mathcal{E}} r$, and

(iii) $t_1 =_{\mathcal{E}} t_2$ iff $\mathcal{R}(t_1) \equiv \mathcal{R}(t_2)$ [**7**, page 21].

Thus, we have a decision procedure for the validity problem in $\mathcal{M}(\mathcal{E})$ and also a decision procedure for the word problem in $\mathcal{I}(\mathcal{E})$.

Now we present the derivation of a canonical term rewriting system for the group axioms using the Knuth-Bendix Completion algorithm.

We start off from the following usual (non-commutative) group equality axioms with the associative operation $+$, the unit 0, and the inverse operation i of arity 1 (see also [**7**, page 33]):

$E1.$ $0 + x = x$

$E2.$ $i\,x + x = 0$

$E3.$ $(x + y) + z = x + (y + z)$

Initially, we have that: $\mathcal{R} = \emptyset$ and $\mathcal{E} = \{E1, E2, E3\}$.

Every step of the derivation corresponds to one execution of the body of the while-do of the Knuth-Bendix Completion. At every step, after the selection of an equation Ej of the form either $t_1 = t_2$ or $t_2 = t_1$, we denote by Rk the rewriting rule of the form $\mathcal{R}(t_1) \longrightarrow \mathcal{R}(t_2)$, where $\mathcal{R}$ is the current rewriting system. Symmetrically, given a rewriting rule Rk of the form $t_1 \longrightarrow t_2$, by Ek we denote the associated

equation $t_1 = t_2$. Thus, after the selection of an equation Ek, in order to get the rewriting rule Rk, we have to choose an orientation and we have to compute the normal forms of the two terms of the equation.

The orientations of the equations have been chosen so to get strongly terminating rewriting systems. We leave it to the reader to check that termination property holds because the following measure $\mu(t)$ of a term t decreases its lexicographic value when replacing a left hand side of an equation by the corresponding right hand side (since rules $R16$ and $R3$ do not decrease the size of the term t, the first component of $\mu(t)$ is required for rule $R16$ and the second one is required for rule $R3$):

$$\begin{aligned}\mu(t) =_{def}\ & \langle \textit{number of } i\textit{'s above } +\textit{'s},\\ & \textit{multisets of sizes of left operands of } +,\\ & \textit{size}(t)\rangle.\end{aligned}$$

On Table 1 on the following page we have listed the various rewriting rules which are computed by the algorithm.

Step 1. Select $E1$. $R1$: $0 + x \longrightarrow x$
$\mathcal{R}_{\mathcal{E}} = \emptyset$; $\mathcal{R}_{\mathcal{R}} = \{R1\}$;
$\mathcal{E} = \{E2, E3\}$.

Step 2. Select $E2$. $R2$: $i\,x + x \longrightarrow 0$
$\mathcal{R}_{\mathcal{E}} = \emptyset$; $\mathcal{R}_{\mathcal{R}} = \{R1, R2\}$;
$\mathcal{E} = \{E3\}$.

Step 3. Select $E3$. $R3$: $(x + y) + z \longrightarrow x + (y + z)$
$\mathcal{R}_{\mathcal{E}} = \emptyset$; $\mathcal{R}_{\mathcal{R}} = \{R1, R2, R3\}$;
$\mathcal{E} = \{\} \cup \{E4\}$, where:

$E4$: $0 + y = i\,x + (x + y)$, because: $(i\,x + x) + y \begin{array}{l}\longrightarrow_{R2}\ 0 + y\\ \longrightarrow_{R3}\ i\,x + (x + y)\end{array}$

Step 4. Select $E4$. $R4$: $i\,x + (x + y) \longrightarrow y$
$\mathcal{R}_{\mathcal{E}} = \emptyset$; $\mathcal{R}_{\mathcal{R}} = \{R1, R2, R3, R4\}$;
$\mathcal{E} = \{\} \cup \{E5, E6, E7\}$, where:

$E5$: $i\,0 + x = x$, because: $i\,0 + (0 + x) \begin{array}{l}\longrightarrow_{R1}\ i\,0 + x\\ \longrightarrow_{R4}\ x\end{array}$

$E6$: $i\,i\,x + 0 = x$, because: $i\,i\,x + (i\,x + x) \begin{array}{l}\longrightarrow_{R2}\ i\,i\,x + 0\\ \longrightarrow_{R4}\ x\end{array}$

$E7$: $i\,i\,x + y = x + y$, because: $i\,i\,x + (i\,x + (x + y)) \begin{array}{l}\longrightarrow_{R4}\ i\,i\,x + y\\ \longrightarrow_{R4}\ x + y\end{array}$

Step 5. Select $E5$. $R5$: $i\,0 + x \longrightarrow x$
$\mathcal{R}_{\mathcal{E}} = \emptyset$; $\mathcal{R}_{\mathcal{R}} = \{R1, R2, R3, R4, R5\}$;
$\mathcal{E} = \{E6, E7\} \cup \{E8\}$, where:

$E8$: $i\,i\,0 + x = x$, because: $i\,i\,0 + (i\,0 + x) \begin{array}{l}\longrightarrow_{R4}\ i\,i\,0 + x\\ \longrightarrow_{R5}\ x\end{array}$

Step 6. Select $E6$. $R6$: $i\,i\,x + x0 \longrightarrow x$
$\mathcal{R}' = \emptyset$; $\mathcal{R}_{\mathcal{R}} = \{R1, R2, R3, R4, R5, R6\}$;
$\mathcal{E} = \{E7, E8\} \cup \{E9\}$, where:

$E9$: $0 = i\,i\,i\,x + x$, because: $i\,i\,i\,x + (i\,i\,x + 0) \begin{array}{l}\longrightarrow_{R4}\ 0\\ \longrightarrow_{R6}\ i\,i\,i\,x + x\end{array}$

$\boxed{R1.}$	$0 + x \longrightarrow x$	
$\boxed{R2.}$	$i\,x + x \longrightarrow 0$	
$\boxed{R3.}$	$(x + y) + z \longrightarrow x + (y + z)$	
$\boxed{R4.}$	$i\,x + (x + y) \longrightarrow y$	
$R5.$	$i\,0 + x \longrightarrow x$	deleted at Step 13
$R6.$	$i\,i\,x + 0 \longrightarrow x$	replaced at Step 10 by: $\boxed{R6'.}$ $x + 0 \longrightarrow x$
$R7.$	$i\,i\,x + y \longrightarrow x + y$	deleted at Step 15
$R8.$	$i\,i\,0 + x \longrightarrow x$	deleted at Step 14
$R9.$	$i\,i\,i\,x + x \longrightarrow 0$	deleted at Step 9
$\boxed{R10.}$	$i\,0 \longrightarrow 0$	
$\boxed{R11.}$	$i\,i\,x \longrightarrow x$	
$\boxed{R12.}$	$x + i\,x \longrightarrow 0$	
$\boxed{R13.}$	$x + (i\,x + y) \longrightarrow y$	
$R14.$	$x + (y + i(x + y)) \longrightarrow 0$	deleted at Step 20
$R15.$	$x + i(y + x) \longrightarrow i\,y$	deleted at Step 22
$\boxed{R16.}$	$i(y + x) \longrightarrow i\,x + i\,y$	

TABLE 1. Rewriting rules computed by the Knuth-Bendix Completion algorithm starting from the group equality axioms: $E1$: $0 + x = x$, $E2$: $i\,x + x = 0$, and $E3$: $(x + y) + z = x + (y + z)$.

Step 7. Select $E7$. $R7$: $i\,i\,x + y \longrightarrow x + y$
$\mathcal{R}_{\mathcal{E}} = \{R6\}$; $\mathcal{R}_{\mathcal{R}} = \{R1, R2, R3, R4, R5, R7\}$;
$\mathcal{E} = \{E8, E9\} \cup \{E6\}$.

Step 8. Select $E8$. $R8$: $i\,i\,0 + x \longrightarrow x$
$\mathcal{R}' = \emptyset$; $\mathcal{R}'' = \{R1, R2, R3, R4, R5, R7, R8\}$;
$\mathcal{E} = \{E9, E6\} \cup \{E10\}$, where:
$E10$: $0 = i\,0$, because: $i\,i\,0 + i\,0 \longrightarrow_{R2} 0$, $i\,0 \longrightarrow_{R7} i\,0$

Step 9. Select $E9$. $R9$: $i\,i\,i\,x + x \longrightarrow 0$
Since $i\,i\,i\,x + x \longrightarrow_{R7} i\,x + x \longrightarrow_{R2} 0$, rule $R9$ is deleted.
$\mathcal{R}_{\mathcal{R}} = \{R1, R2, R3, R4, R5, R7, R8\}$;
$\mathcal{E} = \{E6, E10\}$.

Step 10. Select $E6$. $R6'$: $x + 0 \longrightarrow x$ (because $i\,i\,x + 0 \longrightarrow_{R7} x + 0$)
$\mathcal{R}_{\mathcal{E}} = \emptyset$; $\mathcal{R}_{\mathcal{R}} = \{R1, R2, R3, R4, R5, R7, R8, R6'\}$;
$\mathcal{E} = \{E10\} \cup \{E11\}$, where:
$E11$: $i\,i\,x = x + 0$, because: $i\,i\,x + 0 \longrightarrow_{R6'} i\,i\,x$, $i\,i\,x \longrightarrow_{R7} x + 0$

Step 11. Select $E10$. $R10$: $i\,0 \longrightarrow 0$
$\mathcal{R}_{\mathcal{E}} = \{R5, R8\}$; $\mathcal{R}_{\mathcal{R}} = \{R1, R2, R3, R4, R7, R6', R8\}$;
$\mathcal{E} = \{E11, E5, E8\}$.

Step 12. Select $E11$. $R11$: $i\,i\,x \longrightarrow x$ (because $x + 0 \longrightarrow_{R6'} x$)
$\mathcal{R}_{\mathcal{E}} = \{R7\}$; $\mathcal{R}_{\mathcal{R}} = \{R1, R2, R3, R4, R6', R10, R11\}$;
$\mathcal{E} = \{E5, E8, E7\} \cup \{E12, E13\}$, where:
$E12$: $0 = x + i\,x$, because: $i\,i\,x + i\,x \begin{array}{l}\longrightarrow_{R2} \\ \longrightarrow_{R11}\end{array} \begin{array}{l}0 \\ x + i\,x\end{array}$

$E13$: $y = x + (i\,x + y)$, because: $i\,i\,x + (i\,x + y) \begin{array}{l}\longrightarrow_{R4} \\ \longrightarrow_{R11}\end{array} \begin{array}{l}y \\ x + (i\,x + y)\end{array}$

Step 13. Select $E5$. $R5$: $i\,0 + x \longrightarrow x$
Since $i\,0 + x \longrightarrow_{R10} 0 + x \longrightarrow_{R1} x$, rule $R5$ is deleted.
$\mathcal{R}_{\mathcal{R}} = \{R1, R2, R3, R4, R6', R10, R11\}$;
$\mathcal{E} = \{E8, E7, E12, E13\}$.

Step 14. Select $E8$. $R8$: $i\,i\,0 + x \longrightarrow x$
Since $i\,i\,0 + x \longrightarrow_{R10} i\,0 + x \longrightarrow_{R10} 0 + x \longrightarrow_{R1} x$, rule $R8$ is deleted.
$\mathcal{R}_{\mathcal{R}} = \{R1, R2, R3, R4, R6', R10, R11\}$;
$\mathcal{E} = \{E7, E12, E13\}$.

Step 15. Select $E7$. $R7$: $i\,i\,x + y \longrightarrow x + y$
Since $i\,i\,x + y \longrightarrow_{R11} x + y$, rule $R7$ is deleted.
$\mathcal{R}_{\mathcal{R}} = \{R1, R2, R3, R4, R6', R10, R11\}$;
$\mathcal{E} = \{E12, E13\}$.

Step 16. Select $E12$. $R12$: $x + i\,x \longrightarrow 0$
$\mathcal{R}_{\mathcal{E}} = \{\}$; $\mathcal{R}_{\mathcal{R}} = \{R1, R2, R3, R4, R6', R10, R11, R12\}$;
$\mathcal{E} = \{E13\} \cup \{E14\}$, where:
$E14$: $x + (y + i(x + y)) = 0$, because:

$(x + y) + i(x + y) \begin{array}{l}\longrightarrow_{R3} \\ \longrightarrow_{R12}\end{array} \begin{array}{l}x + (y + i(x + y)) \\ 0\end{array}$

Step 17. Select $E13$. $R13$: $x + (i\,x + y) \longrightarrow y$
$\mathcal{R}_{\mathcal{E}} = \{\}$; $\mathcal{R}_{\mathcal{R}} = \{R1, R2, R3, R4, R6', R10, R11, R12, R13\}$;
$\mathcal{E} = \{E14\}$.

Step 18. Select $E14$. $R14$: $x + (y + i(x + y)) \longrightarrow 0$
$\mathcal{R}_{\mathcal{E}} = \{\}$; $\mathcal{R}_{\mathcal{R}} = \{R1, R2, R3, R4, R6', R10, R11, R12, R13, R14\}$;
$\mathcal{E} = \{\} \cup \{E15\}$, where:
$E15$: $x + i(y + x) = i\,y + 0$, because:

$i\,y + (y + (x + i(y + x))) \begin{array}{l}\longrightarrow_{R4} \\ \longrightarrow_{R14}\end{array} \begin{array}{l}x + i(y + x) \\ i\,y + 0\end{array}$

Step 19. Select $E15$. $R15$: $x + i(y + x) \longrightarrow i\,y$ (because $i\,y + 0 \longrightarrow_{R6'} i\,y$)
$\mathcal{R}_{\mathcal{E}} = \{R14\}$; $\mathcal{R}_{\mathcal{R}} = \{R1, R2, R3, R4, R6', R10, R11, R12, R13, R15\}$;
$\mathcal{E} = \{E14\} \cup \{E16\}$, where:
$E16$: $i(y+x) = i\,x+i\,y$, because: $i\,y + (y + (x + i(y + x))) \begin{array}{l}\longrightarrow_{R4} \\ \longrightarrow_{R15}\end{array} \begin{array}{l}i(y + x) \\ i\,x + i\,y\end{array}$

Step 20. Select $E14$. $R14$: $x + (y + i(x + y)) \longrightarrow 0$
Since $x + (y + i(x + y)) \longrightarrow_{R15} x + i\,x \longrightarrow_{R12} 0$, rule $R14$ is deleted.
$\mathcal{R}_{\mathcal{R}} = \{R1, R2, R3, R4, R6', R10, R11, R12, R13, R15\}$;
$\mathcal{E} = \{E16\}$.

Step 21. Select $E16$. $R16$: $i(y + x) \longrightarrow i\,x + i\,y$
$\mathcal{R}_{\mathcal{E}} = \{R15\}$; $\mathcal{R}'' = \{R1, R2, R3, R4, R6', R10, R11, R12, R13, R16\}$;
$\mathcal{E} = \{\} \cup \{E15\}$.

Step 22. Select $E15$. $R15$: $x + i(y + x) \longrightarrow i\,y$
Since $x + i(y + x) \longrightarrow_{R16} x + (i\,x + i\,y) \longrightarrow_{R13} i\,y$, rule $R15$ is deleted.
$\mathcal{R}_{\mathcal{R}} = \{R1, R2, R3, R4, R6', R10, R11, R12, R13, R16\}$;
$\mathcal{E} = \{\}$.

Since $\mathcal{E}$ is empty, the Knuth-Bendix Completion algorithm terminates and the canonical term rewriting system for the group axioms is given by the ten rules (see the boxed rules in Table 1 on page 52):

$R1$, $R2$, $R3$, $R4$, $R6'$, $R10$, $R11$, $R12$, $R13$, and $R16$.

We can use this rewriting system for proving, for instance, that $x+0 = 0+x$. Indeed, for the left hand side we have that: $x + 0 \longrightarrow_{R6'} x$ and for the right hand side we have that: $0 + x \longrightarrow_{R1} x$.

10. Checking Tautologies of the Propositional Calculus

In this section we present a method for checking tautologies of the Propositional Calculus [**6**]. This method is based on the construction of a canonical term rewriting system constructed from the axioms of the Propositional calculus as explained in the previous section.

In order to check whether or not a given propositional formula φ which uses the operators $\neg$ (not), $\vee$ (or), $\wedge$ (and), $\rightarrow$ (implies), and $\leftrightarrow$ (equivalent to), is a tautology, we first get an equivalent formula, call it ψ, which, instead of those operators, uses the operators $+$ (plus, or symmetric difference), and $\times$ (times, or conjunction) only.

We can derive the formula ψ from the formula φ by applying the following transformations:

$$
\begin{array}{lcl}
\neg\alpha & \Rightarrow & 1 + \alpha \\
\alpha \vee \beta & \Rightarrow & \alpha + \beta + (\alpha \times \beta) \\
\alpha \wedge \beta & \Rightarrow & \alpha \times \beta \\
\alpha \rightarrow \beta & \Rightarrow & (\alpha \times \beta) + \alpha + 1 \\
\alpha \leftrightarrow \beta & \Rightarrow & \alpha + \beta + 1
\end{array}
$$

In these transformations we have that:

(i) 1 stands for the predicate symbol *true* (see page 15), and
for all formulas α and β,

(ii) $\alpha + \beta$ is equivalent to $(\neg\alpha \wedge \beta) \vee (\alpha \wedge \neg\beta)$, and

(iii) $\alpha \times \beta$ is equivalent to $\alpha \wedge \beta$.

Then, we rewrite the derived formula ψ by using the following canonical rewriting system $\mathcal{R}_{\mathcal{PROP}}$, where 0 stands for the predicate symbol *false* (see page 15).

Canonical rewriting system $\mathcal{R}_{\mathcal{PROP}}$ for the Propositional Calculus
(+ and × are associative and commutative)

$$
\begin{array}{lcl}
\alpha + 0 & \longrightarrow & \alpha \\
\alpha + \alpha & \longrightarrow & 0 \\
\alpha \times 1 & \longrightarrow & \alpha \\
\alpha \times \alpha & \longrightarrow & \alpha \\
\alpha \times 0 & \longrightarrow & 0 \\
\alpha \times (\beta + \gamma) & \longrightarrow & (\alpha \times \beta) + (\alpha \times \gamma)
\end{array}
$$

When, starting from ψ, by using this rewriting system we get an irreducible formula (modulo associativity and commutativity of + and ×), we decide the validity, unsatisfiability, and satisfiability of ψ (and thus, of the given equivalent formula φ) by applying the following Theorem 10.1.

THEOREM 10.1. [**Propositional Theorem Prover via Rewritings**] For every propositional formula ψ, let us consider its irreducible formula $\widetilde{\psi}$ using the canonical rewriting system $\mathcal{R}_{\mathcal{PROP}}$. Then, (i) ψ is a tautology, that is, a valid formula, iff $\widetilde{\psi}$ is 1, (ii) ψ is an unsatisfiable formula iff $\widetilde{\psi}$ is 0, and (iii) ψ is a satisfiable formula and not a valid formula iff $\widetilde{\psi}$ is neither 0 nor 1.

EXAMPLE 10.2. Let us show that $p \vee (p \wedge q) \leftrightarrow p$ is a valid formula. Indeed, we have that $(p \vee (p \wedge q) \leftrightarrow p) \;\Rightarrow\; (p \times (p \times q) + p + (p \times q)) + p + 1$, and this last formula can be rewritten as follows (in some of these rewriting steps we have silently applied commutativity and associativity of + and ×):

$$
\begin{array}{ll}
(p \times (q \times p) + p + (p \times q)) + p + 1 & \{\text{by } p{\times}p \longrightarrow p\} \\
\longrightarrow ((p \times q) + p + (p \times q)) + p + 1 & \{\text{by } (p{\times}q) + (p{\times}q) \longrightarrow 0\} \\
\longrightarrow (0 + p) + p + 1 & \{\text{by } p{+}0 \longrightarrow p\} \\
\longrightarrow p + p + 1 & \{\text{by } p{+}p \longrightarrow 0\} \\
\longrightarrow 0 + 1 & \{\text{by } p{+}0 \longrightarrow p\} \\
\longrightarrow 1 & \square
\end{array}
$$

The canonical rewriting system $\mathcal{R}_{\mathcal{PROP}}$ for the Propositional Calculus can be derived from the equality axioms of the boolean rings by using the Knuth-Bendix Completion algorithm as we will see below. These equality axioms are the following ones.

Axioms for Boolean Rings

$$
\begin{array}{lcll}
\alpha + 0 & = & \alpha & \\
\alpha + (-\alpha) & = & 0 & \\
\alpha + \alpha & = & 0 & \\
\alpha + (\beta + \gamma) & = & (\alpha + \beta) + \gamma & \text{(assoc.)} \\
\alpha + \beta & = & \beta + \alpha & \text{(comm.)}
\end{array}
\qquad
\begin{array}{lcll}
\alpha \times 1 & = & \alpha & \\
\alpha \times \alpha & = & \alpha & \\
\alpha \times (\beta \times \gamma) & = & (\alpha \times \beta) \times \gamma & \text{(assoc.)} \\
\alpha \times \beta & = & \beta \times \alpha & \text{(comm.)} \\
\alpha \times (\beta + \gamma) & = & (\alpha \times \beta) + (\alpha \times \gamma) &
\end{array}
$$

where: (i) $-\alpha$ denotes the *additive inverse* of α (which is α itself as we will now show), (ii) 0 is the *additive unit*, (iii) 1 is the *multiplicative unit*.

Here is the proof that $-\alpha = \alpha$.

$\alpha + (-\alpha) = 0$ iff {by adding α to both sides} iff $\alpha + \alpha + (-\alpha) = \alpha$
iff {by associativity and $\alpha+\alpha=0$} iff $0 + (-\alpha) = \alpha$
iff {by commutativity and $\alpha+0 = \alpha$} iff $-\alpha = \alpha$.

Here is the proof that $\alpha \times 0 = 0$.

$\alpha \times 0 = \{\text{by } \alpha+\alpha=0\} = \alpha \times (\alpha + \alpha) = \{\text{by distributivity of } \times \text{ over } +\} =$
$= (\alpha\times\alpha) + (\alpha\times\alpha) = \{\text{by } \alpha+\alpha=0\} = 0.$

From a given formula which uses the operators $-$ (unary minus), $+$ (plus) and $\times$ (times), we can get an equivalent formula which uses, instead, the operators $\neg$ (not), $\vee$ (or), and $\wedge$ (and), by applying the following transformations:

$$\begin{array}{lcl} -\alpha & \Rightarrow & \alpha \\ \alpha + \beta & \Rightarrow & (\neg\alpha \wedge \beta) \vee (\alpha \wedge \neg\beta) \\ \alpha \times \beta & \Rightarrow & \alpha \wedge \beta \end{array}$$

Now we derive a canonical term rewriting system for the axioms of boolean rings using the Knuth-Bendix Completion algorithm.

We start off from the following equality axioms:

$E1$. $\alpha + 0 = \alpha$
$E2$. $\alpha + \alpha = 0$
$E3$. $\alpha \times 1 = \alpha$
$E4$. $\alpha \times \alpha = \alpha$
$E5$. $\alpha \times (\beta + \gamma) = (\alpha \times \beta) + (\alpha \times \gamma)$

Initially, we have that: $\mathcal{R} = \emptyset$ and $\mathcal{E} = \{E1, E2, E3, E4, E5\}$.

Note that, with reference to the axioms of boolean rings listed on page 55, we did not include the associativity and commutativity axioms for $+$ and $\times$ do not occur among the axioms $E1$–$E5$. This is not a problem because we deal with associativity and commutativity by assuming that the notions of *instance* and *matching* have to be understood modulo associativity and commutativity. Indeed, it has been shown that the correctness of the Knuth-Bendix Completion algorithm also holds when, instead of unification and matching, one uses unification and matching modulo associativity and commutativity.

Note also that in the above axioms $E1$–$E5$ we did not include the axiom $\alpha + (-\alpha) = 0$ either, because, as we have already proved, $-\alpha$ is α and, thus, the axiom $\alpha + (-\alpha) = 0$ reduces to $E2$.

Termination of the canonical term rewriting system can be proved by using the following lexicographic measure $\mu(t)$ of a term t (the first component of $\mu(t)$ is for rule $R5$ which increases the size of the term t):

$$\mu(t) =_{def} \langle \textit{multisets of sizes of right operands of } \times, \textit{size}(t) \rangle.$$

On Table 2 we have listed the various rewriting rules which are computed by the algorithm.

Step 1. Select $E1$. $R1$: $\alpha + 0 \longrightarrow \alpha$
$\mathcal{R}_{\mathcal{E}} = \emptyset$; $\mathcal{R}_{\mathcal{R}} = \{R1\}$;
$\mathcal{E} = \{E2, E3, E4, E5\}$.

Step 2. Select $E2$. $R2$: $\alpha + \alpha \longrightarrow 0$
$\mathcal{R}_{\mathcal{E}} = \emptyset$; $\mathcal{R}_{\mathcal{R}} = \{R1, R2\}$;
$\mathcal{E} = \{E3, E4, E5\}$.

Step 3. Select $E3$. $R3$: $\alpha \times 1 \longrightarrow \alpha$
$\mathcal{R}_{\mathcal{E}} = \emptyset$; $\mathcal{R}_{\mathcal{R}} = \{R1, R2, R3\}$;
$\mathcal{E} = \{E4, E5\}$.

Step 4. Select $E4$. $R4$: $\alpha \times \alpha \longrightarrow \alpha$
$\mathcal{R}_{\mathcal{E}} = \emptyset$; $\mathcal{R}_{\mathcal{R}} = \{R1, R2, R3, R4\}$;
$\mathcal{E} = \{E5\}$.

Step 5. Select $E5$. $R5$: $\alpha \times (\beta + \gamma) \longrightarrow (\alpha \times \beta) + (\alpha \times \gamma)$
$\mathcal{R}_{\mathcal{E}} = \emptyset$; $\mathcal{R}_{\mathcal{R}} = \{R1, R2, R3, R4, R5\}$;
$\mathcal{E} = \{\} \cup \{E6\}$, where:
$E6$: $(\alpha \times \alpha) + (\alpha \times \alpha) = \alpha \times 0$, because:
$\alpha \times (\alpha + \alpha) \begin{array}{l} \longrightarrow_{R5} \\ \longrightarrow_{R2} \end{array} \begin{array}{l} (\alpha \times \alpha) + (\alpha \times \alpha) \\ \alpha \times 0 \end{array}$

Step 6. Select $E6$. $R6$: $\alpha \times 0 \longrightarrow 0$
(because $(\alpha \times \alpha) + (\alpha \times \alpha) \longrightarrow_{R2} 0$)
$\mathcal{R}_{\mathcal{E}} = \emptyset$; $\mathcal{R}_{\mathcal{R}} = \{R1, R2, R3, R4, R5, R6\}$;
$\mathcal{E} = \{\}$.

Since $\mathcal{E}$ is empty, the Knuth-Bendix Completion algorithm terminates and the canonical term rewriting system for the axioms of boolean rings is given by the six rules: $R1$, $R2$, $R3$, $R4$, $R5$, and $R6$ (see Table 2).

$R1$. $\alpha + 0 \longrightarrow \alpha$	$R4$. $\alpha \times \alpha \longrightarrow \alpha$
$R2$. $\alpha + \alpha \longrightarrow 0$	$R5$. $\alpha \times (\beta + \gamma) \longrightarrow (\alpha + \beta) \times (\alpha + \gamma)$
$R3$. $\alpha \times 1 \longrightarrow \alpha$	$R6$. $\alpha \times 0 \longrightarrow 0$

TABLE 2. Rewriting rules computed by the Knuth-Bendix Completion algorithm starting from the following equality axioms for boolean rings: $E1$: $\alpha + 0 = \alpha$, $E2$: $\alpha + \alpha = 0$, $E3$: $\alpha \times 1 = \alpha$, $E4$: $\alpha \times \alpha = \alpha$, and $E5$: $\alpha \times (\beta + \gamma) = (\alpha \times \beta) + (\alpha \times \gamma)$.

In Figure 4 on the following page we present a second version of the Knuth-Bendix Completion algorithm which is sometimes found in the literature. This version has been proposed by Nachum Dershowitz and explicitly refers to a well-founded ordering, denoted $>$, required for showing strong termination of the canonical rewriting system $\mathcal{R}$ to be constructed.

Knuth-Bendix Completion. Version 2.

Input: (i) a (possibly empty) finite set $\mathcal{E}$ of equations, and
(ii) a procedure for checking whether or not $t_1 > t_2$, for a given well-founded ordering relation $>$ and any two terms $t_1, t_2 \in T(F \cup \mathit{Vars})$.

$\mathcal{R} := \emptyset$;
while $\mathcal{E} \neq \emptyset$ **do**

1. Remove from $\mathcal{E}$ an equation, which occurs in $\mathcal{E}$ as $M = N$ or $N = M$, such that $M > N$ and $\mathit{vars}(M) \supseteq \mathit{vars}(N)$. If no such equation exists, then stop with failure.
2. Add to $\mathcal{R}$ rule ρ: $M \to N$.
3. Use $\mathcal{R}$ to compute the normal forms of the right hand sides of the rules in $\mathcal{R}$.
4. Add to $\mathcal{E}$ all critical pairs generated in $\mathcal{R}$ by the new rule ρ.
5. (Optional Step) Remove all rules in $\mathcal{R} - \{\rho\}$ whose left hand side contains an instance of M.
6. Use $\mathcal{R}$ to compute the normal forms both sides of the equations in $\mathcal{E}$. Remove any equation in $\mathcal{E}$ whose sides are identical.

od

Output: a canonical rewriting system $\mathcal{R}$ such that
$\forall t_1, t_2 \in T(F \cup \mathit{Vars}),\ t_1 =_{\mathcal{E}} t_2$ iff $\mathcal{R}(t_1) \equiv \mathcal{R}(t_2)$.

FIGURE 4. Knuth-Bendix Completion algorithm. Version 2.

We are given: (i) a finite set $\mathcal{E}$ of equations, and (ii) a procedure for checking whether or not $t_1 > t_2$, for a given well-founded ordering relation $>$ and any two terms $t_1, t_2 \in T(F \cup \mathit{Vars})$.

The procedure of Point (ii) was implicitly given in the first version of the Knuth-Bendix Completion algorithm (see Figure 3 on page 49) and it is required for orienting the rewriting rules so that strong termination of the rewriting system $\mathcal{R}$ is guaranteed.

Also for this version of Knuth-Bendix Completion algorithm, we have that when it terminates, it generates a canonical rewriting system $\mathcal{R}$ for $\mathcal{E}$ such that

$$\forall t_1, t_2 \in T(F \cup \mathit{Vars}),\ t_1 =_{\mathcal{E}} t_2 \text{ iff } \mathcal{R}(t_1) \equiv \mathcal{R}(t_2).$$

REMARK 10.3. If we considered the equality axioms $E1$–$E5$ together with the axiom $\alpha + (-\alpha) = 0$, then the Knuth-Bendix Completion algorithm produces the rewriting rules $R1$–$R6$ listed in Table 2 on the preceding page together with the rule $R7$: $\alpha + (-\alpha) \longrightarrow 0$. Strong termination of the rewriting system made out of the rules $R1$–$R7$ is a consequence of the following two facts: (i) the system made out of the rules $R1$–$R6$ is strongly terminating, and (ii) the size of the rewritten term is reduced when applying $R7$. □

CHAPTER 4

Induction Rules and Semantic Domains

In this chapter we introduce: (i) some induction rules, (ii) some basic theorems, and (iii) some mathematical domains, which will be useful in later chapters for defining the semantics of imperative and functional programming languages.

1. Induction Rules

When we have to prove properties of finite sets we can proceed by examining their elements one at a time. But if the sets are infinite, we cannot proceed that way, because proofs should be finite objects. We need deduction rules which allow us to infer properties of infinite sets.

These deduction rules are of various kinds. In this section we will consider some of them for different kinds of infinite sets.

In this section N denotes the set of natural numbers with the nullary constructor 0 and the unary constructor successor function s from N to N. In order to avoid confusion between the evaluation relation, denoted $\rightarrow$ (see Section 1.4 on page 64), and the logical implication, also denoted $\rightarrow$, in this section we will denote the logical implication by $\Rightarrow$.

1.1. Mathematical Induction.

For the set of natural numbers as axiomatized by Peano Arithmetics, we have already seen (see page 19) the *mathematical induction* rule which is as follows. Let us consider a predicate $P(n)$ over the set N of the natural numbers. In order to prove that $\forall k \in N.\, P(k)$ by mathematical induction it is enough to show:

(i) $P(0)$ and (ii) $\forall n \in N.\, (P(n) \Rightarrow P(s(n)))$.

This rule is denoted as follows.

(Mathematical Induction)

$$\frac{P(0) \qquad \forall n \in N.\, (P(n) \Rightarrow P(s(n)))}{\forall k \in N.\, P(k)}$$

In order to avoid *regressio ad infinitum*, the second premise of mathematical induction can be proved by using the generalization rule of First Order Predicate Calculus (see page 17).

For a predicate $P(m, n)$ with two arguments over the set $N \times N$, the mathematical induction rule is as follows.

(Mathematical Induction with two arguments)

$$\frac{P(0,0) \quad \forall n \in N.\,(P(0,n) \Rightarrow P(0,s(n))) \qquad \forall m,n \in N.\,(P(m,n) \Rightarrow P(s(m),n))}{\forall h,k \in N.\,P(h,k)}$$

Now we derive this rule from the mathematical induction rule for predicates with one argument only.

Let us consider the predicate $Q(m) =_{def} \forall n \in N.\,P(m,n)$. It has one argument only. In order to show that $\forall m \in N.\,Q(m)$ by mathematical induction we need to show: (i) $Q(0)$, and (ii) $\forall m \in N.\,Q(m) \Rightarrow Q(s(m))$.

(i) In order to show $Q(0)$, that is, $\forall n \in N.\,P(0,n)$, by mathematical induction for predicates with one argument only, we need to show:

(α) $P(0,0)$ and (β) $\forall n \in N.\,(P(0,n) \Rightarrow P(0,s(n)))$.

(ii) In order to show $\forall m \in N.\,Q(m) \Rightarrow Q(s(m))$, that is, $\forall m \in N.\,\big(\forall n \in N.\,P(m,n)\big) \Rightarrow \big(\forall n \in N.\,P(s(m),n)\big)$, it is enough to show:

(γ) $\forall m,n \in N.\,(P(m,n) \Rightarrow P(s(m),n))$,

because for all binary predicates A and B, we have that $\forall m,n.\,(A(m,n) \Rightarrow B(m,n))$ implies $\forall m.\,(\forall n.\,A(m,n)) \Rightarrow (\forall n.\,B(m,n))$.

This concludes the proof of the mathematical induction rule for predicates with two arguments from the mathematical induction rule for predicates with one argument only.

Exercise 1.1. Prove by mathematical induction that, for all $n \geq 0$,

(i) $\sum_{i=0}^{n} i = \dfrac{n \cdot (n+1)}{2}$, and (ii) $\sum_{i=0}^{n} i^2 = \dfrac{n}{3} \cdot (n+1) \cdot (n+\dfrac{1}{2})$.

Exercise 1.2. Prove by mathematical induction that for all $n \geq 1$,

$$\frac{a_1 + \ldots + a_n}{n} \geq \sqrt[n]{a_1 \cdot \ldots \cdot a_n}.$$

Now we will prove by mathematical induction with two arguments the commutativity of addition, denoted $+$, on natural numbers. We will show the following theorem.

Theorem 1.3. **[Commutativity of Plus]** For all natural numbers m and n, we have that $m+n = n+m$, where the $+$ operation is defined by primitive recursion as follows:

$$\begin{array}{lll} (P0) & \forall n. & 0+n = n \\ (Ps) & \forall m,n. & s(m)+n = s(m+n) \end{array}$$

Proof. By mathematical induction for a predicate with two arguments. We have to prove the following three facts:

$$\begin{array}{lll} (F1) & & 0+0 = 0+0 \\ (F2) & \forall n. & 0+n = n+0 \;\Rightarrow\; 0+s(n) = s(n)+0 \\ (F3) & \forall m,n. & m+n = n+m \;\Rightarrow\; s(m)+n = n+s(m) \end{array}$$

Fact $(F1)$ follows from $(P0)$ because both sides are equal to 0. For Fact $(F2)$ let us consider a generic value n and the two sides $0{+}s(n)$ and $s(n){+}0$ of the conclusion.

Now, $0{+}s(n) = \{\text{by } (P0)\} = s(n)$, and

$s(n){+}0 = \{\text{by } (Ps)\} =$

$= s(n{+}0) = \{\text{by inductive hypothesis (see premise } 0{+}n{=}n{+}0 \text{ in Fact } (F2))\} =$

$= s(0{+}n) = \{\text{by } (P0)\} =$

$= s(n).$

Since n is a generic value, by generalization (see page 17) we get Fact $(F2)$.

Similarly, for Fact $(F3)$ let us consider the two generic values m and n. We have that:

$s(m){+}n = \{\text{by } (Ps)\} =$

$= s(m{+}n) = \{\text{by inductive hypothesis (see premise } m{+}n{=}n{+}m \text{ in Fact } (F3))\} =$

$= s(n{+}m) = \{\text{by Lemma 1.4}\} =$

$= n{+}s(m).$

Since m and n are generic values, by generalization we get Fact $(F3)$. □

LEMMA 1.4. For all natural numbers m and n, we have that $s(m{+}n) = m{+}s(n)$.

PROOF. We proceed by mathematical induction on the first argument m of the addition function $\lambda m, n.m + n$ which is defined by primitive recursion on that first argument (see the statement of Theorem 1.3 on the preceding page). This choice of the induction variable is crucial. Indeed, the reader may verify that a proof of this lemma by induction on the second argument n does not go through.

(*Basis*) We take $m{=}0$. We have to show that for all $n \in N$, $s(0{+}n) = 0{+}s(n)$.

This follows from the fact that by $(P0)$ the left hand side $s(0{+}n)$ and the right hand side $0{+}s(n)$ are both equal to $s(n)$.

(*Step*) We assume that for all $m, n \in N$, $s(m{+}n) = m{+}s(n)$.

We have to show that for all $m, n \in N$, $s(s(m){+}n) = s(m){+}s(n)$.

Let us take two generic values m and n. For left hand side we have that:

$s(s(m){+}n) = \{\text{by } (Ps)\} = s(s(m{+}n))$.

For right hand side we have that: $s(m){+}s(n) = \{\text{by } (Ps)\} =$

$= s(m{+}s(n)) = \{\text{by inductive hypothesis}\} =$

$= s(s(m{+}n))$.

Thus, the left and the right hand sides are equal. □

1.2. Complete Induction.

For Peano Arithmetics one may also consider the induction rule called *complete induction* which is as follows. Let us consider a predicate $P(n)$ over the set N of natural numbers.

(Complete Induction)

$$\frac{\forall n \in N.\, ((\forall h \in N.\, h < n \Rightarrow P(h)) \Rightarrow P(n))}{\forall k \in N.\, P(k)}$$

As in the case of mathematical induction, in order to avoid *regressio ad infinitum*, the premise of this rule can be proved by using the generalization rule. If we use complete induction, instead of mathematical induction, the proof of some properties of the natural numbers may become shorter and simpler. However, Theorem 1.6 and Theorem 1.7 on the next page show that complete induction is equivalent to mathematical induction.

REMARK 1.5. For the expert reader we should say that complete induction is equivalent to mathematical induction only 'below' the first limit ordinal ω, because 'above' ω, complete induction is more powerful than mathematical induction. This is due to the fact that, above ω, the complete induction rule uses, so to speak, an infinite number of premises, while mathematical induction uses a finite number of premises only. We will not discuss further this issue here. □

The following theorem shows that complete induction is not more powerful than mathematical induction.

THEOREM 1.6. Every proof by complete induction can be replaced by a proof by mathematical induction.

PROOF. Assume that we have a proof (that is, a sequence of formulas) π ending by the following formula (with k as a free variable):

$$(\forall n \in N.\, (n < k \Rightarrow P(n))) \Rightarrow P(k). \qquad (A1)$$

From $(A1)$, by generalization (recall also that $\forall k.\, \varphi(k)$ implies $\forall k \in N.\, \varphi(k)$), we get:

$$\forall k \in N.\, (\forall n \in N.\, (n < k \Rightarrow P(n))) \Rightarrow P(k) \qquad (A2)$$

and from $(A2)$, by complete induction, we get:

$$\forall k \in N.\, P(k). \qquad (A3)$$

Now let us consider the property $P'(k) =_{def} \forall n \in N.\, (n < k \Rightarrow P(n))$.
We have that $P'(0)$ is $\forall n \in N.\, (n < 0 \Rightarrow P(n))$. Thus, $P'(0)$ holds.
We also have: $P'(k) \Rightarrow P'(s(k))$ iff

$P'(k) \Rightarrow [\forall n \in N.\, (n < s(k) \Rightarrow P(n))]$ iff

$P'(k) \Rightarrow [P'(k) \wedge P(k)]$ iff {by $a \Rightarrow a \wedge b$ iff $a \Rightarrow b$}

$P'(k) \Rightarrow P(k)$, which is $(A1)$.

Thus, the proof π, being a proof of $P'(k) \Rightarrow P(k)$, is also a proof of $P'(k) \Rightarrow P'(s(k))$.
Since $P'(0)$ holds, by mathematical induction we get $\forall k \in N.\, P'(k)$, that is,

$$\forall k \in N.\, (\forall n \in N.\, (n < k \Rightarrow P(n))). \qquad (A4)$$

Having derived $(A4)$, we can prove $(A3)$ by mathematical induction as follows.

(*Basis*) From $(A4)$ for $k = 1$ we get: $(\forall n \in N.\, (n < 1 \Rightarrow P(n))$, that is, $P(0)$.

(*Step*) Let us assume $P(h)$ for some $h \in N$, and let us show $P(s(h))$.

From $(A4)$ for $k=s(s(h))$ we have: $\forall n\in N.\,(n<s(s(h)) \Rightarrow P(n))$. Now take $n=s(h)$ and we have: $s(h)<s(s(h)) \Rightarrow P(s(h))$ which is equivalent to $P(s(h))$. (Note that in order to prove $P(s(h))$, we did not use the hypothesis $P(h)$.)

This completes the proof of the theorem. □

We also have the following theorem which shows that mathematical induction is not more powerful than complete induction.

THEOREM 1.7. Every proof by mathematical induction can be replaced by a proof by complete induction.

PROOF. Suppose by hypothesis that we have a proof of the two formulas:

(i) $P(0)$ and (ii) $\forall n\in N.\,[P(n) \Rightarrow P(s(n))]$.

From (ii) and the fact that $\forall k\in N.\,k<s(n) \Rightarrow P(k)$ implies $P(n)$, we get (recall that if $a \Rightarrow a'$ then $a' \Rightarrow b$ implies $a \Rightarrow b$):

$\forall n\in N.\,[(\forall k\in N.\,k<s(n) \Rightarrow P(k)) \Rightarrow P(s(n))]$.

Then by complete induction we get, as desired, $\forall k\in N.\,P(k)$. □

1.3. Structural Induction.

For sets which are generated by context-free productions, we have an induction rule called *structural induction.*

To fix our ideas, let us see this rule in action in the following example which refers to the set B of binary trees with natural numbers on the nodes. The set B is generated by the following productions, where $n \in N$ and $b, b_1, b_2 \in B$:

$$b \ ::= \ e \ \mid \ \ell(n) \ \mid \ \langle b_1, n, b_2\rangle \qquad \text{(Binary Trees)}$$

where: (i) $n \in N$, (ii) e is the empty binary tree, (iii) the function $\lambda n.\,\ell(n)$ is the unary constructor from N to B for generating a leaf with the natural number n, and (iv) the function $\lambda b_1, n, b_2.\langle b_1, n, b_2\rangle$ is the ternary constructor for generating a node with a natural number n and two son-nodes, the left son-node b_1 and the right son-node b_2.

(Structural Induction for Binary Trees)

$$\frac{P(e) \qquad \forall n\in N.\,P(\ell(n)) \qquad \forall n\in N.\,\forall b_1, b_2\in B.\,P(b_1)\wedge P(b_2) \Rightarrow P(\langle b_1, n, b_2\rangle)}{\forall b\in B.\,P(b)}$$

Structural induction can be viewed as a generalization of mathematical induction to sets generated by context-free grammars. In particular, when we consider the set N of natural numbers as generated by the following two productions, where $n \in N$:

$$n \ ::= \ 0 \ \mid \ s(n) \qquad \text{(Natural Numbers)}$$

we get that structural induction for the set N is mathematical induction.

In Example 6.16 on page 188 and Example 6.18 on page 189 we will see in action the structural induction rule.

1.4. Rule Induction.

In the literature [**19**] one finds the so called *rule induction*, that is used for proving properties of sets which are generated by rules, as we now indicate. Let us first see this induction rule in action in the following example.

Let us consider the set **BasicAexp** of *basic arithmetic expressions* over the set N of natural numbers. It is defined as follows, where $n \in N$ and $a, a_1, a_2 \in \mathbf{BasicAexp}$:

$$a ::= n \mid a_1 + a_2$$

The operational semantics of basic arithmetic expressions is specified by a subset of $\mathbf{BasicAexp} \times N$, and a pair $\langle a, n\rangle$ in $\mathbf{BasicAexp} \times N$, written $a \to n$, denotes that the basic arithmetic expression a evaluates to n.

The rules defining the operational semantics are as follows (the empty set of premises stands for *true* and it is also denoted by $\{\}$ (see page 65)): for all $n, n_1, n_2 \in N$, for all $a_1, a_2 \in \mathbf{BasicAexp}$, we have that:

(A1) $$\frac{}{n \to n}$$

(A2) $$\frac{a_1 \to n_1 \qquad a_2 \to n_2}{a_1 + a_2 \to add(n_1, n_2)}$$

where add is the usual addition operation between any two natural numbers.

Now we can show that a given property $P(a, n)$ holds for every pair $\langle a, n\rangle$ in the subset of $\mathbf{BasicAexp} \times N$ derived by the rules (A1) and (A2), by applying the following rule:

(Rule Induction for **BasicAexp**)

$$\frac{\begin{array}{l}\forall n, n_1, n_2 \in N.\, \forall a_1, a_2 \in \mathbf{BasicAexp}.\\ \quad P(n, n)\\ \wedge\ (a_1 \to n_1 \wedge P(a_1, n_1) \wedge a_2 \to n_2 \wedge P(a_2, n_2)) \Rightarrow P(a_1 + a_2, add(n_1, n_2))\end{array}}{\forall n \in N.\, \forall a \in \mathbf{BasicAexp}.\ P(a, n)}$$

Exercise 1.8. Prove by rule induction that the operational semantics of the basic arithmetic expressions is deterministic, that is, $\forall a \in \mathbf{BasicAexp}.\ \forall n_1, n_2 \in N.\ (a \to n_1 \wedge a \to n_2) \Rightarrow n_1 = n_2$.

Rule induction for **BasicAexp** can be shown to be equivalent to structural induction for **BasicAexp**. Indeed, let us consider the structural induction rule for the property $(a \to n) \wedge P(a, n)$ which refers to the evaluation of any expression $a \in \mathbf{BasicAexp}$ to the value n. The structural induction rule is as follows.

(Structural Induction for **BasicAexp**)

$$\frac{\begin{array}{l}\forall n, n_1, n_2 \in N.\, \forall a_1, a_2 \in \mathbf{BasicAexp}. \\ \quad (n \to n) \ \wedge\ P(n,n) \\ \wedge \big[[(a_1 \to n_1) \ \wedge\ P(a_1,n_1) \ \wedge\ (a_2 \to n_2) \ \wedge\ P(a_2,n_2)] \\ \quad \Rightarrow [(a_1{+}a_2 \to add(n_1,n_2)) \ \wedge\ P(a_1{+}a_2, add(n_1,n_2))]\big]\end{array}}{\forall n \in N.\, \forall a \in \mathbf{BasicAexp}.\ P(a,n)}$$

By rule (A1) we have that

$$\forall n \in N.\, n \to n \tag{1}$$

and by rule (A2) we have that

$$\forall n_1, n_2 \in N.\, \forall a_1, a_2 \in \mathbf{BasicAexp}. \tag{2}$$
$$((a_1 \to n_1) \wedge (a_2 \to n_2)) \ \Rightarrow\ ((a_1{+}a_2) \to add(n_1,n_2))$$

Thus, structural induction for **BasicAexp** is reduced, by using (1) and (2), to rule induction for **BasicAexp**.

Actually, structural induction can be reduced to rule induction, not only in the case of the set **BasicAexp**, but also in the case of every set which is generated by a context-free grammar.

Now let us consider rule induction in the general case. Suppose we are given a (possibly infinite) set of *rule instances* $\overline{R}$ generated by a given *finite* set R of *derivation rules*. Each rule instance in $\overline{R}$ is of the form $\dfrac{a_1, \ldots, a_n}{a}$, for some terms $a, a_1, \ldots a_n$, for some $n \geq 0$.

DEFINITION 1.9. [**Inductive Set**] Given the possibly infinite set $\overline{R}$ of rule instances generated by a *finite* set R of derivation rules, the *inductive set of* R, denoted I_R, is the following set of terms:

$I_R =_{def} \{a \mid \text{there exists } d \text{ such that } d \Vdash_R a\}$

where the binary relation $\Vdash_R$ is recursively defined as follows:

$$\frac{\{\}}{a} \Vdash_R a \quad \text{if} \quad \frac{\{\}}{a} \in \overline{R}, \quad \text{and}$$
$$\frac{\{d_1, \ldots, d_n\}}{a} \Vdash_R a \quad \text{if} \quad \frac{\{a_1, \ldots, a_n\}}{a} \in \overline{R} \text{ and } d_1 \Vdash_R a_1 \text{ and } \ldots \text{ and } d_n \Vdash_R a_n.$$

If for some d and a, $d \Vdash_R a$ holds, we say that d is an *R-derivation* (or simply, *derivation*, when R is understood from the context) of the term a. (Here the notion of a derivation is the analogous to that of Definition 2.1 on page 17 in the case of the First Order Predicate Calculus.) Obviously, if no rule in R has an empty set of premises then I_R is empty.

We will write $\Vdash_R a$ to mean that *there exists* an R-derivation d such that $d \Vdash_R a$. We will feel free to write $\Vdash a$, or simply a, instead of $\Vdash_R a$, whenever the set R of derivation rules is understood from the context.

Given two R-derivations d and d', we say that d' is an *immediate subderivation* of d, and we write $d' \prec d$, iff (i) d is of the form $\dfrac{D}{a}$ for some term a and for some set D of derivations, and (ii) $d' \in D$. As usual, by $\prec^+$ we denote the transitive closure of $\prec$. We say that d' is a *proper subderivation* of d iff $d' \prec^+ d$.

For example, let us consider the set $\overline{R} =_{def} \{\dfrac{\{\}}{a},\ \dfrac{\{a\}}{b},\ \dfrac{\{a,b\}}{c}\}$ of rule instances.

Here is the derivation of c: $\dfrac{\{\dfrac{\{\}}{a},\ \dfrac{\{\dfrac{\{\}}{a}\}}{b}\}}{c}$. The derivation $\dfrac{\{\dfrac{\{\}}{a}\}}{b}$ is an immediate subderivation of that derivation of c.

As a second example, let us consider the set R consisting of the following two derivation rules:

$$\text{(i)}\ \ \frac{\{\}}{0} \qquad\qquad \text{(ii)}\ \ \text{for any } n,\ \ \frac{\{n\}}{s(n)}$$

The set R generates the following infinite set $\overline{R}$ of rule instances:

$$\{\frac{\{\}}{0},\ \frac{\{0\}}{s(0)},\ \frac{\{s(0)\}}{s(s(0))},\ \ldots\}$$

and we have that the inductive set I_R is $\{0, s(0), s(s(0)), \ldots\}$, that is, the set of the natural numbers with the nullary constructor 0 and the unary constructor successor function s.

Let R be a set of derivation rules and $\overline{R}$ be the associated set of rule instances. Let I_R be the inductive set of R. Let $P(x)$ be a property. We have that:

$$\begin{array}{l}\forall x \in I_R.\, P(x) \quad \textit{iff} \\ \text{for all rule instances } \dfrac{X}{y} \in \overline{R} \text{ such that } X \subseteq I_R, \text{ we have that} \\ \quad (\forall x \in X. P(x)) \Rightarrow P(y).\end{array} \qquad (\dagger)$$

Thus, by taking the *if*-part of the above formula (†), we get the following rule.

(Rule Induction)

$$\frac{\text{for every rule instance } \dfrac{X}{y} \in \overline{R},\ \ (\forall x \in X.\ x \in I_R \wedge P(x)) \Rightarrow P(y)}{\forall x \in I_R.\, P(x)}$$

Note that in the premise of rule induction we require that $x \in I_R$, but we do *not* explicitly require that $y \in I_R$ because, by definition of I_R, if $\forall x \in X.\, x \in I_R$ then $y \in I_R$.

In the case of the natural numbers, rule induction is equivalent to mathematical induction. Indeed, (i) $I_R = N$, (ii) from the rule instance $\dfrac{\{\}}{0}$ we get the premise $P(0)$, and (iii) for each $n \in N$, from the rule instance $\dfrac{\{n\}}{s(n)}$ we get the premise

$n \in N \wedge P(n) \Rightarrow P(s(n))$. Thus, we get $P(0) \wedge \forall n \in N.\, P(n) \Rightarrow P(s(n))$ which is the premise of mathematical induction.

Now we prove a completeness result for rule induction (see Theorem 1.13 on the following page). By this result rule induction is shown to be a necessary and a sufficient condition for proving *all* first order properties of the inductive sets.

Let us start by introducing the following Definition 1.10 and Theorem 1.11.

DEFINITION 1.10. [$\overline{R}$**-closed Set**] Given a set $\overline{R}$ of rule instances, a set C is said to be $\overline{R}$*-closed* iff for every rule instance $\dfrac{X}{y} \in \overline{R}$, if $X \subseteq C$ then $y \in C$.

THEOREM 1.11. [**Inductive Set Theorem**] Given a set $\overline{R}$ of rule instances, I_R is the least $\overline{R}$-closed set, that is, (i) I_R is a $\overline{R}$-closed, and (ii) given any set Z, if Z is an $\overline{R}$-closed set then $I_R \subseteq Z$.

PROOF. (i) We have to show that for every rule instance $\dfrac{X}{y} \in \overline{R}$, if $X \subseteq I_R$ then $y \in I_R$. We have two cases.

Case (i.1). Suppose that $X = \{\}$. We have that $y \in I_R$, because there is a derivation which shows that y is in I_R and that derivation is obtained by applying the rule instance $\dfrac{\{\}}{y}$ once only.

Case (i.2). Suppose that $X = \{x_1, \ldots, x_n\}$ with $n > 0$. Since $X \subseteq I_R$, for each $x_i \in X$ there is a derivation of x_i, call it d_i, which uses rule instances in $\overline{R}$. Now, we have a derivation of y of the form: $\dfrac{\{d_1,\ \ldots,\ d_n\}}{y}$ and, thus, $y \in I_R$.

(ii) Take any $\overline{R}$-closed set Z. We have to show that $I_R \subseteq Z$. Take any $y \in I_R$. We have to show that $y \in Z$. Now, since $y \in I_R$ there exists a derivation of y. We have two cases.

Case (ii.1). Suppose there exists a rule instance of the form $\dfrac{\{\}}{y}$.

Since Z is an $\overline{R}$-closed set and $\{\} \subseteq Z$ we get that $y \in Z$ by the rule instance $\dfrac{\{\}}{y}$.

Case (ii.2). Suppose there exists a rule instance of the form $\dfrac{\{x_1, \ldots, x_n\}}{y}$ for some $n > 0$. Since $y \in I_R$ we have n derivations which justify that $x_1, \ldots, x_n$ are in I_R. By well-founded induction on derivations (see Section 1.6 on page 72), we have that for $i = 1, \ldots, n$, $x_i \in Z$. Since there is a rule instance whose premises are all in Z, and Z is $\overline{R}$-closed, we have that also the conclusion of the rule instance, that is y, is in Z.

This concludes the proof of the theorem. □

REMARK 1.12. Rule induction allows very general forms of rules. In particular, for the set N of natural numbers we may assume the following set of rules:

(i) $\dfrac{\{\}}{0}$ (ii) for any n, $\dfrac{\{n\}}{s(n)}$ (iii) for any n, $\dfrac{\{n\}}{n}$

In this case for each natural number n we have finite derivations (which are the ones which justify the fact that n belongs to the inductive set N) and also unbounded derivations (which can be constructed by using the rules of the form (iii)). □

THEOREM 1.13. [**Completeness of Rule Induction**] Given a set $\overline{R}$ of rule instances, rule induction is a necessary and sufficient rule for showing every first order property $P(x)$ that holds for every element x of the inductive set I_R.

PROOF. (i) First we show that rule induction is *necessary* to prove any first order property $P(x)$ that holds for every element $x \in I_R$. We show this by assuming the conclusion of rule induction, that is, $\forall x \in I_R.\, P(x)$, and by proving the premise, that is, for every rule instance $\dfrac{X}{y} \in \overline{R}$, if $(\forall x \in X.\, x \in I_R \wedge P(x))$ then $P(y)$.

Thus, we assume $\forall x \in I_R.\, P(x)$. We consider a rule instance $\dfrac{X}{y} \in \overline{R}$.

We assume $\forall x \in X.\, x \in I_R \wedge P(x)$ and we have to show $P(y)$. Indeed, by Theorem 1.11 (i) on page 67, I_R is $\overline{R}$-closed and thus, we have that $y \in I_R$. Then, by the assumption that $\forall x \in I_R.\, P(x)$, since $y \in I_R$, we get $P(y)$.

(ii) We show that rule induction is *sufficient* to prove any first order property $P(x)$ that holds for every element $x \in I_R$. We have to show that by using rule induction we can prove $\forall x \in I_R.\, P(x)$.

In order to do so, we define the set $Q =_{def} \{x \mid x \in I_R \wedge P(x)\}$ (thus, we get $Q \subseteq I_R$) and we show that $I_R \subseteq Q$. If we show that $I_R \subseteq Q$, we get $I_R = Q$ and, since by definition of Q, we have that $\forall x \in Q.\, P(x)$ holds, we get that $\forall x \in I_R.\, P(x)$, as desired.

Thus, it remains to show that if the premise of rule induction holds, we get $I_R \subseteq Q$.

Let us assume that premise of rule induction, that is,

for every rule instance $\dfrac{X}{y} \in \overline{R}$, if $\forall x \in X.\, x \in I_R \wedge P(x)$ then $P(y)$.

Then, (1) from $\forall x \in X.\, x \in I_R$ and (2) the fact that I_R is $\overline{R}$-closed it follows that $y \in I_R$. Thus, we have that:

for every rule instance $\dfrac{X}{y} \in \overline{R}$, if $\forall x \in X.\, x \in I_R \wedge P(x)$ then $y \in I_R \wedge P(y)$.

Then, by definition of Q, we have that:

for every rule instance $\dfrac{X}{y} \in \overline{R}$, if $\forall x \in X.\, x \in Q$ then $y \in Q$,

that is, we have that Q is an $\overline{R}$-closed set, and by Theorem 1.11 (ii) on page 67, we have $I_R \subseteq Q$, as desired. □

Now we apply rule induction for showing the equivalence of two context-free grammars.

EXAMPLE 1.14. [**Equivalence of Context-Free Grammars**] Let us consider the context-free grammar with axiom A and the following productions:

$$A \;\rightarrow\; \varepsilon \;\mid\; A\,A \;\mid\; 0\,A\,1$$

Let us also consider the context-free grammar with axiom B' and the following productions:

$$\begin{array}{lcl} B' & \rightarrow & 0\,B \\ B & \rightarrow & 1 \quad \mid \quad 0\,B\,B \end{array}$$

Let $L(A)$, $L(B)$, and $L(B')$ denote the languages generated by the nonterminals A, B, and B', respectively.

Now we show by using rule induction that the languages generated by the axioms A and B' satisfy the following equality: $L(A) = (L(B'))^*$.

In order to do so, since $L(B') = 0\,L(B)$, it is enough to show that:

(i) $L(A) \subseteq (0\,L(B))^*$ and (ii) $L(A) \supseteq (0\,L(B))^*$.

The rules for the language $L(A)$ are as follows:

$(A1)$ $\dfrac{\{\}}{\varepsilon}$ $\qquad$ $(A2)$ for any $a_1, a_2 \in L(A)$, $\dfrac{\{a_1,\ a_2\}}{a_1\,a_2}$ $\qquad$ $(A3)$ for any $a \in L(A)$, $\dfrac{\{a\}}{0\,a\,1}$

The rules for the language $L(B)$ are as follows:

$(B1)$ $\dfrac{\{\}}{1}$ $\qquad$ $(B2)$ for any $b_1, b_2 \in L(B)$, $\dfrac{\{b_1,\ b_2\}}{0\,b_1\,b_2}$

It is well known from the theory of formal languages that the inductive set of the rules $(A1)$, $(A2)$, and $(A3)$ is $L(A)$ and, analogously, the inductive set of the rules $(B1)$ and $(B2)$ is $L(B)$.

For reasons of simplicity in this example a singleton $\{x\}$ will be denoted also by x.

Proof of Point (i). It is enough to prove by rule induction that the property

$P(a) =_{def}\ a \subseteq (0\,L(B))^*$

holds for any word $a \in L(A)$. This is a consequence from the following Points (i.1), (i.2), and (i.3).

Point (i.1) By rule $(A1)$ we have to show that: $\varepsilon \subseteq (0\,L(B))^*$. This is obvious.

Point (i.2) By rule $(A2)$ we have to show that: $a_1 a_2 \subseteq (0\,L(B))^*$. Indeed, $a_1\,a_2 \subseteq$ {by hypothesis} $\subseteq (0\,L(B))^*(0\,L(B))^* = (0\,L(B))^*$.

Point (i.3) By rule $(A3)$ we have to show that: $0\,a\,1 \subseteq (0\,L(B))^*$. Indeed, $0\,a\,1 \subseteq$ {by hypothesis} $\subseteq 0\,L(A)\,1 \subseteq$ {by $B \rightarrow 1$} $\subseteq 0\,L(A)\,L(B) \subseteq$ {by inclusion (α) below} $\subseteq 0\,L(B) \subseteq (0\,L(B))^*$.

Now we prove by rule induction that:

$$L(A)\,L(B) \subseteq L(B). \qquad (\alpha)$$

We consider the property $Q(a) =_{def}\ a\,L(B) \subseteq L(B)$ and we show that it holds for all $a \in L(A)$.

By rule $(A1)$ we have to show that: $\varepsilon\,L(B) \subseteq L(B)$. This is obvious.

By rule $(A2)$ we have to show that: $a_1\,a_2\,L(B) \subseteq L(B)$. Indeed, $a_1\,a_2\,L(B) \subseteq$ {by hypothesis} $\subseteq a_1\,L(B) \subseteq$ {by hypothesis} $\subseteq L(B)$.

By rule $(A3)$ we have to show that: $0\,a\,1\,L(B) \subseteq L(B)$. Indeed, $0\,a\,1\,L(B) \subseteq$ {by $B \to 1$} $\subseteq 0\,a\,L(B)\,L(B) \subseteq$ {by hypothesis} $\subseteq 0\,L(A)\,L(B)\,L(B) \subseteq$ {by hypothesis} $\subseteq 0\,L(B)\,L(B) \subseteq$ {by $B \to 0\,B\,B$} $\subseteq L(B)$.

This completes the proof of (α).

Proof of Point (ii). It is enough to prove by rule induction that the property

$P(b) =_{def} (0\,b)^* \subseteq L(A)$

holds for any word $b \in L(B)$. This is a consequence from the following Points (ii.1) and (ii.2).

Point (ii.1) By rule $(B1)$ we have to show that: $(0\,1)^* \subseteq L(A)$. This follows from $\forall n \geq 0$, $(01)^n \subseteq L(A)$, which can be proved by induction on n.
(*Basis*) Since $(01)^0 = \varepsilon$ we have to show that $\varepsilon \subseteq L(A)$. This holds because $A \to \varepsilon$.
(*Step*) Take any $k \geq 0$. We assume $(0\,1)^k \subseteq L(A)$ and we show $(0\,1)^{k+1} \subseteq L(A)$ as follows. We have that $(0\,1)^{k+1} = (0\,1)^k\,(0\,1) \subseteq$ {by induction hypothesis} $\subseteq L(A)\,0\,1 \subseteq$ {by $A \to 0\,A\,1 \to 0\,1$} $\subseteq L(A)\,L(A) \subseteq$ {by $A \to A\,A$} $\subseteq L(A)$.

Point (ii.2) By rule $(B2)$ we have to show that: $(0\,0\,b_1\,b_2)^* \subseteq L(A)$. Indeed, $(0\,0\,b_1\,b_2)^*$ $\subseteq$ {by hypothesis} $\subseteq (0\,0\,L(B)\,L(B))^* \subseteq$ {by inclusion (β) below} $\subseteq (0\,L(A)\,L(B))^*$ $\subseteq$ {by inclusion (α) above} $\subseteq (0\,L(B))^* \subseteq$ {by inclusion (β) below} $\subseteq (L(A))^* \subseteq$ $L(A)$. This last inclusion follows from $\forall n \geq 0$, $(L(A))^n \subseteq L(A)$, which can be proved by induction on n.
(*Basis*) We have to show that $\varepsilon \subseteq L(A)$. This holds because $A \to \varepsilon$.
(*Step*) Take any $k \geq 0$. We assume $(L(A))^k \subseteq L(A)$ and we show $(L(A))^{k+1} \subseteq L(A)$ as follows. We have that $(L(A))^{k+1} = (L(A))^k\,L(A) \subseteq$ {by induction hypothesis} $\subseteq L(A)\,L(A) \subseteq$ {by $A \to A\,A$} $\subseteq L(A)$.

Now we prove by rule induction that:

$0\,L(B) \subseteq L(A)$. $\qquad (\beta)$

We consider the property $R(b) =_{def} 0\,b \subseteq L(B)$ and we show that it holds for all $b \in L(B)$.
By rule $(B1)$ we have to show that: $0\,1 \subseteq L(A)$. This is obvious because: $A \to$ {by $A \to 0A1$} $\to 0A1 \to$ {by $A \to \varepsilon$} $\to 01$.
By rule $(B2)$ we have to show that: $0\,0\,b_1\,b_2 \subseteq L(A)$. Indeed, $0\,0\,b_1\,b_2 \subseteq$ {by hypothesis} $\subseteq 0\,L(A)\,b_2 \subseteq$ {by hypothesis} $\subseteq 0\,L(A)\,L(B) \subseteq$ {by inclusion (γ) below} $\subseteq 0\,L(A)\,L(A)\,1 \subseteq$ {by $A \to 0A1 \to 0AA1$} $\subseteq L(A)$.

This completes the proof of (β).

Now we prove by rule induction that:

$L(B) \subseteq L(A)\,1$. $\qquad (\gamma)$

We consider the property $S(b) =_{def} b \subseteq L(A)\,1$ and we show that it holds for all $b \in L(B)$.
By rule $(B1)$ we have to show that: $1 \subseteq L(A)\,1$. This is obvious because $A \to \varepsilon$.
By rule $(B2)$ we have to show that: $0\,b_1\,b_2 \subseteq L(A)\,1$. Indeed, $0\,b_1\,b_2 \subseteq$ {by hypothesis} $\subseteq 0\,L(A)\,1\,b_2 \subseteq$ {by $A \to 0A1$} $\subseteq L(A)\,b_2 \subseteq$ {by hypothesis} $\subseteq L(A)\,L(A)\,1 \subseteq$ {by $A \to AA$} $\subseteq L(A)\,1$.

This completes the proof of (γ) and the proof of $L(A) = (0\,L(B))^*$. $\square$

1.5. Special Rule Induction.

There is a particular version of rule induction, called *special rule induction*, which can be used when dealing with properties referring to more than one set.

Let us first see this induction rule in action in the following example that deals with basic boolean expressions and extends the example of Section 1.4 (see page 64) dealing with the set **BasicAexp** of the basic arithmetic expressions.

The set **BasicBexp** of the *basic boolean expressions* is defined as follows, where $a_1, a_2 \in \mathbf{BasicAexp}$ and $b, b_1, b_2 \in \mathbf{BasicBexp}$:

$$b ::= \mathbf{true} \mid \mathbf{false} \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2$$

The operational semantics of the basic boolean expressions is specified by a subset of $\mathbf{BasicBexp} \times \{\mathbf{true}, \mathbf{false}\}$, and a pair $\langle b, t \rangle$ in $\mathbf{BasicBexp} \times \{\mathbf{true}, \mathbf{false}\}$, written $b \to t$, denotes that the basic boolean expression b evaluates to t.

The rules defining the operational semantics of the basic boolean expressions are as follows: for all $b, b_1, b_2 \in \mathbf{BasicBexp}$, for all $t_1, t_2 \in \{\mathbf{true}, \mathbf{false}\}$, for all $a_1, a_2 \in \mathbf{BasicAexp}$, we have that:

(B1) $$\frac{}{\mathbf{true} \to \mathbf{true}}$$

(B2) $$\frac{}{\mathbf{false} \to \mathbf{false}}$$

(B3.1) $$\frac{a_1 \to n_1 \qquad a_2 \to n_2}{a_1 \leq a_2 \to \mathbf{true}} \text{ if } n_1 \leq n_2$$

(B3.2) $$\frac{a_1 \to n_1 \qquad a_2 \to n_2}{a_1 \leq a_2 \to \mathbf{false}} \text{ if } n_1 \not\leq n_2$$

(B4.1) $$\frac{b \to \mathbf{true}}{\neg\, b \to \mathbf{false}}$$

(B4.2) $$\frac{b \to \mathbf{false}}{\neg\, b \to \mathbf{true}}$$

(B5) $$\frac{b_1 \to t_1 \qquad b_2 \to t_2}{b_1 \wedge b_2 \to and(t_1,t_2)}$$ where $and(t_1, t_2)$ performs the usual *and* operation between any two boolean values t_1 and t_2.

together with the rules (A1) and (A2) for the evaluation of the basic arithmetic expressions (see page 64).

Now we can show that a given property $P(b, t)$ holds for every pair $\langle b, t \rangle$ in the subset of $\mathbf{BasicBexp} \times \{\mathbf{true}, \mathbf{false}\}$ derived by the rules (B1)–(B5), by applying the following rule.

(Special Rule Induction for **BasicBexp**)

$$\begin{array}{l}
\forall n_1, n_2 \in N.\, \forall a_1, a_2 \in \mathbf{BasicAexp}. \\
\forall t_1, t_2 \in \{\mathbf{true}, \mathbf{false}\}.\, \forall b, b_1, b_2 \in \mathbf{BasicBexp}. \\
\quad P(\mathbf{true}, \mathbf{true}) \\
\wedge\ P(\mathbf{false}, \mathbf{false}) \\
\wedge\ (a_1 \rightarrow n_1 \wedge a_2 \rightarrow n_2 \wedge n_1 \leq n_2) \Rightarrow P(a_1 \leq a_2, \mathbf{true}) \\
\wedge\ (a_1 \rightarrow n_1 \wedge a_2 \rightarrow n_2 \wedge n_1 \not\leq n_2) \Rightarrow P(a_1 \leq a_2, \mathbf{false}) \\
\wedge\ (b \rightarrow \mathbf{true} \wedge P(b, \mathbf{true})) \Rightarrow P(\neg b, \mathbf{false}) \\
\wedge\ (b \rightarrow \mathbf{false} \wedge P(b, \mathbf{false})) \Rightarrow P(\neg b, \mathbf{true}) \\
\wedge\ (b_1 \rightarrow t_1 \wedge P(b_1, t_1) \wedge b_2 \rightarrow t_2 \wedge P(b_2, t_2)) \Rightarrow P(b_1 \wedge b_2, and(t_1, t_2)) \\
\hline
\forall t \in \{\mathbf{true}, \mathbf{false}\}.\, \forall b \in \mathbf{BasicBexp}.\ P(b, t)
\end{array}$$

Note that in the special rule induction for **BasicBexp** we have assumed that the property $P(b,t)$ holds for the elements $\langle b, t\rangle$ of the set $\mathbf{BasicBexp} \times \{\mathbf{true}, \mathbf{false}\}$ which occur in the premises of the rules, while it does *not* hold for the elements in the set $\mathbf{BasicAexp} \times N$ (and in that sense this rule induction is said to be 'special').

Here is the special rule induction in the general case. Let us consider a set $A \subseteq I_R$. Let $P(x)$ be a property. Then

$$\begin{array}{l}
\forall x \in A.\, P(x) \quad \textit{iff} \\
\text{for all rule instances } \dfrac{X}{y} \in \overline{R} \text{ such that } X \subseteq I_R \text{ and } y \in A, \text{ we have that} \qquad (\dagger\dagger) \\
\quad (\forall x \in X \cap A.\, P(x)) \Rightarrow P(y).
\end{array}$$

Thus, by taking the *if*-part of the above formula ($\dagger\dagger$), we have the following rule.

(Special Rule Induction)

$$\frac{\text{for every rule instance } \dfrac{X}{y} \in \overline{R},\ \ (\forall x \in X \cap A.\ \ x \in I_R \wedge y \in A \wedge P(x)) \Rightarrow P(y)}{\forall x \in A.\, P(x)}$$

In the case of the evaluation of basic boolean expressions, we have that:

$$\begin{array}{lcl}
I_R & = & \mathbf{BasicAexp} \times N \ \cup\ \mathbf{BasicBexp} \times \{\mathbf{true}, \mathbf{false}\} \quad \text{and} \\
A & = & \mathbf{BasicBexp} \times \{\mathbf{true}, \mathbf{false}\}.
\end{array}$$

We leave it to the reader to show that by using the rule induction for the property $x \in A \Rightarrow P(x)$, we have the special rule induction for the property $P(x)$ and the set A. Thus, special rule induction is *not* more powerful than rule induction.

1.6. Well-founded Induction.

Let us now consider one more induction rule, called *well-founded induction*. It is a very general induction rule from which we can derive other induction rules. We start off by introducing the following definition.

DEFINITION 1.15. [**Well-founded Order**] A binary relation $\prec$ on a set B is a subset of $B \times B$ and it said to be *well-founded* or a *well-founded order* if there is no infinite descending sequence of the form: $\ldots \prec b_2 \prec b_1 \prec b_0$.

Any well-founded relation is irreflexive. As usual, by $\prec^+$ we denote the transitive closure of $\prec$, and by $\prec^*$ we denote the reflexive, transitive closure of $\prec$. We have the following theorem.

THEOREM 1.16. Given a set B and a well-founded binary relation $\prec \subseteq B \times B$, we have that $\prec$ is well-founded iff $\prec^+$ is well-founded.

PROOF. (i) If $\prec$ is not well-founded also $\prec^+$ is not well-founded because an infinite descending sequence $\ldots \prec b_2 \prec b_1 \prec b_0$ is also an infinite descending sequence of the form $\ldots \prec^+ b_2 \prec^+ b_1 \prec^+ b_0$. (ii) If $\prec^+$ is not well-founded then there exists an infinite descending sequence of the form $\ldots \prec^+ b_2 \prec^+ b_1 \prec^+ b_0$. Now, by definition of $\prec^+$, $\forall b_{i+1}, b_i \in B$, $b_{i+1} \prec^+ b_i$ means that there exists a finite sequence $b^1 \prec \ldots \prec b^{k_i}$ of elements in B such that $b^1 = b_{i+1}$ and $b^{k_i} = b_i$. Thus, from the infinite sequence $\ldots \prec^+ b_2 \prec^+ b_1 \prec^+ b_0$ we can get an infinite descending sequence for $\prec$ which shows that $\prec$ is not well-founded. □

The rule of well-founded induction for a set B whose elements are ordered by a well-founded binary relation $\prec$, is as follows.

(Well-founded Induction. W1)

$$\frac{\forall x \in B.\, (\forall y \prec x.\, P(y)) \Rightarrow P(x)}{\forall b \in B.\, P(b)}$$

From this rule we can derive an equivalent well-founded induction rule, called W2. First we need the following definition.

DEFINITION 1.17. [**Minimal Element**] Let A be a set with an irreflexive binary relation $\ll \subseteq A \times A$. We say that m is a *minimal element* of A w.r.t. $\ll$ iff we have that $\neg\, \exists\, a \in A.\, a \ll m$.

The well-founded induction rule W2 is based on the absence of a minimal element. For any set B whose elements are ordered by a well-founded binary relation $\prec \subseteq B \times B$, the well-founded induction rule W2 is as follows.

(Well-founded Induction. W2)

$$\frac{\begin{array}{l}\overline{B} = \{b \mid b \in B \wedge \neg P(b)\} \\ \text{and no minimal element w.r.t. } \prec \text{ exists in } \overline{B}\end{array}}{\forall b \in B.\, P(b)}$$

This rule can be derived from the rule W1 of well-founded induction because from the premises of W2, by recalling that $\prec$ is well-founded, we get that $\overline{B} = \emptyset$, and thus, $\forall b \in B.\, P(b)$ holds.

From the well-founded induction rule W1 we can derive other induction rules. In particular, we can derive:
(i) mathematical induction (see Section 1.1 on page 59) by taking

$\forall n, m \in N.\, n \prec m$ iff $s(n) = m$,

(ii) complete induction (see Section 1.2 on page 61) by taking

$\forall n, m \in N.\, n \prec m$ iff $n < m$, and

(iii) structural induction for the binary trees (see Section 1.3 on page 63) by taking

$\forall b_1, b_2 \in B, \forall n \in N.\ b_1 \prec \langle b_1, n, b_2\rangle \wedge b_2 \prec \langle b_1, n, b_2\rangle$.

Some more induction rules which are particularly useful for proving properties of programs, will be presented in Section 6 on page 103.

2. Recursion Theorem

Let us introduce the following notations.

(i) Given a binary relation $\rho \subseteq B \times B$, for all $b \in B$, we denote by $\rho^{-1}\{b\}$ the set $\{x \mid x\, \rho\, b\}$. In particular, if we consider a well-founded binary relation $\prec \subseteq B \times B$, for all $b \in B$, we denote by $\prec^{-1}\{b\}$ the set $\{x \mid x \prec b\}$.

(ii) Given a function $f : B \to C$ and a subset A of B, *the restriction of the function f to the set A*, denoted $f \restriction A$, is the function from A to C which is the following set of pairs $\{\langle x, f(x)\rangle \mid x \in A\}$.

(iii) The *domain* of a relation (or a function) $\rho \subseteq B \times C$, denoted $dom(\rho)$, is the set $\{b \mid \exists c\, \langle b, c\rangle \in \rho\}$.

(iv) By $\mathcal{P}(A)$ we denote the powerset of a set A, that is, the set of all subsets of A.

We have the following theorem.

THEOREM 2.1. [**Recursion Theorem**] Let $\prec$ be a well-founded relation on a set B. Suppose that for all $b \in B$, for all functions $h_b : \{x \mid x \prec b\} \to C$, there exists a function $F : (B \times (B_b \to C)) \to C$, where B_b denotes the subset $\{x \mid x \prec b\}$ of B that depends on the value $b \in B$ of the first argument of F.

Then, there exists a *unique function* f: $B \to C$ such that:

$\forall b \in B,\ f(b) = F(b, f \restriction \prec^{-1}\{b\})$. (*R*0)

PROOF. The proof is made out of two parts. In Part 1 we show that there exists *at most* one function satisfying (*R*0) and in Part 2 we define a function that satisfies (*R*0).

Part 1. Let $R(f, g, y)$ be the formula:

$f(y) = F(y, f \restriction \prec^{-1}\{y\}) \wedge g(y) = F(y, g \restriction \prec^{-1}\{y\})$.

Let $P(z)$ be the formula: $\big((\forall y \prec^* z,\ R(f, g, y)\big) \to f(z) = g(z)$. Now we prove that $\forall b \in B,\ P(b)$, that is:

$\forall b \in B,\ \big(\big(\forall y \prec^* b,\ R(f, g, y)\big) \to f(b) = g(b)\big)$ (*R*1)

by well-founded induction on $\prec$. Thus, (i) we take any $x \in B$, (ii) we assume that $\forall z \prec x,\ P(z)$, and (iii) we have to show $P(x)$. By assumption we have:

$\forall z \prec x,\ P(z)$, and (1.1)

$\forall y \prec^* x,\ R(f,g,y)$, (1.2)

and we have to show that $f(x)=g(x)$.

Take any $z \prec x$. Since $z \prec x$ and $\prec^*$ is reflexive and transitive, from (1.2) we get:

$\forall y \prec^* z,\ R(f,g,y)$. (1.3)

From (1.1) and (1.3) we get: $f(z)=g(z)$. Since we have taken any $z \prec x$ and we have derived $f(z)=g(z)$, we have that:

$\forall z \prec x,\ f(z)=g(z)$. (1.4)

By the definition of function restriction, since $\prec^{-1}\{x\}$ is the set of elements z such that $z \prec x$, from (1.4) we get that:

$f \restriction \prec^{-1}\{x\} = g \restriction \prec^{-1}\{x\}$.

Since F is a function we get: $F(x, f \restriction \prec^{-1}\{x\}) = F(x, g \restriction \prec^{-1}\{x\})$. (1.5)

From (1.2), for $y=x$ we get:

$f(x)=F(x, f \restriction \prec^{-1}\{x\}) \wedge g(x)=F(x, g \restriction \prec^{-1}\{x\})$. (1.6)

From (1.5) and (1.6) we get: $f(x) = g(x)$ and this concludes the proof of $(R1)$.

From $(R1)$ it follows, as we now show, that if there are two functions, say f and g, that satisfy $(R0)$ then they are equal, that is,

$\big(\big(\forall b \in B,\ f(b)=F(b, f \restriction \prec^{-1}\{b\}\big) \ \wedge \ \big(\forall b \in B,\ g(b)=F(b, g \restriction \prec^{-1}\{b\})\big)\big)$
$\rightarrow \forall b \in B,\ f(b)=g(b)$,

which is equivalent to:

$\forall b \in B,\ \big(f(b)=F(b, f \restriction \prec^{-1}\{b\} \ \wedge \ g(b)=F(b, g \restriction \prec^{-1}\{b\})\big)$ $(R2)$
$\rightarrow \forall b \in B,\ f(b)=g(b)$.

Indeed, let us assume the premise of $(R2)$ and let us show the conclusion of $(R2)$. Take any $b \in B$, by the premise of $(R2)$ we have that $\forall y \prec^* b$, $R(f,g,y)$ holds. By $(R1)$ we get that $f(b)=g(b)$ holds, and this completes the proof of Part 1.

Part 2. In this part we prove the following three Properties (2.1), (2.2), and (2.3).

(*Property* 2.1) For all $x \in B$ there exists a total function $f_x : \{y \mid y \prec^* x\} \rightarrow C$ which is defined as follows:

$\forall y \prec^* x,\ f_x(y) =_{def} F(y, f_x \restriction \prec^{-1} \{y\})$, (2.1.1)

(*Property* 2.2) Given any two functions f_{b1} and f_{b2} in the set $\{f_x \mid x \in B\}$ of functions (each of which is defined as in Property 2.1), we have that:

$\forall b1, b2 \in B,\ \forall y \prec^* b1,\ \forall y \prec^* b2,\ f_{b1}(y) = f_{b2}(y)$, and

(*Property* 2.3) Given the set $\{f_x \mid x \in B\}$ of functions (each of which is defined as in Property 2.1), we have that:

$dom(\bigcup_{x \in B} f_x) = B$ and $\forall b \in B,\ (\bigcup_{x \in B} f_x)(b) = F(b, (\bigcup_{x \in B} f_x) \restriction \prec^{-1} \{b\})$.

From Properties 2.1, 2.2, and 2.3, it follows that $\bigcup_{x \in B} f_x$ is a function which satisfies Property $(R0)$ on the preceding page and whose domain is B. Thus, in order to complete the proof of the theorem, we have to prove the three Properties 2.1, 2.2, and 2.3, and this will be done in the following three Points 2.1, 2.2, and 2.3, respectively.

Point 2.1. Let $Q(x)$ be the formula:

there exists a total function $f_x : \{y \mid y \prec^* x\} \to C$ such that

$\forall y \prec^* x,\ f_x(y) = F(y, f_x \restriction \prec^{-1} \{y\})$.

Now we prove that $\forall x \in B,\ Q(x)$ holds by well-founded induction on $\prec$. We do so by: (i) taking any $x \in B$, (ii) assuming $\forall z \prec x, Q(z)$, and (iii) showing $Q(x)$.

From the assumption that $\forall z \prec x, Q(z)$ we have that

$$\forall z \prec x,\ dom(f_z) = \{y \mid y \prec^* z\}. \tag{2.1.2}$$

Let us consider the set h_x defined as follows:

$$h_x =_{def} \bigcup_{z \prec x} f_z. \tag{2.1.3}$$

We show that:

(2.1.i) $dom(h_x) = \{y \mid y \prec^+ x\}$ (see Point 2.1.a), and

(2.1.ii) h_x is a function from $\{y \mid y \prec^+ x\}$ to C, that is, for all $y \in dom(h_x)$ there exists a unique $v \in C$ such that $\langle y, v\rangle \in h_x$. This means that any two functions f_{b1} and f_{b2} such that $b1 \prec x$ and $b2 \prec x$, agree on the intersection of their domains (see Point 2.1.b).

Point 2.1.a. By (2.1.3) we have that $dom(h_x) = \bigcup_{z \prec x} dom(f_z)$. By (2.1.2) we have that:

$\bigcup_{z \prec x} dom(f_z) = \bigcup_{z \prec x} \{y \mid y \prec^* z\} = \{y \mid y \prec^+ x\}$.

Point 2.1.b. Take any $b1 \prec x$, any $b2 \prec x$, any $y \in dom(f_{b1}) \cap dom(f_{b2})$ (i.e., $y \prec^* b1$ and $y \prec^* b2$). By the induction hypothesis which states that $\forall z \prec x, Q(z)$ holds, we have that f_{b1} is a function and

$\forall u \prec^* b1,\ f_{b1}(u) = F(u, f_{b1} \restriction \prec^{-1} \{u\})$.

Analogously, by the induction hypothesis, we have that f_{b2} is a function and

$\forall u \prec^* b2,\ f_{b2}(u) = F(u, f_{b2} \restriction \prec^{-1} \{u\})$.

Thus, we get:

$\forall u \prec^* y,\ [f_{b1}(u) = F(u, f_{b1} \restriction \prec^{-1} \{u\})] \wedge \forall u \prec^* y,\ \big(f_{b2}(u) = F(u, f_{b2} \restriction \prec^{-1} \{u\})\big)$.

Now, let us consider the set $\{u \mid u \prec^* y\}$, the well-founded relation $\prec$ on that set, and the two functions f_{b1} and f_{b2} restricted to that set. By using the uniqueness result of Part 1 in the case of the functions f_{b1} and f_{b2} restricted to the set $\{u \mid u \prec^* y\}$, we have that for all $u \in \{u \mid u \prec^* y\}$, $f_{b1}(u) = f_{b2}(u)$ and, in particular, $f_{b1}(y) = f_{b2}(y)$.

This concludes the proof of Point 2.1.b.

Now let us define f_x to be the set of pairs $h_x \cup \{\langle x, F(x, h_x \restriction \prec^{-1} \{x\})\rangle\}$.

Since h_x is a function and $x \notin dom(h_x)$, we have that f_x is a function. We also have that $dom(f_x) = dom(h_x) \cup \{x\}$, that is, $dom(f_x) = \{y \mid y \prec^* x\}$ because $h_x = \{y \mid y \prec^+ x\}$.

Now in order to show $Q(x)$, we need to show that $f_x(y) = F(y, f_x \restriction \prec^{-1} \{y\})$ holds for all y such that $y \prec^* x$. We do so by cases and, in particular, we show that $f_x(y) = F(y, f_x \restriction \prec^{-1} \{y\})$ holds:

(2.1.c) for $y = x$ (see Point 2.1.c), and

(2.1.d) for any y such that $y \prec^+ x$ (see Point 2.1.d).

Point 2.1.c. We have to show that:

$f_x(x) = F(x, f_x \restriction \prec^{-1} \{x\})$. Since, by definition, $f_x(x) =_{def} F(x, h_x \restriction \prec^{-1} \{x\})$ and F is a function, it is enough to show that:

$$h_x \restriction \prec^{-1} \{x\} = f_x \restriction \prec^{-1} \{x\}.$$

Indeed, we have that: $h_x \restriction \prec^{-1} \{x\} =$

$= \{$adding to h_x a value for x does not modify the restriction to $\prec^{-1} \{x\}\} =$

$= h_x \cup \{\langle x, F(x, h_x \restriction \prec^{-1} \{x\})\rangle\}) \restriction \prec^{-1} \{x\} = \{$by definition of $f_x\} =$

$= f_x \restriction \prec^{-1} \{x\}$.

Point 2.1.d. We have to show that:

$\forall y \prec^+ x$, $f_x(y)\,(=_{def} h_x(y)) = F(y, f_x \restriction \prec^{-1} \{y\})$. Since, by definition, for all y such that $y \prec^+ x$, we have that $f_x(y) =_{def} h_x(y)$, it is enough to show that $h_x(y) = F(x, h_x \restriction \prec^{-1} \{x\})$. In order to do so, let us take any y such that $y \prec^+ x$. We have:

$h_x(y) = \{$by the defining Equation 2.1.3$\} =$

$= (\bigcup_{z \prec x} f_z)(y) = \{$by Point 2.1.b$\} =$

$= f_{\overline{z}}(y)$, for all $\overline{z}$ such that $y \prec^* \overline{z} \prec x$. (2.1.4)

Note that: (i) $f_{\overline{z}}$ is a function because, by induction hypothesis, $\forall z \prec x, Q(z)$ holds, and (ii) every y such that $y \prec^* \overline{z} \prec x$ belongs to $dom(f_{\overline{z}})$, because, by induction hypothesis, $dom(f_{\overline{z}}) = \{y \mid y \prec^* \overline{z}\}$.

Now, for all $y, \overline{z}$, such that $y \prec^* \overline{z} \prec x$ we have:

$f_{\overline{z}}(y) = \{$by the induction hypothesis $\forall z \prec x, Q(z)\} =$

$= F(y, f_{\overline{z}} \restriction \prec^{-1} \{y\}) =$

$= \{$by $\forall y \prec^+ x.\, h_x(y) = f_{\overline{z}}(y)$ (see Equation 2.1.4) and the fact that

$\forall y \prec^+ x.\, h_x(y) = f_{\overline{z}}(y)$ implies $\forall w \prec y.\, f_{\overline{z}}(w) = h_x(w)\} =$

$= F(y, h_x \restriction \prec^{-1} \{y\}) = \{h_x$ and f_x differ in x only, and $y \neq x\} =$

$= F(y, f_x \restriction \prec^{-1} \{y\})$.

This complete the proof of Point 2.1.d and also the proof of Point 2.1.

Point 2.2. The proof of this point is equal to the proof of Point 2.1.b.

Take any $b1 \in B$, any $b2 \in B$, any $y \in dom(f_{b1}) \cap dom(f_{b2})$ (i.e., $y \prec^* b1$ and $y \prec^* b2$). By induction hypothesis we have that $\forall z \prec x, Q(z)$ holds, and thus:

$$\forall u \prec^* y,\, [f_{b1}(u) = F(u, f_{b1} \restriction \prec^{-1} \{u\})] \wedge \forall u \prec^* y,\, [f_{b2}(u) = F(u, f_{b2} \restriction \prec^{-1} \{u\})].$$

Now, let us consider the set $\{u \mid u \prec^* y\}$, the well-founded relation $\prec$ on that set, and the two functions f_{b1} and f_{b2} restricted to that set. By using the uniqueness result of Part 1 in the case of the functions f_{b1} and f_{b2} restricted to the set $\{u \mid u \prec^* y\}$, we have that for all $u \in \{u \mid u \prec^* y\}$, $f_{b1}(u) = f_{b2}(u)$ and, in particular, $f_{b1}(y) = f_{b2}(y)$.

Point 2.3. We have that $dom(\bigcup_{x \in B} f_x) \supseteq B$ because for any $x \in B$, $x \in dom(f_x)$. We also have that $dom(\bigcup_{x \in B} f_x) = \bigcup_{x \in B} dom(f_x) = \bigcup_{x \in B} \{y \mid y \prec^* x\} \subseteq B$ because $\prec \subseteq B \times B$.

Thus, $dom(\bigcup_{x \in B} f_x) = B$.

Now we show that $\forall b \in B$, $(\bigcup_{x \in B} f_x)(b) = F(b, (\bigcup_{x \in B} f_x) \restriction \prec^{-1} \{b\})$. We proceed as follows. Take any $b \in B$. We have that:

$(\bigcup_{x\in B} f_x)(b) = \{$by the fact that $b \in dom(f_b)\} =$
$= f_b(b) = \{$by Equation 2.1.1$\} =$
$= F(b,\ f_b \restriction \prec^{-1}\{b\}) = \{$by $\{y \mid y \prec^{-1} b\} \subseteq dom(f_b)$ and by Point 2.2$\} =$
$= F(b,\ (\bigcup_{x\in B} f_x) \restriction \prec^{-1}\{b\})$.
This completes the proof of Point 2.3 and the proof of the theorem. □

Now, let us consider a particular instance of the Recursion Theorem by taking:
(i) the set B to be the set $N = \{0, s(0), s(s(0)), \ldots\}$ of the natural numbers,
(ii) the well-founded relation $\prec$ on B to be the familiar $<$ (*less-than*) relation on N, and
(iii) the function $F : N \times \mathcal{P}(N \times N) \to N$ to be the following function (note that the set $\{k \in N \mid k < 0\}$ is empty and the set $\{k \in N \mid k < s(n)\}$ is the singleton $\{n\}$):

$$\begin{array}{ll} F(0, \{\}) & = a \\ F(s(n), \{\langle n, m\rangle\}) & = G(n, m) \end{array}$$

where $a \in N$ and G is a function from $N \times N$ to N. (The fact that F is a function derives from the fact that G is a function and we do not need to apply the Recursion Theorem because F is *not* recursively defined.)

Having made the above choices, the Recursion Theorem tells us that there exists a unique function $f : N \to N$ such that:

$$\begin{array}{ll} f(0) & = a \\ f(s(n)) & = G(n, f(n)) \end{array}$$

Moreover, if we take $a = 1$ and $\lambda n, m.\, G(n, m) =_{def} \lambda n, m.\, s(n) \cdot m$, we have that the equations:

$$\begin{array}{ll} f(0) & = 1 \\ f(s(n)) & = s(n) \cdot f(n) \end{array}$$

determine a unique function from N to N (it is the familiar factorial function).

Similarly, by taking:
(i) B to be N,
(ii) the well-founded relation $\prec$ to be the following one:

$\{n \prec m \mid m > s(0)$ *and* $[m = s(n)$ *or* $m = s(s(n))]\} \subseteq N \times N$, and

(iii) a suitable choice of the function F, by the Recursion Theorem we have that for any natural number n_0 and n_1, the equations:

$$\begin{array}{ll} \mathit{fib}(0) & = n_0 \\ \mathit{fib}(s(0)) & = n_1 \\ \mathit{fib}(s(s(n))) & = \mathit{fib}(s(n)) + \mathit{fib}(n) \end{array}$$

determine a unique function from N to N (it is the familiar Fibonacci function).

The Recursion Theorem provides a justification for the definition of functions via the following schema, called *primitive recursion schema*:

$$\begin{array}{lll} f(0, x_2, \ldots, x_k) & = g(x_2, \ldots, x_k) & \text{(PR1)} \\ f(n{+}1, x_2, \ldots, x_k) & = h(n, x_2, \ldots, x_k, f(n, x_2, \ldots, x_k)) & \text{(PR2)} \end{array}$$

where g and h are functions from N^{k-1} to N and from N^{k+1} to N, respectively. In these hypotheses by the Recursion Theorem we have that there exists a *unique* function f from N^k to N satisfying (PR1) and (PR2).

A different setting for defining functions by using equations is that of the monotonic (or continuous) functionals and in that setting functions are defined as fixpoints of those functionals. It may be the case that for those functionals we *cannot* find any well-founded relation among the arguments of the function calls on the left hand side and the right hand side of the equations and, thus, the Recursion Theorem cannot be applied. For instance, in the case of the following two equations:

(i) $f(x) \;=\; f(x)$
(ii) $f(x) \;=\; f(x)+1$

which correspond to the following two functionals, respectively:

(i) $\lambda f.(\lambda x.\, f(x)) \quad : (N_\bot \to N_\bot) \times N_\bot \to N_\bot$
(ii) $\lambda f.(\lambda x.\, f(x)+1) : (N_\bot \to N_\bot) \times N_\bot \to N_\bot$

one cannot find any well-founded relation between x (that is, the argument of $f(x)$ on the left hand side) and x itself (that is, the argument of $f(x)$ on the right hand side) because any well-founded relation must be irreflexive.

The monotonicity (or the continuity) of a functional allows us to establish that the function defined via a functional is the *unique minimal fixpoint* of that functional (see the following Section 3). Indeed, by the Knaster-Tarski Theorem, any monotonic operator on a complete lattice has a unique minimal fixpoint, also called the *least fixpoint*. If this functional is also continuous then the minimal fixpoint can be computed, as stated by Kleene Theorem 4.7 on page 96, by performing ω iterations starting from the bottom of the lattice. In the case of the lattice of the above functionals (i) $\lambda f, x.\, f(x)$ and (ii) $\lambda f, x.\, f(x)+1$, the bottom element is the everywhere undefined function $\lambda x \in N_\bot .\, \bot$.

3. Knaster-Tarski Theorem on Complete Lattices

A *lattice* L is a set, which we denote by the same letter L, together with: (i) a partial order $\leq \subseteq L \times L$, and (ii) two binary operations, denoted by *glb* and *lub*, called the *greatest lower bound* (w.r.t. $\leq$) and the *least upper bound* (w.r.t. $\leq$), respectively.

In this case we say that the lattice L is ordered by $\leq$.

By definition, we have that for any x, y, and z in L,
(i.1) $glb(x, y) \leq x$ and $glb(x, y) \leq y$, that is, $glb(x, y)$ is a *lower bound* of x and y, and
(ii.1) if $z \leq x$ and $z \leq y$ then $z \leq glb(x, y)$, that is, $glb(x, y)$ is the greatest among the lower bounds of x and y.

Analogously, by definition, we have that for any x, y, and z in L,
(i.2) $x \leq lub(x, y)$ and $y \leq lub(x, y)$, that is, the $lub(x, y)$ is a *upper bound* of x and y, and
(ii.2) if $x \leq z$ and $y \leq z$ then $lub(x, y) \leq z$, that is, $lub(x, y)$ is the least among the upper bounds of x and y.

Conditions (ii.1) and (ii.2) above imply that given any two elements x and y in L, $glb(x, y)$ and $lub(x, y)$ are unique.

A lattice L is said to be *complete* iff the *glb* and the *lub* operations are defined for every (finite or infinite) subset S of L. The greatest lower bound of a set $S \subseteq L$ will be denoted by $glb(S)$ or $glb\, S$. Analogously, the least upper bound of a set $S \subseteq L$ will be denoted by $lub(S)$ or $lub\, S$.

The greatest lower bound of a complete lattice L is denoted by $\bot$. Thus, $\bot$ is the least element of L, that is, for all $x \in L$, $\bot \leq x$.

Let L be a complete lattice ordered by $\leq$ and T be a function from L to L. We say that x is a *prefixpoint* of T iff $T(x) \leq x$. We say that x is a *postfixpoint* of T iff $x \leq T(x)$. We also say that x is a *fixpoint* of T iff $T(x) = x$.

Let us define:

$$\begin{array}{lcll} T^0(x) & = & x & \\ T^{k+1}(x) & = & T(T^k(x)) & \text{for any } k \geq 0 \\ T^\omega(x) & = & lub\,\{T^k(x) \mid k \geq 0\} & \end{array}$$

The function $T\colon L \to L$ is said to be *monotonic on* L (or *monotonic*, for short) iff for every x and y if $x \leq y$ then $T(x) \leq T(y)$.
The function $T\colon L \to L$ is said to be *continuous on* L (or *continuous*, for short) iff it is monotonic on L and for every infinite ω-chain $x_0 \leq x_1 \leq \ldots \leq x_n \leq \ldots$ of elements (not necessarily distinct) of L we have that: $T(lub\{x_i \mid i \geq 0\}) = lub\{T(x_i) \mid i \geq 0\}$.

LEMMA 3.1. Let $T\colon L \to L$ be a monotonic function on a complete lattice L ordered by $\leq$. We have that:
(A) $glb\{x \mid T(x) \leq x\}$, that is, the *glb* of all prefixpoints of T, is the least prefixpoint of T, and
(B) $lub\{x \mid x \leq T(x)\}$, that is, the *lub* of all postfixpoints of T, is the greatest postfixpoint of T.

PROOF. Let us first show (A). We have to show that: (1) $glb\{x \mid T(x) \leq x\}$ is a prefixpoint of T, that is, $T(glb\{x \mid T(x) \leq x\}) \leq glb\{x \mid T(x) \leq x\}$, and (2) given any other prefixpoint z of T we have that $glb\{x \mid T(x) \leq x\} \leq z$.
Proof of (1). For every y such that $T(y) \leq y$, we have that:

$$\begin{array}{rl} glb\{x \mid T(x) \leq x\} \leq y & \text{(by definition of } glb\text{, because } y \in \{x \mid T(x) \leq x\}) \\ T(glb\{x \mid T(x) \leq x\}) \leq T(y) & \text{(by monotonicity of } T) \\ T(glb\{x \mid T(x) \leq x\}) \leq y & \text{(by transitivity and } T(y) \leq y). \qquad (\dagger) \end{array}$$

Now, since Inequality (†) holds for every y such that $T(y) \leq y$, we have that $T(glb\{x \mid T(x) \leq x\})$ is a lower bound of the set $\{x \mid T(x) \leq x\}$. Since, by definition, $glb\{x \mid T(x) \leq x\}$ is the greatest among the lower bounds of the set $\{x \mid T(x) \leq x\}$, we get (1).
Proof of (2). Given a complete lattice L, for every subset S of L, we have that $glb(S) \leq x$ for every element x in S. The thesis follows from: (i) $glb(S) \leq x$ by taking S to be $\{x \mid T(x) \leq x\}$, and (ii) the fact that z is a prefixpoint of T, that is, z is an element of $\{x \mid T(x) \leq x\}$.

The proof of (B) follows from that of (A) because the complete lattice L ordered by the partial order $\leq$ is also a complete lattice ordered by the converse relation $\geq$, defined as follows: for all x and y in L, $x \geq y$ iff $y \leq x$. Note that also $\geq$ is a partial order. □

THEOREM 3.2. **[Knaster-Tarski Theorem]** (1955) Let $T : L \to L$ be a monotonic function on a complete lattice L ordered by $\leq$.
(A) T has a least fixpoint, denoted by $lfp(T)$, and
$lfp(T) = glb\{x \mid T(x) = x\} = glb\{x \mid T(x) \leq x\}$.
(B) T has a greatest fixpoint, denoted by $gfp(T)$, and
$gfp(T) = lub\{x \mid T(x) = x\} = lub\{x \mid x \leq T(x)\}$.

PROOF. Let us first show Part (A). We start off by showing that:
(1) $T(glb\{x \mid T(x) \leq x\}) = glb\{x \mid T(x) \leq x\}$,
that is, $glb\{x \mid T(x) \leq x\}$ is a fixpoint of T. This can be shown by proving that:
(1.1) $T(glb\{x \mid T(x) \leq x\}) \leq glb\{x \mid T(x) \leq x\}$, and
(1.2) $T(glb\{x \mid T(x) \leq x\}) \geq glb\{x \mid T(x) \leq x\}$.
The proof of (1.1) is Point (1) of proof of Lemma 3.1. The proof of (1.2) is as follows. From (1.1) by monotonicity of T we get:
$T(T(glb\{x \mid T(x) \leq x\})) \leq T(glb\{x \mid T(x) \leq x\})$.
Thus, $T(glb\{x \mid T(x) \leq x\})$ is a prefixpoint of T and hence, it belongs to the set $\{x \mid T(x) \leq x\}$. Thus, $glb\{x \mid T(x) \leq x\} \leq T(glb\{x \mid T(x) \leq x\})$, as stated in (1.2). We also have:
(2) $glb\{x \mid T(x) = x\} \leq glb\{x \mid T(x) \leq x\}$
because $glb\{x \mid T(x) = x\} \leq x$ for every x which is a fixpoint of T, and
$glb\{x \mid T(x) \leq x\}$ is a fixpoint of T, as we have shown in (1).
We also have that:
(3) $glb\{x \mid T(x) \leq x\} \leq glb\{x \mid T(x) = x\}$
because $\{x \mid T(x) = x\} \subseteq \{x \mid T(x) \leq x\}$.
From (2) and (3) we get: $glb\{x \mid T(x) \leq x\} = glb\{x \mid T(x) = x\}$. Now, since $glb\{x \mid T(x) \leq x\}$ is a fixpoint of T, as we have shown in (1) above, we have that:
(4) $glb\{x \mid T(x) = x\}$ is a fixpoint of T.
To complete the proof of Part (A) of this theorem we have to show that:
(5) $glb\{x \mid T(x) = x\}$ is the least fixpoint of T.
Since in (4) we have proved that $glb\{x \mid T(x) = x\}$ is a fixpoint of T, it remains to show that $glb\{x \mid T(x) = x\}$ is less than or equal to than every other fixpoint of T.

Now, since by (2) and (3), $glb\{x \mid T(x) = x\}$ is equal to $glb\{x \mid T(x) \leq x\}$, it is enough to show that $glb\{x \mid T(x) \leq x\}$ is less than or equal to every other fixpoint of T. By Lemma 3.1 on the facing page, $glb\{x \mid T(x) \leq x\}$ is the least prefixpoint of T. Thus, since $glb\{x \mid T(x) \leq x\}$ is less than or equal to every prefixpoint of T, it is also less than or equal to every fixpoint of T.

The proof of Part (B) follows from that of Part (A) because the complete lattice L ordered by the partial order $\leq$ is also a complete lattice ordered by the converse relation $\geq$, defined as follows: for all x and y in L, $x \geq y$ iff $y \leq x$. Note that also $\geq$ is a partial order. □

NOTE 3.3. The proof of Knaster-Tarski Theorem shows the usefulness of introducing the notions of a prefixpoint and a postfixpoint, besides that of a fixpoint.

4. Complete Partial Orders and Continuous Functions

In this section we introduce a class of mathematical structures, called cpo's, which will be used in the sequel for providing the semantics of programming languages in which one may declare and evaluate recursively defined functions.

The introduction of suitable mathematical structures for providing the semantics of recursive or interdependent definitions should not be novel to the reader. Indeed, given a system of linear equations whose coefficients are *integer numbers*, the values of the unknowns which are solutions of the system, and in this sense they are the semantics of the system, are, in general, *rational numbers* (recall the fractions occurring in the expression of the Cramer rule). For instance, the solution of the following system, which defines x in terms of y and y in terms of x:

$$\begin{cases} x = 4\,y + 1 \\ y = 2\,x + 3 \end{cases}$$

is the pair of rational numbers $x = \frac{-13}{7}$ and $y = \frac{-5}{7}$. Thus, in order to provide the meaning to the systems of linear equations with integer coefficients, one has introduced the mathematical structure of the rational numbers which is closed with respect to sums, subtractions, and multiplications, as the integer numbers, but they are also closed with respect to (non-zero) divisions.

Let us start off by introducing the following concepts.

Partial order and upper bound. Given a set D with a partial order $\sqsubseteq$, the least upper bound of a subset A of D is an element u in D such that: (i) for all $a \in A$, $a \sqsubseteq u$, that is, u is an *upper bound* of A, and (ii) for all z such that z is an upper bound of A, we have that $u \sqsubseteq z$, that is, u is the least among all upper bounds.

Cpo. A *complete partial order* (or a *cpo*, for short) $(D, \sqsubseteq_D)$ is a set D with a partial order $\sqsubseteq_D \subseteq D \times D$ such that for each chain of elements $d_0 \sqsubseteq_D d_1 \sqsubseteq_D \ldots \sqsubseteq_D d_i \sqsubseteq_D \ldots$ (also called an *ω-chain*, because that chain is function from ω to D) there exists in D the least upper bound (*lub*, for short) of the set $\{d_i \mid i \in \omega\}$, denoted $\bigsqcup_{i \in \omega} d_i$.

The set D of the cpo $(D, \sqsubseteq_D)$ is called the *carrier* of the cpo and often, in the terminology we use, it is identified with the cpo itself. For instance, we will feel free to say that the set A is a subset of the cpo $(D, \sqsubseteq_D)$, instead of saying that the set A is a subset of the carrier of D of the cpo $(D, \sqsubseteq_D)$.

The lub of an ω-chain is also called the *limit point* of the chain. Every element d_i, for $i \in \omega$, is said to be an *approximant* of the least upper bound $\bigsqcup_{i \in \omega} d_i$.

When understood from the context, we will write $\sqsubseteq$, instead of $\sqsubseteq_D$. When confusion does not arise, the cpo $(D, \sqsubseteq_D)$ will also be denoted simply by D. Sometimes, for historical reasons, we will say 'domain', instead of 'cpo'.

Cpo with bottom. Given a cpo $(D, \sqsubseteq)$, we say that it is a *cpo with bottom* if there exists in D a least element, denoted $\bot$ and called *bottom*, such that for all $d \in D$, $\bot \sqsubseteq d$.

Sometimes we will say 'cpo', instead of 'cpo with bottom', when it is understood from the context.

Discrete cpo. A *discrete cpo* is a set A with the partial order which is the identity relation (that is, for all $a, b \in A$, $a \sqsubseteq b$ iff a is b).

Flat cpo. Any set A can be extended to a *flat cpo*, denoted $A_\perp$, by: (i) adding a bottom element $\perp$, not in A, and (ii) injecting every element a of A into an element, denoted $\lfloor a \rfloor$, in $A_\perp$. We consider the following partial order on $A_\perp$: for all $a, b \in A_\perp$, $a \sqsubseteq b$ iff $a = \perp$ or $a = b$.

Given the set $T =_{def} \{true, false\}$ of the truth values and the set $N =_{def} \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ of the integer numbers, we have, respectively, the two flat cpo's $T_\perp$ and $N_\perp$ depicted in Figure 1. (The reader may also look at the notions of the lifted cpo and the lifting function $\lambda x.\lfloor x \rfloor$ on page 90.)

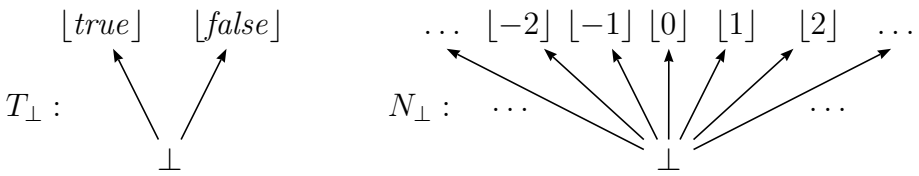


FIGURE 1. The cpo's $T_\perp$ and $N_\perp$. We have that $a \sqsubseteq b$ iff $a = b$ or there is an arrow from a to b.

Monotonic and continuous function. Let us consider the cpo's D and E. A function f from D to E is said to be *monotonic* if for all $d_1, d_2 \in D$, if $d_1 \sqsubseteq_D d_2$ then $f(d_1) \sqsubseteq_E f(d_2)$. Sometimes, in the literature, 'monotone' is used instead of 'monotonic'.

A function f from D to E is said to be *continuous* iff

(i) it is monotonic, and

(ii) it preserves the lub's, that is, for all ω-chains $d_0 \sqsubseteq_D d_1 \sqsubseteq_D \ldots \sqsubseteq_D d_i \sqsubseteq_D \ldots$ in D we have that $f(\bigsqcup_{i \in \omega} d_i) = \bigsqcup_{i \in \omega} f(d_i)$.

The set of all continuous functions from the cpo D to the cpo E is denoted by $[D \to E]$.

The composition of two continuous functions is a continuous function.

Strict function. A function f from a cpo D with bottom element $\perp_D$ to a cpo E with bottom element $\perp_E$ is said to be *strict* iff $f(\perp_D) = \perp_E$.

EXAMPLE 4.1. **[Streams]** In this example we illustrate the importance of continuous functions for making models of computational processes.

Let us consider the context-free grammar whose axiom is $\overline{s}$ and whose productions are:

$\overline{s} \to s\,\$ \mid s$ (*finite stoppered* sequences or *finite extensible* sequences)

$s \to \varepsilon \mid 0\,s \mid 1\,s$ (*finite extensible* sequences)

Thus, the finite sequences generated from the axiom $\overline{s}$ are of the form $(0+1)^*\$ + (0+1)^*$. A finite sequence generated by $\overline{s}$ is said to be a *finite stoppered stream* if it ends by \$, otherwise it is said to be a *finite extensible stream*.

As we will see in Section 3.1 of Chapter 9 (see page 278), the set of sequences which is the *minimal solution* (or the *minimal fixpoint*) of the language equation $X = AX + B$ in the unknown X, is the set A^*B.

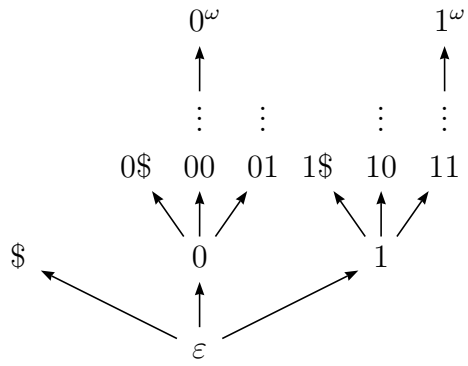


FIGURE 2. The cpo of *Streams*. We have that $s_1 \sqsubseteq s_2$ iff $s_1 = s_2$ or there is a path from s_1 to s_2 'following the arrows'. The maximal elements in the cpo of *Streams* are the streams which terminate by \$ and the infinite streams.

Thus, $(0{+}1)^*\$ + (0{+}1)^*$ is the minimal solution in the unknown $\overline{s}$ of the language equations:

$$\begin{cases} \overline{s} = s\,\$ + s \\ s = \varepsilon + 0\,s + 1\,s \end{cases}$$

Also in Section 3.1 of Chapter 9 (see page 279) we will see that the set of sequences which is the *maximal solution* (or the *maximal fixpoint*) of the equation $X = AX + B$ in the unknown X, is the set $A^*B + A^\omega$.

Thus, the set *Streams* defined as follows:

$Streams =_{def} (0{+}1)^*\$ + (0{+}1)^* + (0{+}1)^\omega$.

is the maximal solution of the language equations:

$$\begin{cases} \overline{s} = s\,\$ + s \\ s = \varepsilon + 0\,s + 1\,s \end{cases}$$

in the unknown $\overline{s}$. We call a *stream* every finite or infinite sequence in the set *Streams*.

Concatenation of streams, denoted by $\cdot$, is defined as follows:

(i) for every s_1 in $(0{+}1)^*$, for every s in *Streams*, $s_1 \cdot s$ is a (possibly infinite) stream whose prefix is s_1, and

(ii) for every s_1 in $(0{+}1)^\omega$, for every s in *Streams*, $s_1 \cdot s = s_1$.

Note that for every s_1 in $(0{+}1)^*\$$, for every s in *Streams*, $s_1 \cdot s$ is *not* defined.

The set *Streams* can be structured as a cpo by defining the following partial order $\sqsubseteq$ between its elements.

For all streams s_1 and s_2, we stipulate that (see also Figure 2):

$s_1 \sqsubseteq s_2$ iff *either* $s_1 = s_2$
or s_1 is a finite extensible sequence and $\exists s \in Streams,\ s_1 \cdot s = s_2$.

For instance, $01 \sqsubseteq 01\$$ and $01 \sqsubseteq 01001$. We have that for every finite stoppered stream $s_1\$$, the only stream s such that $s_1\$ \sqsubseteq s$, is $s_1\$$ itself.

The cardinality of the maximal elements in the cpo of *Streams* is that of the real numbers (that is, $\aleph_1$), because: (i) the cardinality of the set of streams ending by \$ is $\aleph_0$, and (ii) the cardinality of the infinite streams is $\aleph_1$.

Now, suppose that we want to define a function from *Streams* to $T_\perp$ (see Figure 1 on page 83) which tests whether or not there exists a symbol 1 in the given stream. Let that function be called *isone*.

We have the following equations: for every (finite or infinite) s in *Streams*,

$$\begin{array}{lcl} \mathit{isone}(\varepsilon) & = & \perp \\ \mathit{isone}(\$) & = & \lfloor \mathit{false} \rfloor \\ \mathit{isone}(0\,s) & = & \mathit{isone}(s) \\ \mathit{isone}(1\,s) & = & \lfloor \mathit{true} \rfloor \end{array}$$

We have that, for every $s \in (0+1)^\omega$, $\mathit{isone}(s) = \lfloor \mathit{true} \rfloor$ if there exists a symbol 1 in s. Now we have to define the value of $\mathit{isone}(0^\omega)$. It may seem reasonable to stipulate that $\mathit{isone}(0^\omega) = \lfloor \mathit{false} \rfloor$.

However, if we require that the function *isone* be a continuous function, we have that:

$\mathit{isone}(0^\omega) = \{\text{by definition of } 0^\omega\} =$

$= \mathit{isone}(\bigsqcup_{n\in\omega} 0^n) = \{\text{by requiring the continuity of the function } \mathit{isone}\} =$

$= \bigsqcup_{n\in\omega} \mathit{isone}(0^n) = \{\text{by the fact that } \mathit{isone}(0^n) = \perp\} =$

$= \bigsqcup_{n\in\omega} \perp = \perp.$

Thus, we have two options: either (i) $\mathit{isone}(0^\omega) = \lfloor \mathit{false} \rfloor$ or (ii) $\mathit{isone}(0^\omega) = \perp$.

We will reject option (i) and we will take option (ii), because we want to function *isone* to be computable according to the usual notion of Turing computability, as we now explain.

Turing computability requires that the value of a function (*isone* in our case) when it is applied to a *limit point* (0^ω in our case), should be the limit of a sequence of values, each of which is obtained by applying that function to an element of an infinite sequence of *finite approximants* of that limit point.

Turing computability requires also that, for every finite approximant y of the value of a function f applied to a limit point x_ω, there exists a suitable, finite approximant x of x_ω such that $y \sqsubseteq f(x) \sqsubseteq f(x_\omega)$.

In our case, an infinite sequence of finite approximants of 0^ω is:

$$\varepsilon \sqsubseteq 0 \sqsubseteq 00 \sqsubseteq \ldots \sqsubseteq 0^n \sqsubseteq \ldots$$

and we have that, for all $n \in \omega$, $\mathit{isone}(0^n) = \perp$. Since the least upper bound of an ω-chain whose values are all $\perp$, is $\perp$, we get that the value of $\mathit{isone}(0^\omega)$ should be $\perp$. This ensures that the function *isone* is continuous and Turing computable. □

Finite products of cpo's. Let us consider the cpo's D_1, ..., D_k. Their *product*, denoted $D_1 \times \ldots \times D_k$, is the cpo whose underlying set is the set of tuples $(d_1, \ldots, d_k)$ where $d_1 \in D_1$, ..., $d_k \in D_k$, ordered as follows:

$(d_1, \ldots, d_k) \sqsubseteq (d'_1, \ldots, d'_k)$ iff for $i = 1, \ldots, k$, we have that $d_i \sqsubseteq d'_i$.

A tuple $(d_1, \ldots, d_k)$ will also be denoted by $\langle d_1, \ldots, d_k \rangle$.
The limits points in $D_1 \times \ldots \times D_k$ are defined componentwise, that is,

$$\bigsqcup_{i\in\omega}(d_{1i}, \ldots, d_{ki}) = (\bigsqcup_{i\in\omega} d_{1i}, \ldots, \bigsqcup_{i\in\omega} d_{ki}). \qquad (\dagger)$$

Figure 3 on the next page shows the product cpo $T_\perp \times T_\perp$.

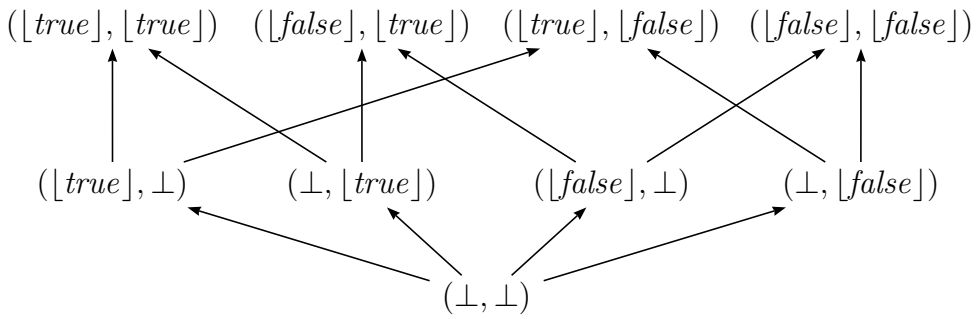


FIGURE 3. The product cpo $T_\perp \times T_\perp$. We have that $(a_1, a_2) \sqsubseteq (b_1, b_2)$ iff $(a_1, a_2) = (b_1, b_2)$ or there a path from (a_1, a_2) to (b_1, b_2) following the arrows.

NOTATION 4.2. In what follows, a function f with one argument which is a k-tuple, for $k \geq 1$, will be considered to be a function with k arguments. Thus, for instance, we will write $f(d_1, ..., d_k)$, instead of $f((d_1, ..., d_k))$, and $\lambda d_1, ..., d_k.\ f(d_1, ..., d_k)$, instead of $\lambda(d_1, ..., d_k).\ f(\pi_1((d_1, ..., d_k)),\ ...,\ \pi_k((d_1, ..., d_k)))$. □

If $k = 0$ then the product of zero cpo's is the cpo whose underlying set is the singleton $\{()\}$ with only element which is the empty tuple $()$. In what follow when considering the product of k cpo's, we will assume that k is greater than 0.

Given the cpo $D_1 \times \ldots \times D_k$, we define, for $i = 1, \ldots, k$, the *projection* function π_i from $D_1 \times \ldots \times D_k$ to D_i as follows: $\pi_i(d_1, \ldots, d_k) = d_i$ (see Figure 4). Every projection function π_i is continuous.

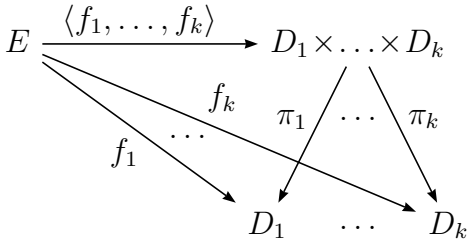


FIGURE 4. The product cpo $D_1 \times \ldots \times D_k$, the function $\langle f_1, \ldots, f_k \rangle$: $E \to D_1 \times \ldots \times D_k$, and the projection functions π_i's.

Given the continuous functions f_1: $E \to D_1, \ldots,\ f_k$: $E \to D_k$ from cpo's to cpo's, we get the tupled function $\langle f_1, \ldots, f_k \rangle$: $E \to D_1 \times \ldots \times D_k$ which is defined componentwise, that is, for $i = 1, \ldots, k$,

$\langle f_1, \ldots, f_k \rangle(e) =_{def} (f_1(e), \ldots, f_k(e))$.

Thus, the definition of the function $\langle f_1, \ldots, f_k \rangle$ is derived by imposing the commutativity of the diagram of Figure 4.

The function $\langle f_1, \ldots, f_k \rangle$ is continuous, as we now show. Take an ω-chain $e_0 \sqsubseteq e_1 \sqsubseteq \ldots \sqsubseteq e_i \sqsubseteq \ldots$ of elements in E. We have that

$\langle f_1, \ldots, f_k \rangle(\bigsqcup_{i \in \omega} e_i) = \bigsqcup_{i \in \omega}(\langle f_1, \ldots, f_k \rangle(e_i))$.

Indeed, $\langle f_1, \ldots, f_k\rangle(\bigsqcup_{i\in\omega} e_i) = \{$by definition of $\langle f_1, \ldots, f_k\rangle\} =$
$= (f_1(\bigsqcup_{i\in\omega} e_i), \ldots, f_k(\bigsqcup_{i\in\omega} e_i)) = \{$each function f_i is continuous$\} =$
$= (\bigsqcup_{i\in\omega} f_1(e_i), \ldots, \bigsqcup_{i\in\omega} f_k(e_i)) = \{$lub's of tuples are defined componentwise$\} =$
$= \bigsqcup_{i\in\omega}(f_1(e_i), \ldots, f_k(e_i)) = \{$by definition of $\langle f_1, \ldots, f_k\rangle\} =$
$= \bigsqcup_{i\in\omega}(\langle f_1, \ldots, f_k\rangle(e_i))$.

We have that: for $i = 1, \ldots, k$, for all $e \in E$, $\pi_i(\langle f_1, \ldots, f_k\rangle(e)) = f_i(e)$. Actually, it can easily be shown that $\langle f_1, \ldots, f_k\rangle$ is the unique function which enjoys that property (see Figure 4 on the facing page).

Now let us prove the following lemmata which are useful in what follows.

LEMMA 4.3. Given the cpo's E, D_1, ..., D_k, we have that h: $E \to D_1 \times \ldots \times D_k$ is a continuous function iff for $i = 1, \ldots, k$, $\pi_i \circ h$ is a continuous function.

LEMMA 4.4. Consider a cpo D and a subset $\{d_{ij} \mid i, j \in \omega\}$ of its elements. Suppose that for all m_1, m_2, n_1, $n_2 \in \omega$, $d_{m_1,m_2} \sqsubseteq d_{n_1,n_2}$ iff $m_1 \leq n_1$ and $m_2 \leq n_2$. We have that:

$$\underset{(1)}{\bigsqcup_{m,n\in\omega} d_{m,n}} = \underset{(2)}{\bigsqcup_{m\in\omega}(\bigsqcup_{n\in\omega} d_{m,n})} = \underset{(3)}{\bigsqcup_{n\in\omega}(\bigsqcup_{m\in\omega} d_{m,n})} = \underset{(4)}{\bigsqcup_{n\in\omega} d_{n,n}}$$

PROOF. First we prove that (1) $\sqsubseteq$ (4). By definition of the least upper bound, it is enough to prove that for all $m, n \in \omega$, $d_{m,n} \sqsubseteq \bigsqcup_{i\in\omega} d_{i,i}$. Indeed, we have that $d_{m,n} \sqsubseteq d_{\max(m,n),\max(m,n)}$. Moreover, $d_{\max(m,n),\max(m,n)} \sqsubseteq \bigsqcup_{i\in\omega} d_{i,i}$ because $d_{\max(m,n),\max(m,n)}$ is one of the elements of $\{d_{i,i} \mid i \in \omega\}$. Then by transitivity of $\sqsubseteq$, we get that (1) $\sqsubseteq$ (4).

Then we prove that (4) $\sqsubseteq$ (1). This follows from the property that for all $n \in \omega$, $d_{n,n} \sqsubseteq \bigsqcup_{i,j\in\omega} d_{i,j}$ because $d_{n,n}$ is one of the elements of $\{d_{i,j} \mid i, j \in \omega\}$.

Thus, we get that (1) = (4). By similar proofs we can show that (2) = (4) and (3) = (4).

In particular, the proof of (4) $\sqsubseteq$ (2) follows from the facts that: (i) for all $n \in \omega$, $d_{n,n} \sqsubseteq \bigsqcup_{j\in\omega} d_{n,j}$ (because $d_{n,n}$ is one of the elements of $\{d_{n,j} \mid j \in \omega\}$) and (ii) for all $n \in \omega$, $(\bigsqcup_{j\in\omega} d_{n,j}) \sqsubseteq \big(\bigsqcup_{i\in\omega}(\bigsqcup_{j\in\omega} d_{i,j})\big)$ (because $\bigsqcup_{j\in\omega} d_{n,j}$ is one of the elements of $\{\bigsqcup_{j\in\omega} d_{i,j} \mid i \in \omega\}$). □

LEMMA 4.5. Let us take a natural number $k \geq 1$. The function f: $D_1 \times \ldots \times D_k \to E$ is continuous iff f is *continuous in each of its k arguments*, that is, for $i = 1, \ldots, k$, for all $(d_1, \ldots, d_{i-1}, d_{i+1}, \ldots, d_k) \in D_1 \times \ldots \times D_{i-1} \times D_{i+1} \times \ldots \times D_k$, the function f_i: $D_i \to E$ such that $\forall d \in D_i$, $f_i(d) = f(d_1, \ldots, d_{i-1}, d, d_{i+1}, \ldots, d_k)$ is continuous.

PROOF. (*only-if part*) This proof follows from the fact that a particular ω-chain of elements in $D_1 \times \ldots \times D_k$ can be obtained by fixing the values of $k-1$ coordinates and taking an ω-chain of the remaining coordinate.

(*if part*) Let us consider the case of $k = 2$. The proof of the general case where k is any positive natural number, is similar. Let f be a function in $D_1 \times D_2 \to E$. Let us consider the ω-chain $(d_{10}, d_{20}) \sqsubseteq \ldots \sqsubseteq (d_{1n}, d_{2n}) \sqsubseteq \ldots$ in $D_1 \times D_2$. We have that:
$f(\bigsqcup_{n\in\omega}(d_{1n}, d_{2n})) = \{$lub's are computed componentwise$\} =$
$= f(\bigsqcup_{n\in\omega} d_{1n}, \bigsqcup_{m\in\omega} d_{2m}) = \{f$ is continuous in its first argument$\} =$
$= \bigsqcup_{n\in\omega} f(d_{1n}, \bigsqcup_{m\in\omega} d_{2m}) = \{f$ is continuous in its second argument$\} =$

$= \bigsqcup_{n\in\omega} \left(\bigsqcup_{m\in\omega} f(d_{1n},\, d_{2m}) \right) =$ {by Lemma 4.4 on the previous page} $=$
$= \bigsqcup_{n\in\omega} f(d_{1n}, d_{2n})$.

Note that, as usual, the application of the function f to a pair (d_1, d_2) has been denoted by $f(d_1, d_2)$, instead of $f((d_1, d_2))$. □

With reference to the above Lemma 4.5 on the preceding page, note that in Real Analysis it may be the case that a function f is continuous in each of its arguments and yet f is *not* continuous. Consider, for instance, the function

$$f(x,y) = \begin{cases} \dfrac{xy}{x^2+y^2} & \textit{if } \ (x,y) \neq (0,0) \\ 0 & \textit{otherwise} \end{cases}$$

Indeed, (i) along the line $x = y$, we have that $f(x, y) = 1/2$ if $(x, y) \neq (0, 0)$ and $f(x, y) = 0$ if $(x, y) = (0, 0)$, (ii) along the line $x = 0$ we have that $f(x, y) = 0$, and (iii) along the line $y = 0$ we have that $f(x, y) = 0$.

Function space. Now we show that given any cpo D and any cpo E (not necessarily distinct), the set of continuous functions from D to E, denoted $[D \to E]$, is a cpo. We order any two functions f and g in $[D \to E]$ pointwise, that is,

$f \sqsubseteq g$ iff for all $d \in D$, $f(d) \sqsubseteq g(d)$

If E has a bottom element $\bot$ then also the set of all continuous functions in $[D \to E]$ has the bottom element which is the constant function $\lambda x.\bot$.

The function which is a *limit function*, that is, the least upper bound of an ω-chain of continuous functions in $[D \to E]$ is computed pointwise, that is,

$$\text{for all } d \in D,\ \left(\bigsqcup_{i\in\omega} f_i\right)(d) =_{def} \bigsqcup_{i\in\omega} (f_i\, d). \tag{1}$$

To show that function space $[D \to E]$ is indeed a cpo, we have to show that for every ω-chain of continuous functions $f_0 \sqsubseteq f_1 \sqsubseteq \ldots \sqsubseteq f_i \sqsubseteq \ldots$ we have that its least upper bound $\bigsqcup_{i\in\omega} f_i$ is a continuous function, that is, we have to show that for every ω-chain $d_0 \sqsubseteq d_1 \sqsubseteq \ldots \sqsubseteq d_j \sqsubseteq \ldots$, we have that

$\left(\bigsqcup_{i\in\omega} f_i\right) \left(\bigsqcup_{j\in\omega} d_j\right) = \bigsqcup_{j\in\omega} \left(\left(\bigsqcup_{i\in\omega} f_i\right) d_j \right)$.

Indeed, $\left(\bigsqcup_{i\in\omega} f_i\right) \left(\bigsqcup_{j\in\omega} d\right)$ = {by (1) on the current page} $=$
$= \bigsqcup_{i\in\omega} \left(f_i \left(\bigsqcup_{j\in\omega} d\right)\right)$ = {by continuity of the f_i's} $=$
$= \bigsqcup_{i\in\omega} \left(\bigsqcup_{j\in\omega} (f_i\, d)\right)$ = {by Lemma 4.4 on the preceding page} $=$
$= \bigsqcup_{j\in\omega} \left(\bigsqcup_{i\in\omega} (f_i\, d)\right)$ = {by (1) on the current page} $=$
$= \bigsqcup_{j\in\omega} \left(\left(\bigsqcup_{i\in\omega} f_i\right) d_j \right)$.

Apply, curry and uncurry. Let us consider the three cpo's F, D, and E. Let us also consider the continuous function $f \in [[F \times D] \to E]$. The diagram of Figure 5 on the facing page, where:

$$\begin{aligned} &\mathit{curry} \in [[F \times D \to E] \to [F \to [D \to E]]] &&=_{def}\ \lambda f \in [[F \times D] \to E].\, \lambda v \in F.\, \lambda d \in D.\, f(v,d) \\ &\mathit{id}_D \in [D \to D] &&=_{def}\ \lambda d \in D.\, d \\ &\mathit{apply} \in [[[D \to E] \times D] \to E] &&=_{def}\ \lambda (g,d) \in [[D \to E] \times D].\, g(d) \end{aligned}$$

is commutative, that is, $\forall v \in F,\ \forall d \in D,\ f(v,d) = \mathit{apply}((\mathit{curry}(f))\, v,\, d)$.

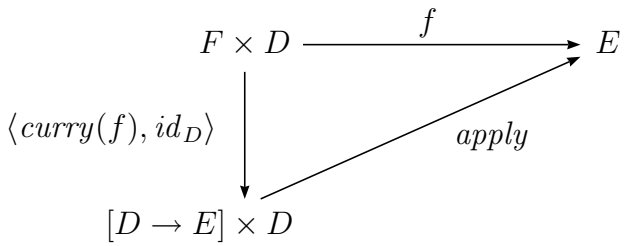


FIGURE 5. The function f is a continuous function in $[F \times D \to E]$. The function $curry \in [[F \times D \to E] \to [F \to [D \to E]]]$ and the function $apply \in [[D \to E] \times D \to E]$. The function $curry(f)$ is a continuous function in $[F \to [D \to E]]$. The function $id_D \in [D \to D]$ is the identity function (which is a continuous function).

Let us consider the continuous function $f \in [[F \times D \to E]$ with two arguments, say v and d. Then the function $curry(f) \in [F \to [D \to E]]$ is a continuous function which given the first argument $v \in F$, returns the function $\lambda d.\, f(v, d) \in [D \to E]$. This function, given the second argument $d \in D$, returns the value of $f(v, d)$. In Figure 5 $id_D \in [D \to D]$ is the (continuous) identity function.

The function from the cpo $[F \times D]$ to the cpo $[D \to E] \times D$ is made out of the following two components: (i) $curry(f)$ that maps F to $[D \to E]$, and (ii) id_D that maps D to D.

Now let us give an example of use of the functions *curry* and *apply*.

Let us consider the function $\lambda x, y.\, sum(x, y) \in [[N \times N] \to N]$. We have that $curry(sum))$ is the function $\lambda x.(\lambda y.\, sum(x, y)) \in [N \to [N \to N]]$. For any input value x, $(curry(sum))\, x$ evaluates to the function $\lambda z.\, x + z$. This function is then applied to the argument y, which is the second argument of the given function *sum*. Formally, this application of $\lambda z.\, x + z$ to the argument y is the evaluation of the expression $apply(\lambda z.x{+}z,\ y)$, and the evaluation of this expression returns the desired output value $x{+}y$.

This example shows that, in general, a function of $m{+}n$ arguments, with $m \geq 1$ and $n \geq 1$, can be viewed as a function of m arguments which returns a function of the remaining n arguments.

The functions *curry* and *apply* allow us to formalize also the behaviour of a software system M which evaluates a given input program P for a given input value u by using a compiler.

Indeed, the value of $M(P, u)$ can be computed by: (i) first, applying the function $curry(M)$ to the program P, thereby getting the object program P_{obj} (thus, $curry(M)$ can be viewed as a compiler) (see also Figure 6 on the following page), and then (ii) applying P_{obj} to the argument u, that is, evaluating $apply(P_{obj}, u)$.

Now the value of $apply(P_{obj}, u)$ is the desired output result $P(u)$ because we have that:

$apply(P_{obj}, u) = \{$by definition of $apply\} =$
$= P_{obj}(u) = \{$by definition of $curry\} =$
$= (curry(M)(P))\, u =$
$= \{$by commutativity of the diagram of Figure 5 on the previous page$\} =$
$= M(P, u) = \{$by definition of the system $M\} =$
$= P(u).$

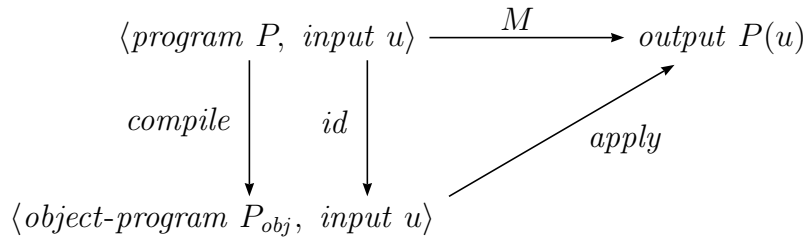


FIGURE 6. The compilation process. We have that:
(i) $compile = curry(M)$, and (ii) $apply(P_{obj}, u) = P_{obj}(u) = P(u)$.

One can show that given a continuous function $f \in [[F \times D \to E]$, $curry(f)$ is the unique continuous function in $[F \to [D \to E]]$ such that for all $v \in F$, $d \in D$, we have that $f(v, d) = apply((curry(f))\, v,\, d)$.

The function *curry* has an inverse function, called *uncurry*, which is defined as follows:

$$uncurry =_{def} \lambda f \in [F \to [D \to E]].\, \lambda (v, d) \in [F \times D].\, (f(v))\, (d).$$

We have that for all functions $f \in [F \times D \to E]$,

$$uncurry(curry(f)) = f,$$

and for all functions $f \in [F \to [D \to E]$,

$$curry(uncurry(f)) = f.$$

One can show that *curry*, *apply*, and *uncurry* are continuous functions. In particular, *apply* is a continuous function because, by Lemma 4.5 on page 87, it is continuous in each argument separately.

Lifted cpo. Given a cpo D, the *lifted cpo* of D, denoted $D_\perp$, is defined as follows. (The reader may also look at the definition of a flat cpo on page 83.)

The underlying set of $D_\perp$ is $\{\lfloor d \rfloor \mid d \in D\} \cup \{\perp_D\}$, where:
(i) $\perp_D$ is the bottom element of $D_\perp$,
(ii) for every $d \in D$, $\lfloor d \rfloor \neq \perp_D$, and
(iii) the function $\lambda x. \lfloor x \rfloor$ from D to $D_\perp$ is a bijection from D to $D_\perp - \{\perp_D\}$.

The function $\lambda x. \lfloor x \rfloor$ from D to $D_\perp$ is called the *lifting function*.

Given the partial order $\sqsubseteq$ on D, the partial order on $D_\perp$, also denoted $\sqsubseteq$, is defined as follows: for all $d'_1, d'_2 \in D_\perp$,

$$d'_1 \sqsubseteq d'_2 \quad \text{iff} \quad d'_1 = \perp_D \ \vee\ (\exists d_1, d_2 \in D.\, d'_1 = \lfloor d_1 \rfloor \ \wedge\ d'_2 = \lfloor d_2 \rfloor \ \wedge\ d_1 \sqsubseteq d_2).$$

As a consequence of this definition, we say that the lifting function $\lambda x.\lfloor x \rfloor$ *preserves the order.*

We define $\bigsqcup_{i\in\omega} \lfloor d_i \rfloor$ to be $\lfloor \bigsqcup_{i\in\omega} d_i \rfloor$. This definition is a consequence of the following two points.
Point (1): $\lfloor \bigsqcup_{i\in\omega} d_i \rfloor$ is an upper bound of the set $\{\lfloor d_i \rfloor \mid i \geq 0\}$, and
Point (2): $\lfloor \bigsqcup_{i\in\omega} d_i \rfloor$ is the least upper bound of the set $\{\lfloor d_i \rfloor \mid \geq 0\}$.

Point (1) follows from the fact that for each $i \geq 0$, $\lfloor d_i \rfloor \sqsubseteq \lfloor \bigsqcup_{i\in\omega} d_i \rfloor$ because $d_i \sqsubseteq \bigsqcup_{i\in\omega} d_i$.

Point (2) can be shown as follows. Let us consider any other element, say e, such that for all $i \geq 0$, $\lfloor d_i \rfloor \sqsubseteq e$ then we have that $\lfloor \bigsqcup_{i\in\omega} d_i \rfloor \sqsubseteq e$. Indeed, let e' be the element in D such that $e = \lfloor e' \rfloor$. We have that for all $i \geq 0$, $d_i \sqsubseteq e'$ because $\lambda x.\lfloor x \rfloor$ preserves the order. Thus, $\bigsqcup_{i\in\omega} d_i \sqsubseteq e'$ and, thus, $\lfloor \bigsqcup_{i\in\omega} d_i \rfloor \sqsubseteq e$.

The lifting function $\lambda x.\lfloor x \rfloor$ is continuous.

Down function. Given a cpo D with bottom element $\bot$ and its lifted cpo $D_\bot$ with bottom element $\bot_D$ (see Figure 7), let us introduce the function *down*: $D_\bot \to D$ which is defined as follows: for all $d' \in D_\bot$,

$$down(d') =_{def} \begin{cases} d & \text{if for some } d \in D,\ d' = \lfloor d \rfloor \\ \bot & \text{otherwise (that is, if } d' = \bot_D \in D_\bot) \end{cases}$$

The function $\lambda x.down(x)$ is continuous.

We have that: $\big(d' = \bot_D \text{ or } d' = \lfloor \bot \rfloor\big)$ iff $down(d') = \bot$.

For any $d \in D$, $down(\lfloor d \rfloor) = d$.

For any $d' \in D_\bot$, if $d' \neq \bot_D$ then $\lfloor down(d') \rfloor = d'$.

Let construct. Let us consider a lifted cpo $D_\bot$ with bottom element $\bot_D$ and a cpo E with bottom element $\bot_E$. We define the *let* construct from $D_\bot$ to E as follows (see also Figure 7 on the next page):

$$let\, x \Leftarrow d' \,.\, e \ =_{def}\ \ if\ d' = \bot_D\ then\ \bot_E\ else\ ((\lambda x.e)d),\ \text{ where } d' = \lfloor d \rfloor.$$

Recall that $(\lambda x.e)d = e[d/x]$. The *let* construct is continuous in the sense that the function

$$\lambda d'.\,(let\, x \Leftarrow d' \,.\, e)$$

is continuous.

For reasons of simplicity, we will write $let\, x_1 \Leftarrow d_1, \ldots, x_n \Leftarrow d_n \,.\, e$, instead of $let\, x_1 \Leftarrow d_1 \,.\, (\ldots \,.\, (let\, x_n \Leftarrow d_n \,.\, e) \ldots)$.

By definition of the *let* construct we have that for any $d' \in D_\bot$, the function *down*: $D_\bot \to D$ satisfies the following:

$$down(d') \ =\ let\, x \Leftarrow d' \,.\, x$$

Lifted function. Let us consider a cpo D, its lifted cpo $D_\bot$ with bottom element $\bot_D$, a cpo E with bottom element $\bot_E$, and function $\lambda x.e \in [D \to E]$. We define the *lifted function* $\lfloor \lambda x.e \rfloor$ to be the function in $[D_\bot \to E]$ as follows:

$$\lfloor \lambda x.e \rfloor\, d' =_{def}\ if\ d' = \bot_D\ then\ \bot_E\ else\ (\lambda x.e)\, d, \quad \text{where } d' = \lfloor d \rfloor.$$

We have that: $\lfloor \lambda x.e \rfloor\, d' = let\, x \Leftarrow d' \,.\, e$.

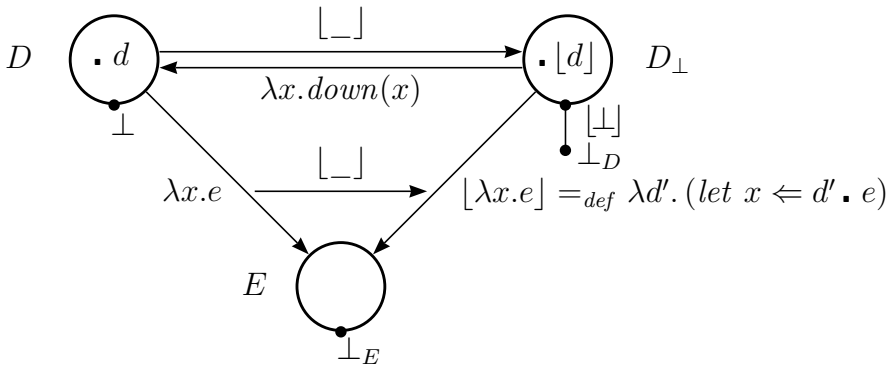


FIGURE 7. The lifting operation $\lfloor \_ \rfloor$ and the *let* construct. The cpo D need not have a bottom element $\bot$ (but in this picture we have assumed that there exists one and we have represented both $\bot$ in D and $\lfloor \bot \rfloor$ in $D_\bot$), while the cpo E has the bottom element $\bot_E$. The function $\lambda d'.(let\ x \Leftarrow d' \centerdot e)$ behaves as follows: $\lambda d'.\big(if\ d' = \bot_D\ then\ \bot_E\ else\ ((\lambda x.e)\ d)$, where $d' = \lfloor d \rfloor\big)$.

Strict extension of arithmetic and boolean operations. Given the arithmetic operation $+\colon N \times N \to N$, its *strict extension* (or *natural extension*) $+_\bot\colon N_\bot \times N_\bot \to N_\bot$, where $N_\bot$ is the lifted cpo of N, is defined as follows:
for all $n_1, n_2 \in N_\bot$,

$$n_1 +_\bot n_2 =_{def} \begin{cases} \lfloor m_1 + m_2 \rfloor & \text{if for some } m_1, m_2 \in N,\ \ n_1 = \lfloor m_1 \rfloor \text{ and } n_2 = \lfloor m_2 \rfloor \\ \bot & \text{otherwise} \end{cases}$$

Analogously, for the arithmetic operations $-$ and $\times$, and the boolean operations *not*, *or*, *and*, and *implies*. Thus, if any of the arguments of the strict extension of an arithmetic or a boolean operation is $\bot$, then the result of that arithmetic or boolean operation is $\bot$. In particular, we have that $0 \times_\bot \bot = \bot \times_\bot 0 = \bot$.

The strict extension of an arithmetic or a boolean operation can be expressed in terms of the *let* construct. For instance, for the operation $+_\bot$ we have that:

$$n_1 +_\bot n_2 =_{def} let\ m_1 \Leftarrow n_1,\ m_2 \Leftarrow n_2 \centerdot \lfloor m_1 + m_2 \rfloor$$

Sum. Let us consider the cpo's $D_1, \ldots, D_k$, for $k \geq 1$. Their *sum*, denoted $D_1 + \ldots + D_k$, is the cpo whose underlying set is:

$$\{in_1(d) \mid d \in D_1\} \cup \ldots \cup \{in_k(d) \mid d \in D_k\}$$

where: (1) for $i = 1, \ldots, k$, each function $in_i\colon D_i \to D_1 + \ldots + D_k$ is an injection (thus, so to speak, we have a copy of each D_i in $D_1 + \ldots + D_k$), and (2) for $i, j = 1, \ldots, k$, for all $d \in D_i$, $d' \in D_j$, if $i \neq j$ then $in_i(d) \neq in_j(d')$.

The partial order of the sum cpo is defined as follows:

$$d \sqsubseteq d' \quad \text{iff} \quad \exists i,\ 1 \leq i \leq k,\ \exists\, a, b \in D_i,\ d = in_i(a) \text{ and } d' = in_i(b) \text{ and } a \sqsubseteq b$$

(See also Figure 8 on the facing page). If $k = 0$ we have that the sum of zero cpo's is the cpo whose underlying set is the empty set which, indeed, is the neutral element of the union operation between sets. In what follow when considering the sum of k cpo's, we will assume that k is greater than 0.

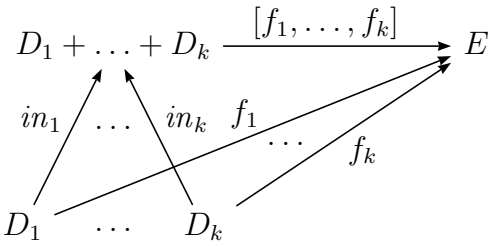


FIGURE 8. The sum cpo $D_1 + \ldots + D_k$, the sum function $[f_1, \ldots, f_k]$: $D_1 + \ldots + D_k \to E$, and the injection functions $in_1, \ldots, in_k$.

Note that even if for $i = 1, \ldots, k$, every cpo D_i has a bottom element then the sum cpo $D_1 + \ldots + D_k$ may lack a bottom element. We leave it to the reader to show that for $i = 1, \ldots, k$, the injection function in_i is a continuous function.

Given the continuous functions f_1: $D_1 \to E$, ..., f_k: $D_k \to E$, we define the sum function $[f_1, \ldots, f_k]$: $D_1 + \ldots + D_k \to E$ by imposing the commutativity of the diagram of Figure 8), that is:

for $i = 1, \ldots, k$, for all $d \in D_i$, $[f_1, \ldots, f_k](in_i(d)) = f_i(d)$.

The function $[f_1, \ldots, f_k]$ is continuous, that is,

$[f_1, \ldots, f_k](\bigsqcup_{n \in \omega} d_n) = \bigsqcup_{n \in \omega}([f_1, \ldots, f_k](d_n))$.

Indeed, take an ω-chain $d_0 \sqsubseteq d_1 \sqsubseteq \ldots \sqsubseteq d_n \sqsubseteq \ldots$ of elements in $D_1 + \ldots + D_k$. By definition of the partial order $\sqsubseteq$ in $D_1 + \ldots + D_k$, we have that: (1) there exists i, with $1 \leq i \leq k$, such that $d_{i0} \sqsubseteq d_{i1} \sqsubseteq \ldots \sqsubseteq d_{in} \sqsubseteq \ldots$ is an ω-chain in D_i, and (2) for all $n \in \omega$, $d_n = in_i(d_{in})$. Thus, we have that:

$[f_1, \ldots, f_k](\bigsqcup_{n \in \omega} d_n)$ iff {by definition of $[f_1, \ldots, f_k]$}

iff there exists i, with $1 \leq i \leq k$, $f_i(\bigsqcup_{n \in \omega} d_{in})$ iff {f_i is a continuous function}

iff there exists i, with $1 \leq i \leq k$, $\bigsqcup_{n \in \omega}(f_i(d_{in}))$ iff {by definition of $[f_1, \ldots, f_k]$}

iff $\bigsqcup_{n \in \omega}([f_1, \ldots, f_k](d_n))$.

We have that: for $i = 1, \ldots, k$, for all $d \in D_i$, $[f_1, \ldots, f_k](in_i(d)) = f_i(d)$. Actually, it can easily be shown that $[f_1, \ldots, f_k]$ is the unique function which enjoys that property.

Conditional. We will consider three kinds of conditional constructs (see the following Points (1), (2), and (3)).

(1) Let T be the discrete cpo $\{true, false\}$. Given a cpo D, the function *cond*: $T \times D \times D \to D$ is defined as follows:

for all $b \in \{true, false\}$, for all $d_1, d_2 \in D$,

$$cond(b, d_1, d_2) =_{def} [\lambda x \in \{true\}.\, d_1,\ \lambda x \in \{false\}.\, d_2]\,(b)$$

where: (i) x does *not* occur in d_1, (ii) x does *not* occur in d_2, and (iii) $[f, g]$ denotes the sum (in the cpo sense) of the functions f and g (see also Figure 9 on the following page).

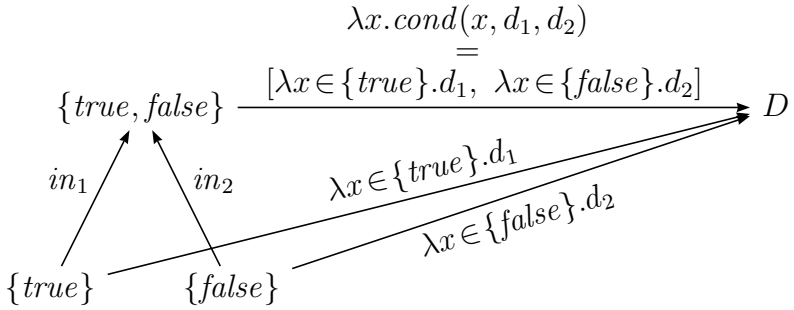


FIGURE 9. The function *cond*: $\{true, false\} \times D \times D \to D$. We have that for all $d_1, d_2 \in D$, the function $\lambda x.cond(x, d_1, d_2)$ behaves as the sum function $[\lambda x \in \{true\}.d_1,\ \lambda x \in \{false\}.d_2]$. We assume that x occurs neither in d_1 nor in d_2.

From the commutativity of the diagram of Figure 9 we have that:

for all $d_1, d_2 \in D$,

$$[\lambda x \in \{true\}.d_1,\ \lambda x \in \{false\}.d_2]\,(true) = d_1$$
$$[\lambda x \in \{true\}.d_1,\ \lambda x \in \{false\}.d_2]\,(false) = d_2$$

Thus, by the definition of *cond*, we have that:

for all $d_1, d_2 \in D$,

$$cond(true, d_1, d_2) = d_1$$
$$cond(false, d_1, d_2) = d_2$$

The function *cond* is continuous.

(2) Let $T_\perp$ be the cpo $\{true, false\}_\perp$ and D be a cpo with bottom element $\perp_D$. The function $\lambda b, d_1, d_2.\ b \to d_1 \,|\, d_2 : T_\perp \times D \times D \to D$ is defined as follows:

for all $d_1, d_2 \in D$,

$$\begin{aligned} \perp \to d_1 \mid d_2 &= \perp_D \quad \text{(where } \perp \in T_\perp\text{)} \\ \lfloor true \rfloor \to d_1 \mid d_2 &= d_1 \\ \lfloor false \rfloor \to d_1 \mid d_2 &= d_2 \end{aligned}$$

The function $\lambda b, d_1, d_2.\ b \to d_1 \,|\, d_2$ is continuous, but it is *not* the strict extension of any function in $T_\perp \times D \times D \to D$ because the result may be different from $\perp_D$ even if an argument is $\perp_D$ (indeed, if $d_1 \neq \perp_D$ then $\lfloor true \rfloor \to d_1 \mid \perp_D = d_1 \neq \perp_D$) (see also Figure 10 on the facing page). We have that for all $b \in T_\perp$, for all $d_1, d_2 \in D$,

$$b \to d_1 \,|\, d_2 =_{def} let\, b' \Leftarrow b \,.\, cond(b', d_1, d_2).$$

(3) Given the cpo $N_\perp$ with bottom element $\perp$ and the cpo D with bottom element $\perp_D$, the function *Cond*: $N_\perp \times D \times D \to D$ is defined as follows:

for all $n \in N$, for all $d_1, d_2 \in D$,

$$\begin{aligned} Cond(\perp, d_1, d_2) &= \perp_D \quad \text{(where } \perp \in N_\perp\text{)} \\ Cond(\lfloor 0 \rfloor, d_1, d_2) &= d_1 \\ Cond(\lfloor n \rfloor, d_1, d_2) &= d_2 \quad \text{if } n \neq 0 \end{aligned}$$

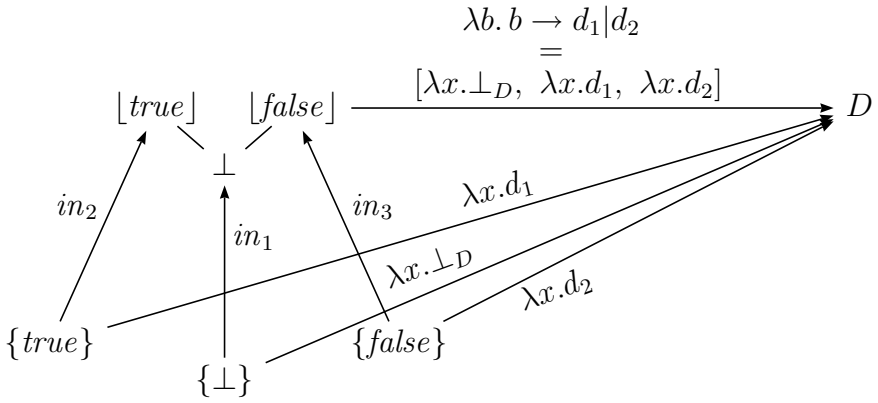


FIGURE 10. The function $\lambda b, d_1, d_2.\, b \to d_1|d_2 : T_\perp \times D \times D \to D$. The bottom element of $T_\perp$ is $\perp$. The cpo D is assumed to have a bottom element $\perp_D$. We have that for all $d_1, d_2 \in D$, $\lambda b.b \to d_1|d_2$ behaves as the sum function $[\lambda x.\perp_D,\ \lambda x.d_1,\ \lambda x.d_2]$. We assume that b occurs neither in d_1 nor in d_2.

We can define the function *Cond* in terms of the function *iszero*: $N \to \{true, false\}$ defined as follows:

$$iszero(0) = true$$
$$iszero(n) = false \quad \text{if } n \neq 0$$

We have that:

$$Cond(z, d_1, d_2) =_{def} (let\ n \Leftarrow z \centerdot \lfloor iszero(n) \rfloor) \to d_1 \,|\, d_2.$$

The function *Cond* is continuous (see also Figure 11).

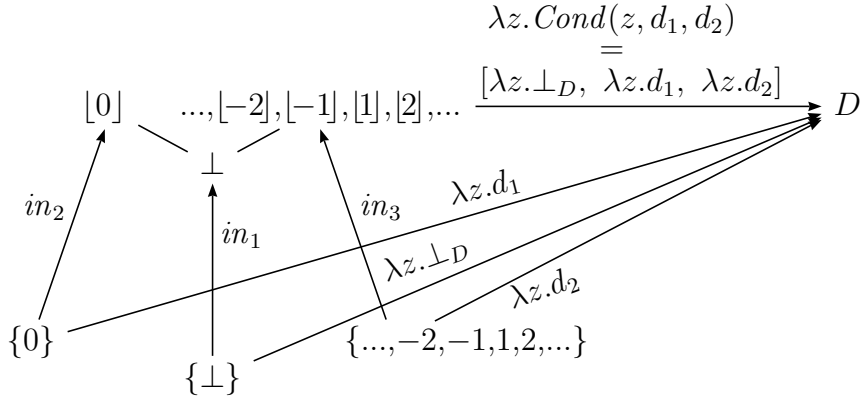


FIGURE 11. The function *Cond*: $N_\perp \times D \times D \to D$. The bottom element of $N_\perp$ is $\perp$. The cpo D is assumed to have a bottom element $\perp_D$. We have that for all $d_1, d_2 \in D$, $\lambda z.Cond(z, d_1, d_2)$ behaves as the sum function $[\lambda z.\perp_D,\ \lambda z.d_1,\ \lambda z.d_2]$. We assume that z occurs neither in d_1 nor in d_2.

Note that for any $n \in N$, $Cond(\perp, \lfloor n \rfloor, \lfloor n \rfloor) = \perp_D$ and $Cond(\lfloor 0 \rfloor, \lfloor n \rfloor, \perp_D) = \lfloor n \rfloor$. Thus, *Cond* is *not* the strict extension of any function in $N \times D \times D \to D$.

Case construct. Given the cpo E and the sum cpo $D_1 + \ldots + D_k$. Let us consider the $k\,(\geq 1)$ continuous functions $\lambda x_1.e_1$: $D_1 \to E$, $\ldots$, $\lambda x_k.e_k$: $D_k \to E$. Then the function $[\lambda x_1.e_1, \ldots, \lambda x_k.e_k]$: $D_1 + \ldots + D_k \to E$ behaves as follows:

for $i = 1, \ldots, k$, for all $d \in D_1 + \ldots + D_k$,

if $d = in_i(x_i)$ for some $x_i \in D_i$, then $[\lambda x_1.e_1, \ldots, \lambda x_k.e_k](d) = e_i$.

Obviously, for $i = 1, \ldots, k$, the value of the expression e_i may depend on the value of the input x_i. Thus, the function $[\lambda x_1.e, \ldots, \lambda x_k.e_k]$ behaves as the *case construct*

$$\begin{array}{lll} \mathit{case}\ d\ \mathit{of}\ & in_1(x_1)\,.\,e_1 & | \\ & \ldots & | \\ & in_k(x_k)\,.\,e_k & \end{array}$$

which, for $i = 1, \ldots, k$, returns e_i whenever $d = in_i(x_i)$ for some $x_i \in D_i$.

Note that the *case construct* is a generalization of the function *cond*. Indeed, if we take $k = 2$, $D_1 = \{true\}$, and $D_2 = \{false\}$, the *case construct* becomes the function *cond*.

Fixpoint. Let us consider a cpo D with bottom $\bot$. Let us define the following function $fix \in [D \to D] \to D$:

$$fix =_{def} \lambda f.\ \bigsqcup_{n\in\omega}(f^n(\bot)). \qquad (FixDefinition)$$

As we will show below, the function *fix* is itself continuous, that is, $fix \in [[D \to D] \to D]$. We have the following important Theorems 4.7–4.10.

We first introduce the notion of a prefixpoint (see also page 80).

DEFINITION 4.6. **[Prefixpoint]** Given a cpo E and a continuous function $f \in [E \to E]$, we say that an element $x \in E$ is a *prefixpoint* of f iff $f(x) \sqsubseteq x$.

THEOREM 4.7. **[Kleene Theorem]** Given a cpo D, ordered by $\sqsubseteq$, with bottom element $\bot$, and a continuous function $f \in [D \to D]$, we have that:

(i) $fix(f) =_{def} \bigsqcup_{n\in\omega}(f^n(\bot))$ is a fixpoint of f, that is, $f(fix(f)) = fix(f)$,

(ii) for all $z \in D$, if $f(z) \sqsubseteq z$ then $fix(f) \sqsubseteq z$, that is, $fix(f)$ is the minimal prefixpoint of f, and

(iii) $fix(f)$ is the minimal fixpoint of f, that is, for all $z \in D$, if $f(z) = z$ then $fix(f) \sqsubseteq z$.

PROOF. (i) $fix(f)$ is a fixpoint of f. We have to show that $f(\bigsqcup_{n\in\omega} f^n(\bot)) = \bigsqcup_{n\in\omega} f^n(\bot)$.

Since $f^0(\bot) \sqsubseteq f^1(\bot) \sqsubseteq \ldots \sqsubseteq f^n(\bot) \sqsubseteq \ldots$ is an ω-chain and f is continuous, we have that: $f(\bigsqcup_{n\in\omega} f^n(\bot)) = \bigsqcup_{n\in\omega} f^{n+1}(\bot)$. Now $\bigsqcup_{n\in\omega} f^{n+1}(\bot) = \bigsqcup_{n\in\omega} f^n(\bot)$ because for all $d \in D$, $d \sqcup \bot = d$ (by $d \sqcup \bot$ we have denoted the least upper bound, in the cpo D, of d and $\bot$).

(ii) $fix(f)$ is the minimal prefixpoint of f. Take any z such that $f(z) \sqsubseteq z$. We have to show that $fix(f) \sqsubseteq z$. It is enough to show that $\forall n \in \omega$, $f^n(\bot) \sqsubseteq z$. This can be done by induction on $n \geq 0$.

(*Basis*) Obviously, we have that $\bot \sqsubseteq z$.

(*Step*) Assume $f^n(\bot) \sqsubseteq z$. By monotonicity, we have $f^{n+1}(\bot) \sqsubseteq f(z)$ and since $f(z) \sqsubseteq z$ we get, by transitivity, $f^{n+1}(\bot) \sqsubseteq z$.

(iii) The proof of this point is like that of Point (ii). □

Kleene Theorem can be generalized to the case of the so called α-chain complete cpo's (which we now define) and *monotonic* functions as follows.

Given a partially ordered set D, for every ordinal α, an *α-chain* of D is a function x from α to D such that for all ordinals $\mu, \nu < \alpha$, if $\mu < \nu < \alpha$ then $x_\mu \sqsubseteq x_\nu$, where for every ordinal $\mu < \alpha$, we have written x_μ, instead of $x(\mu)$. Such an α-chain is also denoted by $\langle x_\beta \,|\, \beta < \alpha \rangle$. Recall that: (i) given any two ordinals μ and α, we have that $\mu < \alpha$ iff $\mu \in \alpha$, and (ii) the ordinal 0 is the empty set.

Let us consider an *α-chain complete cpo* D, that is, a cpo such that *for every ordinal* α (not only the ordinal ω), every α-chain $\langle x_\beta \,|\, \beta < \alpha \rangle$ in D has a least upper bound, denoted $\bigsqcup_{\beta<\alpha} x_\beta$. In particular, the least upper bound of the empty chain (that is, the 0-chain $\langle x_\beta \,|\, \beta < 0 \rangle$) in D is the least element of D, denoted $\bot$.

Let f be any *monotonic* function from D to D. Let $|D|$ be the cardinality of D. Let $|D|^+$ be the least ordinal whose cardinality is strictly greater than $|D|$. Let us consider the following sequence of elements in D defined by transfinite induction up to $|D|^+$:

$f^0 =_{def} \bot$

and for every ordinal $\alpha \in |D|^+$,

$f^\alpha =_{def}$ *if* $\bigsqcup_{\beta<\alpha} f^\beta$ exists in D *then* $f(\bigsqcup_{\beta<\alpha} f^\beta)$ *else* $\bot$

It is shown in [**4**, Theorem 2.2] that there exists a least ordinal $\alpha < |D|^+$ such that: (i) $f^\alpha = f(f^\alpha)$, that is, f^α is a fixpoint of f, (ii) f^α is the minimal fixpoint of f, and (iii) for all $z \in D$, if $f(z) \sqsubseteq z$ then $f^\alpha \sqsubseteq z$, that is, f^α is the minimal prefixpoint of f.

Now let us continue our study of the fixpoint operator *fix* on cpo's. We have the following theorem.

THEOREM 4.8. [*fix* **as a Least Upper Bound**] Given a cpo D with bottom $\bot$. Let $fix =_{def} \lambda f.\, \bigsqcup_{n\in\omega}(f^n(\bot))$ be a function in $[[D \to D] \to D]$. We have that:

$$fix = \bigsqcup_{n\in\omega} \lambda f.(f^n(\bot)). \qquad (FixLub)$$

PROOF. We have to show that:

(A1) $\lambda f.\, \bigsqcup_{n\in\omega}(f^n(\bot)) \sqsubseteq \bigsqcup_{n\in\omega} \lambda f.(f^n(\bot))$ and

(A2) $\lambda f.\, \bigsqcup_{n\in\omega}(f^n(\bot)) \sqsupseteq \bigsqcup_{n\in\omega} \lambda f.(f^n(\bot))$.

Proof of (A1). Since the lub of an ω-chain of functions is a function, (A1) holds iff

(A1*) for every f, $\bigsqcup_{n\in\omega}(f^n(\bot)) \sqsubseteq (\bigsqcup_{n\in\omega} \lambda f.(f^n(\bot)))\, f$.

To show (A1*), it is enough to show that for all $i \in \omega$, $f^i(\bot) \sqsubseteq (\bigsqcup_{n\in\omega} \lambda f.(f^n(\bot)))\, f$. This holds because $f^i(\bot) = \lambda f.(f^i(\bot))\, f \sqsubseteq (\bigsqcup_{n\in\omega} \lambda f.(f^n(\bot)))\, f$. This completes the proof of (A1).

Proof of (A2). By definition of $\bigsqcup$, it is enough to show that for all $i \in \omega$:

(A2*) $\lambda f.\, \bigsqcup_{n\in\omega}(f^n(\bot)) \sqsupseteq \lambda f.(f^i(\bot))$.

To show (A2*), it is enough to show that for all f, for all $i \in \omega$, $\bigsqcup_{n\in\omega}(f^n(\bot)) \sqsupseteq f^i(\bot)$. This holds by definition of $\bigsqcup$. This completes the proof of (A2).

The proof of this theorem can also be done by applying extensionality, that is, by showing that for every function $g \in [D \to D]$, if we apply the left hand side and the right hand side of Equation (*FixLub*) to g, we get the same value.

For the left hand side we get that: $(\lambda f.\,(\bigsqcup_{n\in\omega}(f^n(\bot))))\,g = \bigsqcup_{n\in\omega}(g^n(\bot))$.

For the right hand side, we get that: $(\bigsqcup_{n\in\omega}\lambda f.\,(f^n(\bot)))\,g =$ {by (1) on page 88} $= \bigsqcup_{n\in\omega}((\lambda f.\,(f^n(\bot)))\,g) =$ {by function application} $= \bigsqcup_{n\in\omega}(g^n(\bot))$. □

Thus, by Theorem 4.8 on the previous page, the function *fix* is the least upper bound of the following set of functions in the cpo $[[D \to D] \to D]$:

$F = \{\lambda f.\,\bot,\ \lambda f.\,f(\bot),\ \lambda f.\,f^2(\bot), \dots\}$

where $\bot$ is the bottom element in the cpo D. Actually, we have that:

(i) for all $n \in \omega$, $\lambda f.\,f^n(\bot) \sqsubseteq \lambda f.\,f^{n+1}(\bot)$, that is, F is an ω-chain of functions in $[[D \to D] \to D]$, and

(ii) each function in F is a continuous function.

Here is the proof of Point (i) by induction on n.
(*Basis*) For $n = 0$ we have to show that for all $g \in [D \to D]$, $\bot \sqsubseteq g(\bot)$. This is obvious.
(*Step*) We assume for all $g \in [D \to D]$, $g^n(\bot) \sqsubseteq g^{n+1}(\bot)$ and we show that for all $g \in [D \to D]$, $g^{n+1}(\bot) \sqsubseteq g^{n+2}(\bot)$. This follows from the monotonicity of every function $g \in [D \to D]$, because every function in $[D \to D]$ is continuous.

Here is the proof of Point (ii). Take any chain $g_0 \sqsubseteq g_1 \sqsubseteq \dots g_n \sqsubseteq \dots$ of continuous functions in $[D \to D]$. We have to show that: for all $n \in \omega$, we have that $(\lambda f.\,f^n(\bot))\,(\bigsqcup_{i\in\omega} g_i) = \bigsqcup_{i\in\omega}((\lambda f.f^n(\bot))\,g_i)$. Indeed, take any $n \in \omega$. For the left hand side we have that:

$(\lambda f.\,f^n(\bot))\,(\bigsqcup_{i\in\omega} g_i) = (\bigsqcup_{i\in\omega} g_i)^n(\bot) =$

$=$ {by definition of the application of a limit function and continuity of g_i} $=$

$= \bigsqcup_{i\in\omega}(g_i^n(\bot))$.

For we have that:

$\bigsqcup_{i\in\omega}((\lambda f.f^n(\bot))\,g_i) = \bigsqcup_{i\in\omega}(g_i^n(\bot))$.

This completes the proofs of Points (i) and (ii).

(The above Points (i) and (ii) are also consequences of the results we will show in the following Section 5 starting on page 101.)

Thus, the least upper bound of this chain exists and it is a continuous function in $[[D \to D] \to D]$. We conclude that the function *fix* is a continuous function in $[[D \to D] \to D]$.

Let us consider the equation $d = (f\,d)$, where: (i) d is a variable ranging over the cpo D with bottom $\bot$, (ii) f is a *continuous* function in $[D \to D]$, and (iii) $(f\,d)$ denotes the application of f to d. We assume that d does not occur free in the expression of f. The minimal solution $\widehat{d}$ of the given equation $d = (f\,d)$, is $\mathit{fix}(f)$, that is, the minimal fixpoint of the function f.

In general, given the equation $d = e$, where: (i) d is a variable ranging over the cpo D with bottom $\bot$, (ii) e is an expression with zero or more free occurrences of the variable d, and (iii) e is continuous on its variable d, the minimal solution of that equation is $\mathit{fix}(\lambda d.e)$.

Sometimes, instead of writing $\mathit{fix}(\lambda d.e)$, we will write $\mu d.e$ **[19]**.

By definition of *fix*, we have that:

$\mu d.e = (\lambda d.e)\,(\mu d.e)$ (*FixUnfold*)

THEOREM 4.9. [**Continuity of Fixpoints**] Let us consider a cpo E with bottom $\perp$ and a continuous function $f \in [E \to E]$. Let us also assume that E is a cpo of functions from D to D, for some cpo D. We have that $\mathit{fix}(f)$ is a *continuous* function in $[D \to D]$.

PROOF. We have to show that $(\mathit{fix}\, f)\,(\bigsqcup_{i\in\omega} d_i) = \bigsqcup_{i\in\omega}((\mathit{fix}\, f)\, d_i)$. Indeed, we have that:

$(\mathit{fix}\, f)\,(\bigsqcup_{i\in\omega} d_i) =$ {by Kleene Theorem on page 96} $=$

$= (\bigsqcup_{j\in\omega} f^j(\perp))\,(\bigsqcup_{i\in\omega} d_i) =$ {by definition of the application of a limit function} $=$

$= \bigsqcup_{j\in\omega}\big((f^j(\perp))\,(\bigsqcup_{i\in\omega} d_i)\big) =$ {by continuity of the functions $f^j(\perp)$'s} $=$

$= \bigsqcup_{j\in\omega}\big(\bigsqcup_{i\in\omega}(f^j(\perp))(d_i)\big) =$ {by Lemma 4.4 on page 87} $=$

$= \bigsqcup_{i\in\omega}\big(\bigsqcup_{j\in\omega}(f^j(\perp))(d_i)\big) =$ {by definition of the application of a limit function} $=$

$= \bigsqcup_{i\in\omega}\big((\bigsqcup_{j\in\omega} f^j(\perp))\, d_i\big) =$ {by Kleene Theorem} $=$

$= \bigsqcup_{i\in\omega}((\mathit{fix}\, f)\, d_i).$ □

Bekić Theorem. The following theorem is used for computing the minimal fixpoint of sets of functional equations. Without loss of generality, we will consider the case of two functional equations. The proof of Bekić Theorem can be viewed as an application of the Gauss elimination method for solving a system of linear equations and it is based on the Park Induction rule which we will present on page 105.

THEOREM 4.10. [**Bekić Theorem**] Given the cpo's D and E, let us consider the continuous functions $F \in [[D \times E] \to D]$ and $G \in [[D \times E] \to E]$. The pair $\langle F, G\rangle$ is a continuous function in $[[D \times E] \to [D \times E]]$. The minimal fixpoint $\langle \widehat{f}, \widehat{g}\rangle$ of the function $\langle F, G\rangle$ defined by the equations:

$f = F(f, g)$

$g = G(f, g)$

is the following pair of functions:

$\widehat{f} =_{def} \mu f.\, F(f, \widehat{g})$ (1)

$\widehat{g} =_{def} \mu g.\, G(\mu f.\, F(f, g),\, g)$ (2)

PROOF. First note that by (2) we can express $\widehat{f}$ without referring to $\widehat{g}$, and we have that:

$\widehat{f} =_{def} \mu f.\, F(f,\, \mu g.\, G(\mu f.\, F(f, g),\, g)).$

We have to prove that:

(A) $\langle \widehat{f}, \widehat{g}\rangle$ is a fixpoint of $\langle F, G\rangle$, that is, $\langle \widehat{f}, \widehat{g}\rangle = \langle F, G\rangle(\langle \widehat{f}, \widehat{g}\rangle)$, and

(B) for any other pair $\langle f_0, g_0\rangle$ such that $\langle f_0, g_0\rangle = \langle F, G\rangle(\langle f_0, g_0\rangle)$, we have that $\widehat{f} \sqsubseteq f_0$ and $\widehat{g} \sqsubseteq g_0$.

Proof of (A). By (1) $\widehat{f}$ satisfies the equation $f = F(f, \widehat{g})$ in the unknown f. Thus, we get:

$$\widehat{f} = F(\widehat{f}, \widehat{g}). \tag{3}$$

By (2) $\widehat{g}$ satisfies the equation $g = G(\mu f.\, F(f, g),\, g)$ in the unknown g. Thus, we get:

$$\widehat{g} = G(\mu f.\, F(f, \widehat{g}),\, \widehat{g}). \tag{4}$$

From (1) and (4) we get:

$$\widehat{g} = G(\widehat{f}, \widehat{g}). \tag{5}$$

Equations (3) and (5) show that $\langle \widehat{f}, \widehat{g} \rangle$ is a fixpoint of $\langle F, G \rangle$.

Proof of (B). We first show that $\widehat{g} \sqsubseteq g_0$.

We have that f_0 satisfies the equation $f = F(f, g_0)$ in the unknown f. Thus, by definition of minimal fixpoint, we get:

$$\mu f.F(f, g_0) \sqsubseteq f_0 \tag{6}$$

From (6) by monotonicity of G, we get:

$$G(\mu f.F(f, g_0), g_0) \sqsubseteq G(f_0, g_0) \tag{7}$$

Since g_0 satisfies the equation $g = G(f_0, g)$ in the unknown g, from (7) we get:

$$G(\mu f.F(f, g_0), g_0) \sqsubseteq g_0. \tag{8}$$

Relation (8) tells us that g_0 is a prefixpoint of the function $\lambda g.\, G(\mu f.F(f, g), g)$. By (2) we have that $\widehat{g}$ is the minimal fixpoint of the function $\lambda g.\, G(\mu f.\, F(f, g),\, g)$. Since the minimal fixpoint is also the minimal prefixpoint (see Park Induction on page 105), we get that:

$$\widehat{g} \sqsubseteq g_0. \tag{9}$$

Now we show that $\widehat{f} \sqsubseteq f_0$.

From (9) by monotonicity of F, we get:

$$F(f_0, \widehat{g}) \sqsubseteq F(f_0, g_0) \tag{10}$$

Since f_0 satisfies the equation $f = F(f, g_0)$ in the unknown f, from (10) we get:

$$F(f_0, \widehat{g}) \sqsubseteq f_0. \tag{11}$$

Relation (11) tells us that f_0 is a prefixpoint of the function $\lambda f.\, F(f, \widehat{g})$. By (1) we have that $\widehat{f}$ is the minimal fixpoint of the function $\lambda f.\, F(f, \widehat{g})$. Since the minimal fixpoint is also the minimal prefixpoint (see Park Induction on page 105), we get that:

$$\widehat{f} \sqsubseteq f_0.$$

This completes the proof of Bekić Theorem. □

Since Park Induction holds for complete lattices and monotonic functions (see page 106) and not only for cpo's and continuous functions, we have that Bekić Theorem holds also for complete lattices and monotonic functions.

Exercise 4.11. We leave it to the reader to show a symmetric form of Bekić Theorem where, instead of Equations (1) and (2) of page 99, we have the following two equations:

$$\widehat{f} =_{def} \mu f.\, F(f,\, \mu g.\, G(f, g)) \tag{1*}$$

$$\widehat{g} =_{def} \mu g.\, G(\mu f.\, F(f, g),\, g). \tag{2}$$

□

5. Metalanguage for Denotational Semantics

An expression e denoting an element of the cpo E is said to be *continuous in the variable* x, which may or may not occur in e, ranging over the cpo D iff the function $\lambda x.e$ is a continuous function, that is, $\lambda x \in D.e \in [D \to E]$.

We say that an expression e is *continuous in its variables* iff e is continuous in every variable which occurs in it.

Since every expression is continuous in a variable which does *not* occur in it (because the function $\lambda x.e$ is a constant function if x does not occur in e), we have that an expression e is continuous in its variables iff e is continuous in all variables.

For reasons of simplicity, whenever an expression e is continuous in its variables we will also say that e is *continuous*.

The reader should not confuse the related notions of the continuity of expressions and the continuity of functions.

The following expressions are continuous.

(1) *Variables* are continuous. Indeed, let us consider the variable x. We have that the identity function $\lambda x.x$ is continuous and the constant function $\lambda y.x$ is continuous.

(2) *Constants* are continuous. In particular, $\bot$ (that is, the bottom element of a cpo), *true*, *false*, the constant functions *curry*, *uncurry*, *apply*, $\lfloor\_\rfloor$, *down*, *fix*, the projection functions π_i's, and the injection functions in_i's are all continuous.

The following constructs preserve continuity.

(3) *Product and sum.* Given the expressions e_1, ..., e_k in the cpo's E_1, ..., E_k, respectively, the tuple $(e_1, \ldots, e_k) \in E_1 \times \ldots \times E_k$ (see page 85) is continuous iff for $i = 1, \ldots, k$, we have that e_i is continuous. Indeed, for all variables x,

$\lambda x.(e_1, \ldots, e_k)$ is continuous iff {by Lemma 4.3 on page 87}

iff for $i = 1, \ldots, k$, $\pi_i \circ \lambda x.(e_1, \ldots, e_k)$ is continuous

iff for $i = 1, \ldots, k$, $\lambda x.e_i$ is continuous

iff for $i = 1, \ldots, k$, e_i is continuous.

Similarly, given the expressions e_1, ..., e_k in the cpo's E_1, ..., E_k, respectively, the term $[e_1, \ldots, e_k] \in E_1 + \ldots + E_k$ (see page 92) is continuous iff for $i = 1, \ldots, k$, we have that e_i is continuous.

(4) *Function application.* If the expressions e_1 and e_2 are continuous then the expression $e_1(e_2)$ is continuous.

Note that in the above statement we cannot replace 'if-then' by 'iff' because $e_1(e_2)$ can be continuous even if e_2 is not continuous. Indeed, e_1 can be a constant function which does not depend on its argument.

(5) *Lambda abstraction.* For any variable y and any expression e, if e is continuous then $\lambda y.e$ is continuous. Indeed,
(5.i) if x is y then $\lambda x.(\lambda y.e)$ is continuous because x does not occur free in $\lambda y.e$ (recall that $\lambda y.e$ is equal to $\lambda z.e[z/y]$ for any z different from y), and thus, $\lambda y.e$ is a constant value w.r.t. x. Otherwise,

(5.ii) if x is different from y then

$\lambda x.(\lambda y.e)$ is continuous iff {by Fact 5.1 below}

iff $curry(\lambda(x,y).e)$ is continuous {because *curry* is a constant function}

if $\lambda(x,y).e$ is continuous

iff e is continuous (in the variables x and y).

FACT 5.1. Given any expression e, $\lambda(x,y).e$ is continuous iff $curry(\lambda(x,y).e)$ is continuous.

PROOF. First note that $curry(\lambda(x,y).e)$ is the term $\lambda x.(\lambda y.e)$ and $uncurry(\lambda x.(\lambda y.e))$ is the term $\lambda(x,y).e$.
(*only-if part*) By continuity of *curry* (see Point (2) on page 101) and Point (4) on page 101. (*if part*) This is an easy consequence of the fact that *uncurry* is continuous and we have that $\lambda(x,y).e = uncurry(curry(\lambda(x,y).e))$. □

Using the above results we have also the following facts.

(6) *Lambda abstraction with more than one variable.* If the expression e is continuous also the expression $\lambda(x_1,\ldots,x_k).e$ is continuous.

(7) *Composition.* If the functions $\lambda x.e_1$ and $\lambda x.e_2$ are continuous then the expression $\lambda x.e_1(e_2(x))$ is continuous.

(8) *Let construct.* If the expressions e_1 and e_2 are continuous then $let\, x \Leftarrow e_1 \centerdot e_2$ is continuous. Indeed, $let\, x \Leftarrow e_1 \centerdot e_2$ is equal to $\lfloor \lambda x.e_2 \rfloor(e_1)$ (see page 91).

(9) *Case construct.* If the expressions $e_1,\ldots,e_k$ are continuous then

$$\begin{array}{lll} case\ d\ of & in_1(x_1) \centerdot e_1 & | \\ & \ldots & | \\ & in_k(x_k) \centerdot e_k & \end{array}$$

is continuous. Indeed, the case construct is defined to be $[\lambda x_1.e_1,\ldots,\lambda x_k.e_k](d)$ and (i) the function $\lambda x_1,\ldots,x_k.[x_1,\ldots,x_k]$, (ii) function application, and (iii) lambda abstraction all preserve continuity (see Points (2), (4), and (5) above).

In particular, for all $b \in \{true, false\}$, for all $b' \in \{true, false\}_\perp$, for all $d_1, d_2 \in D$, for all $z \in N_\perp$, the following conditional expressions are all continuous:

(9.1) $cond(b, d_1, d_2)$

(9.2) $b' \to d_1 \,|\, d_2$

(9.3) $Cond(z, d_1, d_2)$

Indeed, they can be constructed using the *sum* construct and the *let* construct as indicated in Point (1) on page 93, Point (2) on page 94, and Point (3) on page 94, respectively.

(10) *Function updating.* Given a *discrete* cpo D, if the function $f : [D \to E]$ is a continuous function then the updated function $f[e/d]$ is also continuous.

Indeed, we can consider the cpo D as the sum of: (i) the singleton $\{d\}$ via the injection function in_1, and (ii) the set $D - \{d\}$ via the injection function in_2. Thus, $\lambda y \in D.\, f[e/d]$ can be defined as follows:

$$\lambda y{\in}D.\ case\ y\ of\ \ in_1(d)\,.\,e\ \ |\\ \qquad\qquad\qquad\qquad in_2(y')\,.\,f(y')$$

In what follow we will write expressions using the constructs listed above in this section. Thus, all those expressions define continuous functions, and those continuous functions have minimal fixpoints.

6. Induction Rules for Proving Properties of Recursive Programs

In order to prove properties of programs whose semantics is defined by minimal fixpoints of continuous functionals, we will use various induction rules which we will present in this section.

First, let us recall that every unary predicate P on a set A can be viewed as a subset of A. Thus, we will feel free to write $x \in P$, instead of writing $P(x)$. In particular, given (i) a predicate P on the set D of a given cpo $(D, \sqsubseteq_D)$ and (ii) an element $d \in D$, we will write $d{\in}P$, instead of $P(d)$.

We need to introduce the following notions of an inclusive predicate and an inclusive subset.

Definition 6.1. [**Inclusive Predicate**] Given the cpo's $D_1, \ldots, D_n$, an n-ary predicate (or a subset) $P \subseteq D_1 \times \ldots \times D_n$, with $n \geq 1$, is said to be *an inclusive predicate on* (or *an inclusive subset of*) $D_1 \times \ldots \times D_n$ iff for all ω-chains $d_0 \sqsubseteq d_1 \sqsubseteq \ldots \in D_1 \times \ldots \times D_n$ we have that *if* $\forall i \in \omega$, $P(d_i)$ holds *then* $P(\bigsqcup_{i\in\omega} d_i)$ holds.

Let us consider the cpo $\Omega =_{def} \{0 \sqsubseteq 1 \sqsubseteq \ldots \sqsubseteq n \sqsubseteq \ldots \sqsubseteq \infty\}$. The inclusive subsets of this cpo are all the finite subsets of Ω and all the subsets S which satisfy the following property: if $\forall n\, \exists k{>}n.\, k{\in}S$ then $\infty{\in}S$ (see also Fact 1.17 on page 316). In particular, $\Omega{-}\{\infty\}$ is *not* an inclusive subset of Ω.

Remark 6.2. [**Non-inclusive Predicate**] Let us consider the discrete cpo N of the integers, that is, $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$. Let us also consider the continuous functional $\tau \in [[N \to N_\perp] \to [N \to N_\perp]]$ defined as follows:

$\tau\,\varphi =_{def} \lambda n.\, cond(n{=}0, \lfloor 1 \rfloor, \lfloor n \rfloor \times_\perp \varphi(n{-}1))$,

Let us also consider the predicate $P \in [N \to N_\perp] \to \{true, false\}$, that is, a subset of $[N \to N_\perp]$, defined as follows:

$P =_{def} \lambda f.\, \exists n \in N.\, n{\geq}0 \wedge f(n){=}\perp$.

Now the predicate P is *not inclusive*. Indeed, for the ω-chain $\{\tau^i(\perp) \mid i \geq 0\}$ we have that:

(1) $\forall i \geq 0$, $P(\tau^i(\perp))$ holds, and

(2) $P(\bigsqcup_{i\in\omega}(\tau^i(\perp)))$ does not hold.

Point (1) can be proved by mathematical induction on i.

(*Basis*) For $i = 0$, $P(\tau^i(\perp)) = P(\perp)$, where $\perp$ is the function $\lambda n.\perp$ in $[N \to N_\perp]$ which always returns $\perp \in N_\perp$.

Now $P(\lambda n.\perp)$ holds because $\exists n \in N.\, n{\geq}0 \wedge (\lambda n.\perp)(n){=}\perp \in N_\perp$.

(*Step*) Assume $P(\tau^i(\perp))$ and show $P(\tau^{i+1}(\perp))$, where $\perp$ is the function $\lambda n.\perp$ in $[N \to N_\perp]$.

We have that $P(\tau^{i+1}(\bot))$ is $\exists n \in N.\, n \geq 0 \wedge cond(n{=}0, \lfloor 1 \rfloor, \lfloor n \rfloor \times_\bot (\tau^i(\bot))(n{-}1)) = \bot$. By induction hypothesis, there exists n' such that $n' \geq 0 \wedge (\tau^i(\bot))(n') = \bot$. Take $m = n'{+}1$. We have that:

$cond(m{=}0, \lfloor 1 \rfloor, \lfloor m \rfloor \times_\bot (\tau^i(\bot))(m{-}1)) = \lfloor m \rfloor \times_\bot (\tau^i(\bot))(n'))$

because $m \neq 0$. Since $(\tau^i(\bot))(n') = \bot$ we get that:

$cond(m{=}0, \lfloor 1 \rfloor, \lfloor m \rfloor \times_\bot (\tau^i(\bot))(m{-}1)) = \bot$.

Point (2) follows from the negation of $P(\bigsqcup_{i\in\omega}(\tau^i(\bot)))$, that is,

$\neg\big(\exists n \in N.\, n \geq 0 \wedge (\bigsqcup_{i\in\omega}(\tau^i(\bot)))(n) = \bot\big)$, which is equivalent to:

$\forall n \in N.\, (n \geq 0 \rightarrow (\bigsqcup_{i\geq 0}(\tau^i(\bot)))(n) \neq \bot)$ (†)

where: (α) the argument $\bot$ in $\tau^i(\bot)$ is the always undefined function $\lambda n \in N.\, \bot$ in $[N \rightarrow N_\bot]$, and (β) the element $\bot$ on the right hand side of $(\bigsqcup_{i\geq 0} \tau^i(\bot))(n) \neq \bot$ is the bottom element of $N_\bot$. Property (†) can be shown by proving, by mathematical induction, that:

$\forall n \in N.\, (n \geq 0 \rightarrow (\tau^{n+1}(\bot))(n) \neq \bot)$.

(*Basis*) $(n{=}0)$. $(\tau^1(\bot))(0) = cond(0{=}0, \lfloor 1 \rfloor, \lfloor n \rfloor \times_\bot \bot(n{-}1)) = \lfloor 1 \rfloor \neq \bot$.

(*Step*) Assume $(\tau^{n+1}(\bot))(n) \neq \bot$ and show $(\tau^{n+2}(\bot))(n{+}1) \neq \bot$. Indeed,

$(\tau^{n+2}(\bot))(n{+}1) = cond(n{+}1{=}0, \lfloor 1 \rfloor, \lfloor n{+}1 \rfloor \times_\bot (\tau^{n+1}(\bot))(n)) =$

$= \{\text{by induction hypothesis}\} =$

$= cond(n{+}1{=}0, \lfloor 1 \rfloor, \lfloor n{+}1 \rfloor \times_\bot \lfloor m \rfloor)$ for some $m \in N =$

$= cond(n{+}1{=}0, \lfloor 1 \rfloor, \lfloor \ell \rfloor)$ for some $\ell \in N$.

Thus, $(\tau^{n+2}(\bot))(n{+}1) \neq \bot$. □

Now let us present the rule called *Scott induction* or *fixpoint induction*.

6.1. Scott Induction.

Let us consider:

(i) a cpo D with bottom element $\bot$,
(ii) a continuous function f in $[D \rightarrow D]$, and
(iii) an inclusive, unary predicate $P \subseteq D$.

Recall that, given a cpo D with bottom element $\bot$, the function *fix* in $[[D \rightarrow D] \rightarrow D]$ is defined as follows: $fix(f) =_{def} \bigsqcup_{i\in\omega} f^i(\bot)$.

The *Scott induction* (also called *fixpoint induction*) rule is the following one.

(Scott Induction, also called Fixpoint Induction. Version 1)

Let P be a unary inclusive predicate.

$$\frac{P(\bot) \qquad \forall d \in D.\, P(d) \rightarrow P(f(d))}{P(fix(f))}$$

Now let us prove the Scott Induction rule.

PROOF. By the assumptions of Scott Induction we have that $P(\bot)$ holds. Since for all $d \in D$, $P(d) \rightarrow P(f(d))$, by induction on i, we have that for all $i \geq 0$, $P(f^i(\bot))$ hold. Now, since: (i) the elements in the set $\{f^i(\bot) \mid i \geq 0\}$ form the ω-chain $\bot \sqsubseteq f(\bot) \sqsubseteq \ldots \sqsubseteq f^i(\bot) \sqsubseteq \ldots$, and (ii) P is an inclusive predicate, we have that $P(\bigsqcup_{i\in\omega} f^i(\bot))$. Since f is a continuous function, by Kleene Theorem, we have that $\bigsqcup_{i\in\omega} f^i(\bot) = \mathit{fix}(f)$, and thus, we get that $P(\mathit{fix}(f))$ holds as stated by the conclusion of the Scott Induction rule. □

Actually, the proof of the Scott Induction rule allows us to get the following deduction rule (see the Mathematical Induction rule on page 59).

(Scott Induction, also called Fixpoint Induction. Version 2)
Let P be a unary inclusive predicate.

$$\frac{P(\bot) \qquad \forall i{\in}\omega.\ P(f^i(\bot)) \rightarrow P(f^{i+1}(\bot))}{P(\mathit{fix}(f))}$$

REMARK 6.3. Note that the premise (1): $\forall d \in D.\, P(d) \rightarrow P(f(d))$ of the Scott Induction rule is more general that the premise (2): $\forall i \in \omega.\, P(f^i(\bot)) \rightarrow P(f^{i+1}(\bot))$ of the Mathematical Induction rule for inclusive predicates because (1) refers to all elements $d{\in}D$, while (2) refers only to the elements of D in the set $\{f^i(\bot) \mid i \in \omega\}$.

Scott induction also holds in the case of a k-ary predicate P, with $k{\geq}1$, subset of the cpo $D_1{\times}\ldots{\times}D_k$ with bottom element $(\bot_{D_1}, \ldots, \bot_{D_k})$, and a continuous function $f \in [[D_1{\times}\ldots{\times}D_k] \rightarrow [D_1{\times}\ldots{\times}D_k]]$. In that case we have the following rule whose proof is left to the reader.

(Scott Induction, also called Fixpoint Induction. Version 3)
Let P be a k-ary inclusive predicate.

$$\frac{P(\bot_{D_1},\ldots,\bot_{D_k}) \qquad \forall d_1{\in}D_1,\ldots,d_k{\in}D_k.\ P(d_1,\ldots,d_k) \rightarrow P(f(d_1,\ldots,d_k))}{P(\mathit{fix}(f))}$$

6.2. Park Induction.

As a consequence of the Scott Induction rule we have the following rule, called *Park Induction*, where D is a cpo with bottom element $\bot$, and f is a continuous function in $[D \rightarrow D]$. For all $d \in D$,

(Park Induction for cpo's and continuous functions)

$$\frac{f(d) \sqsubseteq d}{\mathit{fix}(f) \sqsubseteq d}$$

PROOF. Let us consider the set $P_d =_{def} \{x \in D \mid x \sqsubseteq d\}$. It is an inclusive subset of D because for all ω-chains $d_0 \sqsubseteq d_1 \sqsubseteq \ldots \in D$, if $\forall i \in \omega.\, d_i \in P_d$ then $(\bigsqcup_{i\in\omega} d_i) \in P_d$. Indeed, if $\forall i \in \omega.\, d_i \sqsubseteq d$ then $(\bigsqcup_{i\in\omega} d_i) \sqsubseteq d$.

Now, we have that: (i) $P_d(\bot)$ because $\bot \sqsubseteq d$, and (ii) $\forall x \in D.\, P_d(x) \to P_d(f(x))$ as we now show. Take any $x \in D$, if $x \in P_d$, that is, $x \sqsubseteq d$, then by monotonicity of f, we have that $f(x) \sqsubseteq f(d)$ and, by assumption, $f(d) \sqsubseteq d$. Thus, by transitivity, we get that $f(x) \sqsubseteq d$, that is, $P_d(f(x))$. Finally, by Scott induction we conclude that $\mathit{fix}(f) \in P_d$, that is, $\mathit{fix}(f) \sqsubseteq d$. □

REMARK 6.4. By the Park Induction rule we have that the minimal fixpoint $\mathit{fix}(f)$ of continuous function f is also the minimal prefixpoint of that function. Indeed, any prefixpoint d is such that $f(d) \sqsubseteq d$ (and thus, $\mathit{fix}(f)$ is a prefixpoint of f), and by the Park Induction rule we have that $\mathit{fix}(f) \sqsubseteq d$.

Now we show that Park Induction holds also if we assume that D is a *complete lattice* and f is a *monotonic* function in $D \to D$. Thus, as already mentioned, Bekić Theorem (see page 99) holds both for continuous functions on cpo's and monotonic functions on complete lattices.

Let D be a complete lattice with bottom element $\bot$, and f is a monotonic function in $D \to D$. For all $d \in D$, we have the following rule.

(Park Induction for complete lattices and monotonic functions)

$$\frac{f(d) \sqsubseteq d}{\mathit{Lfix}(f) \sqsubseteq d}$$

where the function *Lfix* from $D \to D$ to D is defined as follows:

$\mathit{Lfix}(f) =_{def} \mathit{glb}\,\{x \mid f(x) \sqsubseteq x\}$.

(The name of the function *Lfix* comes from the fact that, as we will now show, *Lfix* is a fixpoint operator on lattices.) Note that the function *Lfix* is different from the function *fix* defined on page 96, because *Lfix* returns an element of a complete lattice, while *fix* returns an element of a cpo.

As it is the case for the function *fix*, also the function *Lfix* is a minimal fixpoint operator and we have that: $f(\mathit{Lfix}(f)) = \mathit{Lfix}(f)$, as we now show.

Correctness of the Park Induction rule for complete lattices and monotonic functions. Let D be a complete lattice with bottom element $\bot$, and f is a monotonic function in $D \to D$. Let X be $\{x \in D \mid f(x) \sqsubseteq x\}$. By Knaster-Tarski Theorem (see page 81) we have that $\mathit{glb}\, X$ exists. Let m denote $\mathit{glb}\, X$. We show that m is:

(i) a prefixpoint of f,
(ii) a fixpoint of f,
(iii) the least prefixpoint of f, and
(iv) the least fixpoint of f.

Proof of (i). We have to show that $f(m) \sqsubseteq m$. For any $x \in X$, we have that $m \sqsubseteq x$. By monotonicity of f, $f(m) \sqsubseteq f(x)$. Since $x \in X$, we get $f(m) \sqsubseteq f(x) \sqsubseteq x$. Thus, $f(m)$ is a lower bound of X. Hence, since by definition m is the greatest lower bound of X, we get that $f(m) \sqsubseteq m$. Thus, $m \in X$.

Proof of (ii). From Point (i) we have that $f(m) \sqsubseteq m$ and thus, it remains to show that $m \sqsubseteq f(m)$. From Point (i), by monotonicity of f, we have that $f(f(m)) \sqsubseteq f(m)$. That is, $f(m)$ is a prefixpoint of f and, thus, $f(m) \in X$. Since m is a lower bound of X we get that $m \sqsubseteq f(m)$.

Proof of (iii). Consider a prefixpoint $x \in X$. We have to show that $m \sqsubseteq x$. Now, since m is a lower bound of X, we get that $m \sqsubseteq x$.

Proof of (iv). It follows from (ii) and (iii) because every fixpoint is a prefixpoint. □

6.3. McCarthy Induction.

Let us now introduce the *McCarthy induction* rule (also called *unique fixpoint principle*, or *recursion induction*).

Let us consider the two *flat* cpo's $D_\perp$ and $E_\perp$. Their bottom elements are both denoted by $\perp$. Let $D_\perp^n$ denote the cpo $D_\perp \times \ldots \times D_\perp$ with n copies of $D_\perp$. Let us also consider a continuous function $\tau \in [[D_\perp^n \to E_\perp] \to [D_\perp^n \to E_\perp]]$ and let $\mathit{fix}(\tau)$ denote the minimal fixpoint of τ. Let f_1 and f_2 be two continuous functions in $[D_\perp^n \to E_\perp]$. We have the following rule.

(McCarthy Induction, also called Unique Fixpoint Principle, or Recursion Induction)

$$\frac{f_1 = \tau(f_1) \qquad f_2 = \tau(f_2) \qquad \forall x \in S \subseteq D_\perp^n.\ (\mathit{fix}(\tau))(x) \neq \perp}{\forall x \in S \subseteq D_\perp^n.\ f_1(x) = f_2(x)}$$

The proof of this rule is as follows. Take any $x \in S \subseteq D_\perp^n$. Take any fixpoint f of τ. Since $(\mathit{fix}(\tau))(x) \neq \perp$, we have that: $\exists e \in E_\perp.\, e \neq \perp \wedge (\mathit{fix}(\tau))(x) = e$. By Park Induction, since $\tau(f) \sqsubseteq f$, we get $(\mathit{fix}(\tau))(x) \sqsubseteq f(x)$. Thus, $\exists e \in E_\perp.\, e \neq \perp \wedge f(x) = e$ and, as a consequence, $(\mathit{fix}(\tau))(x) = f(x)$. Hence, $\forall x \in S \subseteq D_\perp^n.\, (\mathit{fix}(\tau))(x) = f(x)$. Since f is any fixpoint, we conclude that for every $x \in S \subseteq D_\perp^n$, all fixpoints are equal.

The McCarthy Induction can be used for proving that two functions are equal in a given domain S by: (i) finding a suitable functional τ of which the two functions are fixpoints, and then (ii) showing that the minimal fixpoint of τ is always different from $\perp$ for every element of S.

We will see in action the McCarthy Induction rule in Example 6.19 on page 190.

6.4. Truncation Induction.

Let us introduce the *Truncation induction* rule.

Let us consider a cpo $D_\perp$ with bottom element $\perp$. Let f_1 and f_2 be two continuous functions in $[D_\perp \to D_\perp]$. Let $\mathit{fix}(f)$ denote the minimal fixpoint of a function f. We have the following rule.

(Truncation Induction)

$$\frac{\forall i \geq 0.\ \exists j \geq 0.\ f_1^i(\bot) \sqsubseteq f_2^j(\bot)}{\mathit{fix}(f_1) \sqsubseteq \mathit{fix}(f_2)}$$

The proof of this rule, which we leave to the reader, is based on the fact that if $\forall i \geq 0.\ \exists j \geq 0.\ f_1^i(\bot) \sqsubseteq f_2^j(\bot)$ then $\forall i \geq 0.\ f_1^i(\bot) \sqsubseteq \mathit{fix}(f_2)$. This rule can be used to show the equality of the two minimal fixpoints $\mathit{fix}(f_1)$ and $\mathit{fix}(f_2)$ by showing both $\mathit{fix}(f_1) \sqsubseteq \mathit{fix}(f_2)$ and $\mathit{fix}(f_2) \sqsubseteq \mathit{fix}(f_1)$.

We will see in action the Truncation Induction rule in Example 8.4 on page 113.

6.5. Vuillemin Rule.

Finally, let us present the *Vuillemin rule* which allows us to establish the equality of two minimal fixpoints.

Let us consider a cpo $D_\bot$ with bottom element $\bot$. Let $f = \lambda x.\, e[x]$ be a continuous function in $[D_\bot \to D_\bot]$. Let $\mathit{fix}(f)$ denote the minimal fixpoint of f. We have the following rule.

(Vuillemin rule)

$$\frac{f{=}\lambda x.\, e[x]}{\mathit{fix}(f) = \mathit{fix}(\lambda x.\, e[\mathit{fix}(f)/x])}$$

where by $e[\mathit{fix}(f)/x]$ we denote the expression $e[x]$ where *some* occurrences of x have been replaced by $\mathit{fix}(f)$.

7. Construction of Inclusive Predicates

Now we will provide a few rules for constructing inclusive predicates. When the properties we want to prove are constructed by using these rules, we are sure that they denote inclusive predicates and thus, in particular, these properties can be proved by Scott induction (see Section 6 on page 176).

We have the following theorem.

THEOREM 7.1. [**Construction of Inclusive Predicates**] A predicate of the form $\forall x_1 \ldots x_n.\, P(x_1, \ldots, x_n)$ is inclusive, where $x_1, \ldots, x_n$ are variables ranging over cpo's and P is built out of conjunctions and disjunctions of basic predicates of the form $e_0 \sqsubseteq e_1$ or $e_0 = e_1$, where e_0 and e_1 are expressions constructed from variables and constants (such as *true*, *false*, *apply*, *curry*, $\lfloor\_\rfloor$, *fix*) by using tupling, conditionals (such as *cond*, $b \to e_1 \mid e_2$, and *Cond*), function applications, lambda abstractions, and case constructs.

The proof of this theorem follows from Facts 7.2–7.11 we will now state and prove.

Recall that by Definition 6.1 on page 103, given the n cpo's D_1, ..., D_n, we have that $Q(x_1, \ldots, x_m)$ is an *inclusive predicate on* $D_1 \times \ldots \times D_n$ iff $\{\langle x_1, \ldots, x_m\rangle \mid Q(x_1, \ldots, x_m)\}$ is an inclusive subset of $D_1 \times \ldots \times D_n$.

FACT 7.2. [**Smaller-than and Equality Relation**] Consider a cpo D. We have that: (i) $\{\langle x, y\rangle \mid x \sqsubseteq y\}$ is an inclusive subset of $D \times D$. (ii) $\{\langle x, y\rangle \mid x = y\}$ is an inclusive subset of $D \times D$.

PROOF. (i) Consider the ω-chain $\langle x_0, y_0\rangle \sqsubseteq \langle x_1, y_1\rangle \sqsubseteq \ldots \sqsubseteq \langle x_i, y_i\rangle \sqsubseteq \ldots$ Assume that $\forall i \in \omega$, $\langle x_i, y_i\rangle \in \sqsubseteq$, that is, $\forall i \in \omega$, $x_i \sqsubseteq y_i$. We have to show that $\langle \bigsqcup_{i\in\omega} x_i, \bigsqcup_{i\in\omega} y_i\rangle \in \sqsubseteq$.

Now, since for all $i \in \omega$, $y_i \sqsubseteq (\bigsqcup_{i\in\omega} y_i)$, we get $\forall i \in \omega$, $x_i \sqsubseteq (\bigsqcup_{i\in\omega} y_i)$. Thus, $(\bigsqcup_{i\in\omega} x_i) \sqsubseteq (\bigsqcup_{i\in\omega} y_i)$.

(ii) The proof is similar to that of Point (i). □

FACT 7.3. [**Inverse Image**] Let us consider the cpo's D and E. If Q is an inclusive predicate on E and f is a continuous function in $[D \to E]$. Let P be the *f-inverse-image* of Q, denoted $P = f^{-1}(Q)$, that is, $P =_{def} \{d \mid Q(f(d))\}$. Then P is an inclusive predicate on D.

PROOF. We have that $Q = f(P)$. We want to show that for all ω-chains $d_0 \sqsubseteq d_1 \sqsubseteq \ldots \sqsubseteq d_1 \sqsubseteq \ldots$ of elements in P such that for all $i \geq 0$, $d_i \in P$, we have that $(\bigsqcup_{i\geq 0} d_i) \in P$.

Take an ω-chain $d_0 \sqsubseteq d_1 \sqsubseteq \ldots \sqsubseteq d_i \sqsubseteq \ldots$ in P. Since $P = f^{-1}(Q)$ we have that for all $i \geq 0$, $d_i \in D$. If we map the given chain via the function f we get an ω-chain $e_0 \sqsubseteq e_1 \sqsubseteq \ldots \sqsubseteq e_i \sqsubseteq \ldots$ in E because f is monotonic. For all $i \geq 0$, $e_i \in Q$, because $Q = f(P)$. (Recall that $P = f^{-1}(Q)$, that is, Q is the *f-image* of P.) Since Q is inclusive, we have that $(\bigsqcup_{i\geq 0} e_i) \in Q$, that is, $(\bigsqcup_{i\geq 0} f(d_i)) \in Q$. Since f is continuous, we have that $f(\bigsqcup_{i\geq 0} d_i) \in Q$. Thus, $(\bigsqcup_{i\geq 0} d_i) \in P$, because $Q = f(P)$. □

REMARK 7.4. [**Direct Image**] Given an inclusive set, its image via a continuous function is not necessarily inclusive, that is, given an inclusive set $P \subseteq D$ and a continuous function $f \in [D \to E]$, the set $\{f(x) \mid x \in P\} \subseteq E$ may not be inclusive.

Indeed, let us consider the discrete cpo $\{0, 1, 2, \ldots\}$ of the natural numbers such that $n \sqsubseteq n$, for all $n \geq 0$. The set $\{0, 1, 2, \ldots\}$ with the partial order $\sqsubseteq$ is trivially inclusive. Let us also consider the cpo Ω (with the element ∞) (see page 103). We have that the continuous function f defined as follows:

for all integers $n \geq 0$, $f(n) = n$

maps every element in the set $\{0, 1, 2, \ldots\}$ into $\Omega - \{\infty\}$. The set $\Omega - \{\infty\}$, that is, $\{0 \sqsubseteq 1 \sqsubseteq \ldots \sqsubseteq n \sqsubseteq \ldots\}$, is *not* inclusive.

REMARK 7.5. [**Order-monic Direct Image**] Let us consider the cpo's D and E. If we assume that the continuous function $f \in [D \to E]$ is *order-monic*, that is,

for all $d_1, d_2 \in D$, if $f(d_1) \sqsubseteq f(d_2)$ then $d_1 \sqsubseteq d_2$

we have that the direct images under the function f of inclusive sets are inclusive, that is, if P is an inclusive unary predicate on D then

$Q(y) =_{def} \{y \mid \exists x \in D.\, y = f(x) \wedge P(x)\}$

is an inclusive unary predicate on E.

The injection functions associated with sums and the lifting function $\lfloor\_\rfloor$ are examples of order-monic functions.

Note that, however, the projection functions associated with products are *not* order-monic. For instance, π_1 is not order-monic because, given the discrete cpo $\{0, 1, 2, \ldots\}$, we have that $0 \sqsubseteq 0$ and yet $(0, 1) \not\sqsubseteq (0, 2)$. □

FACT 7.6. [**Substitution**] Let us consider the cpo's $D =_{def} D_1 \times \ldots \times D_k$ and $E =_{def} E_1 \times \ldots \times E_\ell$. Let us assume that $Q(y_1, \ldots, y_\ell)$ is an inclusive predicate on E, where, for $i = 1, \ldots, k$, the variable y_i ranges over E_i. If $\lambda x_1, \ldots, x_k.(e_1, \ldots, e_\ell)$ is a continuous function in $[D \to E]$, where, for $i = 1, \ldots, k$, the expression e_i is continuous on its variables, then $\{(x_1, \ldots, x_k) \mid Q(e_1, \ldots, e_\ell)\}$ is an inclusive subset of D and thus, $Q(e_1, \ldots, e_\ell)$ is an inclusive predicate on D.

PROOF. This fact is a consequence of Fact 7.3 on the preceding page. □

In particular, let us consider the cpo's D and E. If $P(y)$ is an inclusive predicate on E and $\lambda x.t(x)$ is a continuous function from D to E, then $P'(x) \subseteq D$ defined as follows:

for all $x \in D$, $\quad P'(x)$ iff $P(t(x))$

is an inclusive predicate on D.

FACT 7.7. [**Deletion or Addition of Variables**] Let us consider the cpo's D and E. (i) If $P(x, y)$ is an inclusive predicate on $D \times E$, then for all constants $c \in E$, $P(x, c)$ is an inclusive predicate on D. (ii) If $P(x)$ is an inclusive predicate on D, then the binary predicate $R(x, y) \subseteq D \times E$ defined as follows:

for all $x \in D$, for all $y \in E$, $\quad R(x, y)$ iff $P(x)$

is an inclusive predicate on $D \times E$.

PROOF. (i) It follows from Fact 7.6. (ii) It follows from Fact 7.6 by considering $k = 2$ and the continuous function $\lambda x_1, x_2.(e_1, e_2)$ of Fact 7.6 to be the projection function $\pi_1 =_{def} \lambda x_1, x_2 \in D \times E.\, x_1$. □

FACT 7.8. [**Logical Operators**] (i) The predicate *true* and *false* are inclusive. (ii) The *finite* or *infinite conjunction* preserves inclusiveness of predicates. (iii) The *finite disjunction* preserves inclusiveness of predicates. (iv) The infinite disjunction may not preserve inclusiveness of predicates.

PROOF. (i) Given a cpo D, D itself and $\emptyset$ are inclusive subsets of D. Indeed, (i) all ω-chains of elements of D have their limit points in D itself, and (ii) no ω-chain exists in $\emptyset$.
(ii) Every ω-chain of the (finite or infinite) conjunction of the sets S_0, S_1, ... belongs to a set S_i, for some $i \geq 0$.
(iii) Every ω-chain of the finite disjunction of the sets S_0, S_1, ... has its limit point in a set S_i, for some $i \geq 0$.
(iv) Consider the cpo Ω which is of the form: $0 \sqsubseteq 1 \sqsubseteq \ldots \sqsubseteq n \sqsubseteq \ldots \sqsubseteq \infty$ (see also page 103). For each $m \in \omega$, the unary predicate $P(m) = \{m\}$ is inclusive, while $\bigcup_{m \in \omega} P(m)$ is not inclusive because we have that:
(iv.1) $\infty = \bigsqcup_{m \in \omega} m$ and (iv.2) $\infty \notin \bigcup_{m \in \omega} P(m)$. □

FACT 7.9. [**Discrete Cpo**] Any predicate on a discrete cpo is an inclusive predicate, that is, any subset of a discrete cpo is an inclusive subset.

FACT 7.10. [**Product**] Let P_i be an inclusive subset of the cpo D_i, for $i=1,\ldots,k$. We have that $P_1\times\ldots\times P_k$ is an inclusive subset of $D_1\times\ldots\times D_k$.

PROOF. It follows from the facts that: (i) $P_1\times\ldots\times P_k = \pi_1^{-1}(P_1)\cap\ldots\cap\pi_1^{-1}(P_k)$, and (ii) inverse image and intersection preserve inclusiveness. □

A predicate $P(x_1,\ldots,x_k)\subseteq D_1\times\ldots\times D_k$ is said to be *inclusive in each argument separately* iff for $i=1,\ldots,k$, the predicate $P(d_1,\ldots,d_{i-1},x_i,d_{i+1},\ldots,d_k)\subseteq D_i$ obtained by substituting constants for all but the i-th argument, is inclusive. Note that an inclusive predicate $P(x_1,\ldots,x_k)$ is inclusive in each argument separately, but not vice versa. Indeed, here is an example of a predicate that is inclusive in each argument separately and it is not inclusive.

Let us consider that cpo $\Omega =_{def} \{0\sqsubseteq 1\sqsubseteq\ldots\sqsubseteq n\sqsubseteq\ldots\sqsubseteq\infty\}$. The predicate $P(x,y) =_{def} (x=y\wedge x\neq\infty)\subseteq\Omega\times\Omega$ is inclusive in each argument separately, but it is not inclusive. To see that $P(x,y)$ is not inclusive, let us consider the ω-chain:

$\langle 0,0\rangle\sqsubseteq\langle 1,1\rangle\sqsubseteq\ldots\sqsubseteq\langle k,k\rangle\sqsubseteq\ldots\sqsubseteq\langle\infty,\infty\rangle$.

We have that $\forall k\in\omega$, $P(k,k)$ holds, but $P(\infty,\infty)$ does not hold. Thus, $P(x,y)$ is not inclusive.

Now we show that $P(x,y)$ is inclusive both (i) in its first argument and (ii) in its second argument. Indeed, for Point (i) we fix a value for y, say d. The only ω-chain such that $P(x,d)$ holds for each element of the chain, is: $\langle d,d\rangle\sqsubseteq\langle d,d\rangle\sqsubseteq\ldots\sqsubseteq\langle d,d\rangle\sqsubseteq\ldots$, with $d\neq\infty$, whose limit point is $\langle d,d\rangle$. Thus, $P(x,d)$ is inclusive. The proof of Point (ii) is similar to that of Point (i).

FACT 7.11. [**Sum**] Let P_i be an inclusive subset of the cpo D_i, for $i=1,\ldots,k$. We have that $P_1+\ldots+P_k$ is an inclusive subset of $D_1+\ldots+D_k$. Thus, the predicate $Q(y) =_{def} \big(\exists x_1\in D_1.\, y=in_1(x_1)\wedge P_1(x_1)\big)\vee\ldots\vee\big(\exists x_k\in D_k.\, y=in_k(x_k)\wedge P_k(x_k)\big)$ is an inclusive predicate on $D_1+\ldots+D_k$ if for $i=1,\ldots,k$, $P_i(x_i)$ is an inclusive predicate on D_i.

PROOF. It follows from the facts that: (i) $P_1+\ldots+P_k = in_1(P_1)\cup\ldots\cup in_k(P_k)$, (ii) the injections in_i's are order-monic continuous functions and thus, preserve inclusiveness, and (iii) finite union preserves inclusiveness. □

FACT 7.12. [**Function Space**] Consider the two cpo's D and E. Let P be a subset of D and Q be an inclusive subset of E. We have that the following set of functions

$P\to Q =_{def} \{f\mid\forall x\in P.\, f(x)\in Q$ and f is a continuous function in $[D\to E]\}$

is an inclusive subset of $[D\to E]$. Moreover, since: (i) $x\in P$ implies that $x\in D$, (ii) $\forall x\in P.\, f(x)\in Q$ stands for $\forall x.\,(x\in P\Rightarrow f(x)\in Q)$, and (iii) a subset B of a given set A can be viewed as a unary predicate $B(x)$ on the set A (see the beginning of Section 6 on page 103), we can write $P(x)$, instead of $x\in P$, and $Q(f(x))$, instead of $f(x)\in Q$, and we get that the following unary predicate

$R(f) =_{def} \forall x\in D.\,(P(x)\Rightarrow Q(f(x)))$

is an inclusive predicate on $[D\to E]$, whenever $P(x)$ is a unary predicate on D, $Q(y)$ is an inclusive, unary predicate on E, and f is a continuous function in $[D\to E]$.

The subset $P \to Q$ of $[D \to E]$ can be viewed as the unary predicate $R(f)$ such that $R(f)$ holds iff $f \in (P \to Q)$.

PROOF. Consider an ω-chain $f_0 \sqsubseteq f_1 \sqsubseteq \ldots \sqsubseteq f_i \sqsubseteq \ldots$ of functions in $[D \to E]$. Assume that:

$\forall x \in D, \forall i \geq 0, \big(P(x) \Rightarrow Q(f_i(x))\big)$ (α)

We have to show that $\forall x \in D, \big(P(x) \Rightarrow Q(\bigsqcup_{i\in\omega} f_i(x))\big)$.

Take any $x \in D$. Assume $P(x)$. We have to show $Q(\bigsqcup_{i\in\omega} f_i(x))$.
Since Q is an inclusive predicate, it is enough to show that $\forall i \geq 0$, $Q(f_i(x))$. By hypothesis (α), we have $\forall i \geq 0, \big(P(x) \Rightarrow Q(f_i(x))\big)$. Since $P(x)$ holds we have that $\forall i \geq 0$, $Q(f_i(x))$, as desired. □

FACT 7.13. [**Lifting**] Let P be an inclusive subset of the cpo D. We have that $\{\lfloor x \rfloor \mid P(x)\}$ is an inclusive subset of $D_\perp$. Thus, if $P(x)$ is an inclusive predicate on D then $Q(y) =_{def} \exists x \in D.\, y = \lfloor x \rfloor \wedge P(x)$ is an inclusive predicate on $D_\perp$.

PROOF. It follows from the fact that the lifting function $\lambda x.\lfloor x \rfloor$ is an order-monic continuous function (and thus, it preserves inclusiveness) and $Q(y)$ is the direct image under the lifting function of the inclusive set $P(x)$. □

8. Proving Properties of Recursive Programs by Using Induction

In this section we will present the proofs of some program properties by making use of the induction rules presented in Section 6 on page 103. Let us begin by proving the following lemma which will be useful in Example 8.3 on the facing page.

LEMMA 8.1. [**Distributivity of Strict Functions over the Conditional Construct**] Let us consider two cpo's D and E, both with bottom element. Let $b \in \{\mathit{true}, \mathit{false}\}_\perp$, $d_1, d_2 \in D$, and $h \in [D \to E]$. We have that:

(i) $h(b \to d_1 \,|\, d_2) \sqsupseteq b \to h(d_1) \,|\, h(d_2)$, and

(ii) if $h(\perp) = \perp \in E$ then $h(b \to d_1 \,|\, d_2) = b \to h(d_1) \,|\, h(d_2)$.

PROOF. (i) is immediate by considering the three cases $b = \perp$, $\lfloor \mathit{true} \rfloor$, and $\lfloor \mathit{false} \rfloor$. Also the proof of (ii) is by cases. If $b = \perp$, $\lfloor \mathit{true} \rfloor$, and $\lfloor \mathit{false} \rfloor$, both sides are equal to $\perp \in E$, $h(d_1)$, and $h(d_2)$, respectively. □

EXAMPLE 8.2. Let us consider the flat cpo $N_\perp$ of the integer numbers with bottom element $\perp$ and the following continuous functional $\tau \in [[N_\perp \to N_\perp] \to [N_\perp \to N_\perp]]$:

$\tau\, \varphi = \lambda n.\, ((\varphi\, n) +_\perp \lfloor 3 \rfloor)$

We want to show that $\mathit{fix}(\tau) = \lambda n.\perp \in [N_\perp \to N_\perp]$. We have that:

$\tau^0(\perp) = \lambda n.\perp$

$\tau^1(\perp) = \lambda m.(((\lambda n.\perp)\, m) +_\perp \lfloor 3 \rfloor) = \lambda m.(\perp +_\perp \lfloor 3 \rfloor) = \lambda m.\perp$.

Since $\tau^0(\perp) = \tau^1(\perp)$, we get that: $\mathit{fix}(\tau) = \tau^1(\perp) = \lambda n.\perp \in [N_\perp \to N_\perp]$.

This result can also be proved by fixpoint induction by using the inclusive predicate $P(f) =_{def} (f = \lambda n.\perp)$, which is a subset of $[N_\perp \to N_\perp]$. Indeed, we have that: (i) $P(\perp)$ holds, and (ii) for all $f \in [N_\perp \to N_\perp]$ if $P(f)$ holds then $P(\tau(f))$ holds. Indeed,

$P(\tau(f)) = \lambda n.\,(f(n) +_\perp \lfloor 3 \rfloor) =$
$= \{$by the induction hypothesis $P(f)\} =$
$= \lambda n.\,(\perp +_\perp \lfloor 3 \rfloor) = \perp.$

In Chapter 6 on page 173 we will see that the above functional τ can be associated with the declaration:

$f(x) = f(x) + 3$

where the evaluation of the function f is done according to the call-by-name regime. □

EXAMPLE 8.3. Let us consider a cpo $D_\perp$ with bottom element $\perp$ and the following continuous functional $\tau \in [[D_\perp \to D_\perp] \to [D_\perp \to D_\perp]]$:

$\tau\,\varphi = \lambda x.\, p(x) \to x \,|\, \varphi(\varphi(h\,x))$

where: $p \in [D_\perp \to T_\perp]$ is strict (that is, $p(\perp) = \perp$) and $h \in [D_\perp \to D_\perp]$. (The cpo $T_\perp$ is defined in Figure 1 on page 83.) Let $\widehat{f}$ be $\mathit{fix}(\tau)$. We want to show that:

$\forall x \in D.\, \widehat{f}(\widehat{f}(x)) = \widehat{f}(x).$

We will prove this property by fixpoint induction by considering the predicate

$P(g) =_{def} \forall x \in D_\perp.\, \widehat{f}(g(x)) = g(x).$

We have that $P(g)$ is an inclusive predicate on $[D_\perp \to D_\perp]$. First we have to show $P(\lambda n \in D_\perp.\perp)$. This follows from the fact that for all $x \in D_\perp$, (i) $g(x) = \perp$, and (ii) $\widehat{f}(g(x)) = \widehat{f}(\perp) = \{$since $\widehat{f}$ satisfies (†) and $p(\perp) = \perp\} = \perp$.
Then we have to show that:

$\forall f \in [D_\perp \to D_\perp]$ if $\forall x \in D_\perp.\, \widehat{f}(f(x)) = f(x)$ then $\forall x \in D_\perp.\, \widehat{f}((\tau f)(x)) = (\tau f)(x)$.

Let us assume that $\forall x \in D_\perp.\, \widehat{f}(f(x)) = f(x)$ and let us take any $x \in D_\perp$. Now,

$\widehat{f}((\tau f)(x)) = \{$by definition of $\tau f\} =$
$= \widehat{f}(p(x) \to x \,|\, f(f(h\,x))) = \{p(\perp) = \perp$ and Lemma 8.1 on the facing page$\} =$
$= p(x) \to x \,|\, \widehat{f}(f(f(h\,x))) = \{$by induction hypothesis$\} =$
$= p(x) \to x \,|\, f(f(h\,x)).$ (†)

Since $(\tau f)(x) = p(x) \to x \,|\, f(f(h\,x))$, from (†) we get that $\widehat{f}((\tau f)(x)) = (\tau f)(x)$. □

In the following example we will see in action the Truncation Induction rule (see page 107).

EXAMPLE 8.4. **[Milner Functionals]** Let us consider the cpo $D_\perp$ with bottom element $\perp$ and the following two continuous functionals $\tau_1, \tau_2 \in [[D_\perp \to D_\perp] \to [D_\perp \to D_\perp]]$:

$f = \tau_1(f)$
$g = \tau_2(g)$

Let $\widehat{f}$ be $\mathit{fix}(\tau_1)$ and $\widehat{g}$ be $\mathit{fix}(\tau_2)$. Assume that:

$\tau_1(\perp) = \tau_2(\perp)$, where $\perp \in [D_\perp \to D_\perp]$ (†1)
for all $h \in [D_\perp \to D_\perp]$, $\tau_1(\tau_2(h)) = \tau_2(\tau_2(\tau_1(h)))$ (†2)

By truncation induction we can show that $\widehat{f} = \widehat{g}$. Indeed, we have that:

for all $n \geq 0$, $\tau_1^n(\perp) = \tau_2^{2^n-1}(\perp)$.

The proof of this fact is by mathematical induction.
(*Basis*) Obvious.
(*Step*) Assume $\tau_1^n(\bot) = \tau_2^{2^n-1}(\bot)$. We have to show that $\tau_1^{n+1}(\bot) = \tau_2^{2^{n+1}-1}(\bot)$.
First note that for all $m \geq 0$, $\tau_1(\tau_2^m(\bot)) = \tau_2^{2m}(\tau_1(\bot))$. (†3)
This can easily be shown by induction on $m\,(\geq 0)$ by using (†2). Then,
$\tau_1^{n+1}(\bot) = \tau_1(\tau_1^n(\bot)) = \{\text{by induction hypothesis}\} =$
$= \tau_1(\tau_2^{2^n-1}(\bot)) = \{\text{by (†3)}\} =$
$= \tau_2^{2(2^n-1)}(\tau_1(\bot)) = \{\text{by (†1)}\} =$
$= \tau_2^{2^{n+1}-1}(\bot)$.
This completes the proof that $\widehat{f} = \widehat{g}$.

A proof of $\widehat{f} = \widehat{g}$ can also be done by Scott induction as follows (thanks to Robin Milner). It is enough to show the following three properties $P_1(\widehat{g})$, $P_2(\widehat{f})$, and $P_3(\widehat{g})$. As a consequence of $P_2(\widehat{f})$ and $P_3(\widehat{g})$, we will then get $\widehat{f} = \widehat{g}$.

(1) $P_1(\widehat{g}) =_{def} \tau_1(\widehat{g}) \sqsubseteq \widehat{g}$, where $P_1 =_{def} \lambda h.\, \tau_1(h) \sqsubseteq \widehat{g}$,
(2) $P_2(\widehat{f}) =_{def} \widehat{f} \sqsubseteq \widehat{g}$, where $P_2 =_{def} \lambda h.\, h \sqsubseteq \widehat{g}$, and
(3) $P_3(\widehat{g}) =_{def} \widehat{g} \sqsubseteq \tau_2(\widehat{g}) \wedge \tau_2(\widehat{g}) \sqsubseteq \widehat{f} \wedge \tau_2(\widehat{g}) \sqsubseteq \tau_1(\widehat{g})$,
where $P_3 =_{def} \lambda h.\, h \sqsubseteq \tau_2(h) \wedge \tau_2(h) \sqsubseteq \widehat{f} \wedge \tau_2(h) \sqsubseteq \tau_1(h)$.

Proof of $P_1(\widehat{g})$. By Scott induction we have to prove; (i) $P_1(\bot)$, and (ii) for all h, $P_1(h) \to P_1(\tau_2(h))$.
(*Basis*) We have that: (i) $\tau_1(\bot) = \tau_2(\bot)$ by (†1), and (ii) $\tau_2(\bot) \sqsubseteq \widehat{g}$, because $\widehat{g}$=*fix*(τ_2). From (i) and (ii), by transitivity, we get: $\tau_1(\bot) \sqsubseteq \widehat{g}$.
(*Step*) Take any $h \in [D_\bot \to D_\bot]$. Assume $\tau_1(h) \sqsubseteq \widehat{g}$ and show $\tau_1(\tau_2(h)) \sqsubseteq \widehat{g}$. Indeed,
$\tau_1(\tau_2(h)) = \{\text{by (†2)}\} = \tau_2(\tau_2(\tau_1(h))) \sqsubseteq \{\text{by induction hypothesis}\} \sqsubseteq$
$\sqsubseteq \tau_2(\tau_2(\widehat{g})) = \{\widehat{g}$=*fix*$(\tau_2)\} = \widehat{g}$.

Proof of $P_2(\widehat{f})$. By Scott induction we have to prove; (i) $P_2(\bot)$, and (ii) for all h, $P_2(h) \to P_2(\tau_1(h))$.
(*Basis*) Obviously, $\bot \sqsubseteq \widehat{g}$.
(*Step*) Take any $h \in [D_\bot \to D_\bot]$. Assume $h \sqsubseteq \widehat{g}$ and show $\tau_1(h) \sqsubseteq \widehat{g}$. Indeed, by monotonicity of τ_1, from $h \sqsubseteq \widehat{g}$, we get:
$\tau_1(h) \sqsubseteq \tau_1(\widehat{g}) = \{\widehat{g}$=*fix*$(\tau_2)\} = \tau_1(\tau_2(\widehat{g})) = \{\text{by (†2)}\} =$
$= \tau_2(\tau_2(\tau_1(\widehat{g}))) \sqsubseteq \{\text{by } P_1(\widehat{g})\} \sqsubseteq \tau_2(\tau_2(\widehat{g})) = \{\widehat{g}$=*fix*$(\tau_2)\} = \widehat{g}$.

Proof of $P_3(\widehat{g})$. By Scott induction we have to prove: (i) $P_3(\bot)$, and (ii) for all h, $P_3(h) \to P_3(\tau_2(h))$.
(*Basis*) We have to show that:
$\bot \sqsubseteq \tau_2(\bot) \wedge \tau_2(\bot) \sqsubseteq \widehat{f} \wedge \tau_2(\bot) \sqsubseteq \tau_1(\bot)$.
Now, $\bot \sqsubseteq \tau_2(\bot)$ is obvious. The rest follows from (†1) (that is, $\tau_2(\bot) = \tau_1(\bot)$) and $\tau_1(\bot) \sqsubseteq \widehat{f}$.
(*Step*) Take any $h \in [D_\bot \to D_\bot]$. Assume:
(H1) $h \sqsubseteq \tau_2(h)$,

(H2) $\tau_2(h) \sqsubseteq \widehat{f}$, and
(H3) $\tau_2(h) \sqsubseteq \tau_1(h)$,
and show:
(T1) $\tau_2(h) \sqsubseteq \tau_2(\tau_2(h))$,
(T2) $\tau_2(\tau_2(h)) \sqsubseteq \widehat{f}$, and
(T3) $\tau_2(\tau_2(h)) \sqsubseteq \tau_1(\tau_2(h))$.
Proof of (T1). By monotonicity of τ_2, from H1 we get (T1).
Proof of (T2). By monotonicity of τ_2, from H1 we get:
$\tau_2(\tau_2(h)) \sqsubseteq \tau_2(\tau_2(\tau_2(h))) \sqsubseteq$ {by H3} $\sqsubseteq$
$\quad \sqsubseteq \tau_2(\tau_2(\tau_1(h))) =$ {by (†2)} $= \tau_1(\tau_2(h)) \sqsubseteq$ {by H2} $\sqsubseteq$
$\quad \sqsubseteq \tau_1(\widehat{f}) = \{\widehat{f} = \mathit{fix}(\tau_1)\} = \widehat{f}$.
Proof of (T3). By monotonicity of τ_2, from H1 we get:
$\tau_2(\tau_2(h)) \sqsubseteq \tau_2(\tau_2(\tau_2(h))) \sqsubseteq$ {by H3} $\sqsubseteq$
$\quad \sqsubseteq \tau_2(\tau_2(\tau_1(h))) =$ {by (†2)} $= \tau_1(\tau_2(h))$. □

CHAPTER 5

Syntax and Semantics of Imperative Languages

In this chapter we consider a simple imperative language, called IMP, and we define its operational semantics in Section 2, its denotational semantics in Section 3, and its axiomatic semantics using Hoare triples in Section 4. In Section 5 we prove the correctness of some simple imperative programs.

In Section 6 we consider the language of the so called *annotated commands* which are commands which are associated with the proof of their correctness. In Section 7 we study the relationship among various semantics definitions of the while-do command.

Finally, in Sections 8 and 9 we provide the operational semantics of, respectively, a simple nondeterministic language and a simple parallel language.

1. Syntax of the Imperative Language IMP

In this section we introduce the syntax of a Pascal-like language, called IMP. In this language we can specify deterministic computations as stated in Fact 2.2 on page 120.

In the language IMP we consider the following syntactic domains.

(i) The set of *integers* $N = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. The variables ranging over N are: $n, m, \ldots$

Note that, as in [**19**], here by N we do *not* denote the set of natural numbers. Usually, in other textbooks the set of integers is denoted by $\mathbb{Z}$.

(ii) The set **Loc** of *locations* (or *memory addresses*). The variables ranging over **Loc** are: $X, Y, \ldots$

(iii) The set $\{+, -, \times\}$ of the *arithmetic operators*. **op** ranges over $\mathbf{Aop} = \{+, -, \times\}$. The division operation is not included in **Aop** because we want to make sure that the result of every operation is an element of N.

(iv) The set **Aexp** of the *arithmetic expressions* which is defined as follows:

$$a ::= n \mid X \mid a_1 \ \mathbf{op} \ a_2$$

where $a, a_1, a_2 \in \mathbf{Aexp}$, $n \in N$, and $X \in \mathbf{Loc}$.

(v) The set $\{<, \leq, =, \geq, >\}$ of the *relational operators* (between arithmetic expressions). **rop** ranges over $\mathbf{Rop} = \{<, \leq, =, \geq, >\}$.

(vi) The set $\{\wedge, \vee, \Rightarrow\}$ of the *binary boolean operators* (between boolean expressions). **bop** ranges over $\mathbf{Bop} = \{\wedge, \vee, \Rightarrow\}$.

(vii) The set **Bexp** of the *boolean expressions* which is defined as follows:

$$b ::= \mathbf{true} \mid \mathbf{false} \mid a_1 \ \mathbf{rop} \ a_2 \mid \neg b \mid b_1 \ \mathbf{bop} \ b_2$$

where $b, b_1, b_2 \in \mathbf{Bexp}$ and $a_1, a_2 \in \mathbf{Aexp}$.

(viii) The set **Com** of the *commands* which is defined as follows:

$$c ::= \textbf{skip} \mid X := a \mid c_1; c_2 \mid \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{while } b \textbf{ do } c_0$$

where $c, c_0, c_1, c_2 \in \textbf{Com}$, $a \in \textbf{Aexp}$, and $b \in \textbf{Bexp}$.

2. Operational Semantics of the Imperative Language IMP

In this section we introduce the operational semantics of our Pascal-like language IMP. The operational semantics specifies:
- the evaluation of arithmetical expressions,
- the evaluation of boolean expressions, and
- the execution of commands.

In order to specify the operational semantics we need the notion of a state.
A *state* σ is a function from **Loc** to N. The set of all states is called *State*.

2.1. Operational Semantics of Arithmetic Expressions.

The evaluation of arithmetic expressions is given as a ternary relation which is a subset of $\textbf{Aexp} \times \mathit{State} \times N$. A triple $\langle a, \sigma, n\rangle$ in $\textbf{Aexp} \times \mathit{State} \times N$ is written as $\langle a, \sigma\rangle \to n$ and specifies that the arithmetic expression a in the state σ evaluates to the integer n. The axioms and the inference rule defining the evaluation of arithmetic expressions are as follows.

$$\langle n, \sigma\rangle \to n$$

$$\langle X, \sigma\rangle \to \sigma(X)$$

$$\frac{\langle a_1,\sigma\rangle \to n_1 \qquad \langle a_2,\sigma\rangle \to n_2}{\langle a_1 \textbf{ op } a_2, \sigma\rangle \to n_1 \mathit{ op } n_2}$$

where *op* is the semantic arithmetic operation corresponding to $\textbf{op} \in \{+, -, \times\}$, that is, *op*: $(N \times N) \to N$ performs the usual arithmetic operations of *sum*, *minus*, and *times*, respectively.

We have that the evaluation of the arithmetic expressions according to the above rules is deterministic, that is, for all $a \in \textbf{Aexp}$, for all $\sigma \in \mathit{State}$, for all $n, n' \in N$, if $\langle a, \sigma\rangle \to n$ and $\langle a, \sigma\rangle \to n'$ then $n = n'$. The proof of this fact is similar to that of Fact 2.2 on page 120.

2.2. Operational Semantics of Boolean Expressions.

The evaluation of boolean expressions is given as ternary relation which is a subset of $\textbf{Bexp} \times \mathit{State} \times \{\textbf{true}, \textbf{false}\}$. A triple $\langle b, \sigma, t\rangle$ in $\textbf{Bexp} \times \mathit{State} \times \{\textbf{true}, \textbf{false}\}$ is written as $\langle b, \sigma\rangle \to t$ and specifies that the boolean expression b in the state σ evaluates to the boolean value t. The axioms and the inference rules defining the evaluation of boolean expressions are as follows.

$$\langle \textbf{true}, \sigma\rangle \to \textbf{true}$$

$$\langle \textbf{false}, \sigma\rangle \to \textbf{false}$$

$$\frac{\langle a_1,\sigma\rangle \to n_1 \quad \langle a_2,\sigma\rangle \to n_2}{\langle a_1 \textbf{ rop } a_2, \sigma\rangle \to n_1 \ \mathit{rop}\ n_2}$$

where *rop* is the semantic relational operation corresponding to $\mathbf{rop} \in \{<, \leq, =, \geq, >\}$, that is, $rop\colon (\mathbf{Aexp} \times \mathbf{Aexp}) \to \{\mathbf{true}, \mathbf{false}\}$ performs the usual relational operations of *less-than* ($<$), *less-than-or-equal-to* ($\leq$), *equal-to* ($=$), *greater-than-or-equal-to* ($\geq$), and *greater-than* ($>$), respectively.

$$\frac{\langle b,\sigma\rangle \to \mathbf{true}}{\langle \neg b,\sigma\rangle \to \mathbf{false}} \qquad \frac{\langle b,\sigma\rangle \to \mathbf{false}}{\langle \neg b,\sigma\rangle \to \mathbf{true}}$$

$$\frac{\langle b_1,\sigma\rangle \to t_1 \quad \langle b_2,\sigma\rangle \to t_2}{\langle b_1 \,\mathbf{bop}\, b_2, \sigma\rangle \to t_1 \; bop \; t_2}$$

where *bop* is the semantic boolean operation corresponding to $\mathbf{bop} \in \{\wedge, \vee, \Rightarrow\}$, that is, $bop\colon (\{\mathbf{true}, \mathbf{false}\} \times \{\mathbf{true}, \mathbf{false}\}) \to \{\mathbf{true}, \mathbf{false}\}$ performs the usual boolean operations of *and* ($\wedge$), *or* ($\vee$), and *implies* ($\Rightarrow$), respectively.

We have that the evaluation of the boolean expressions according to the above rules is deterministic, that is, for all $b \in \mathbf{Bexp}$, for all $\sigma \in State$, for all $t, t' \in \{\mathbf{true}, \mathbf{false}\}$, if $\langle b, \sigma\rangle \to t$ and $\langle b, \sigma\rangle \to t'$ then $t = t'$. The proof of this fact is similar to that of Fact 2.2 on the next page.

2.3. Operational Semantics of Commands.

The execution of commands is given as a ternary relation which is a subset of $\mathbf{Com} \times State \times State$. A triple in $\mathbf{Com} \times State \times State$ is written as $\langle c, \sigma\rangle \to \sigma'$ and specifies that the command c from the state σ produces the new state σ'. The axiom and the inference rules defining the execution of commands are as follows.

$$\langle \mathbf{skip}, \sigma\rangle \;\to\; \sigma$$

$$\frac{\langle a,\sigma\rangle \to n}{\langle X{:=}a, \sigma\rangle \to \sigma[n/X]}$$

$$\frac{\langle c_1,\sigma\rangle \to \sigma_1 \quad \langle c_2,\sigma_1\rangle \to \sigma_2}{\langle c_1;c_2, \sigma\rangle \to \sigma_2}$$

$$\frac{\langle b,\sigma\rangle \to \mathbf{true} \quad \langle c_1,\sigma\rangle \to \sigma_1}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \sigma\rangle \to \sigma_1} \qquad \frac{\langle b,\sigma\rangle \to \mathbf{false} \quad \langle c_2,\sigma\rangle \to \sigma_2}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \sigma\rangle \to \sigma_2}$$

$$\frac{\langle b,\sigma\rangle \to \mathbf{false}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \to \sigma} \qquad \frac{\langle b,\sigma\rangle \to \mathbf{true} \quad \langle c,\sigma\rangle \to \sigma_1 \quad \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma_1\rangle \to \sigma_*}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \to \sigma_*}$$

NOTE 2.1. The ternary operator $\langle \_ , \_ \rangle \to \_$ is overloaded and it is used for the operational semantics of arithmetic expressions, boolean expressions, and commands.

Here is Euclid's algorithm for computing the greatest common divisor of M and N using commands (we assume that $M > 0$ and $N > 0$):

$\{n = N > 0 \wedge m = M > 0\}$
while $m \neq n$ **do** **if** $m > n$ **then** $m := m{-}n$ **else** $n := n{-}m$ **od**
$\{gcd(N, M) = n\}$

Similarly to the case of the evaluation of the arithmetic expressions (see Section 2.1 on page 118) and boolean expressions (see Section 2.2 on page 118) we have that, as stated by the following fact, the evaluation of commands according to the above rules is deterministic.

FACT 2.2. [**Determinism of the Operational Semantics of Commands**] The operational semantics of commands defines a function, that is,

$\forall c \in \mathbf{Com}, \forall \sigma_0, \sigma_1, \sigma_1' \in \mathit{State}, \big(\langle c, \sigma_0\rangle \to \sigma_1 \wedge \langle c, \sigma_0\rangle \to \sigma_1'\big) \Rightarrow \sigma_1 = \sigma_1'$.

PROOF. We will write $\dfrac{d}{\langle c, \sigma_0\rangle \to \sigma_1}$, instead of writing $d \Vdash_R \langle c, \sigma_0\rangle \to \sigma_1$ (see page 65), to mean that, given the set of the derivation rules for the operational semantics of commands (see Section 2.3 on the preceding page), d is a derivation of $\langle c, \sigma_0\rangle \to \sigma_1$. Let us consider the following property $P(d)$ of a given derivation d:

$$\forall c \in \mathbf{Com}, \forall \sigma_0, \sigma_1, \sigma_1' \in \mathit{State}, \Big(\frac{d}{\langle c, \sigma_0\rangle \to \sigma_1} \wedge \exists \widetilde{d}\, \frac{\widetilde{d}}{\langle c, \sigma_0\rangle \to \sigma_1'}\Big) \Rightarrow \sigma_1 = \sigma_1'.$$

We do the proof by well-founded induction on derivations: (i) we assume that for all subderivations $d' \prec^+ d$, we have that $P(d')$, that is,

$$\forall c \in \mathbf{Com}, \forall \sigma_0, \sigma_1, \sigma_1' \in \mathit{State}, \Big(\frac{d'}{\langle c, \sigma_0\rangle \to \sigma_1} \wedge \exists \widetilde{d}\, \frac{\widetilde{d}}{\langle c, \sigma_0\rangle \to \sigma_1'}\Big) \Rightarrow \sigma_1 = \sigma_1'$$

and (ii) we have to show $P(d)$.

In order to show $P(d)$, we consider a fixed, generic command c and the fixed, generic states σ_0, σ_1, and σ_1'. We assume:

$$\frac{d}{\langle c, \sigma_0\rangle \to \sigma_1} \wedge \exists \widetilde{d}\, \frac{\widetilde{d}}{\langle c, \sigma_0\rangle \to \sigma_1'} \qquad (\mathit{Hyp})$$

and we have to show that $\sigma_1 = \sigma_1'$. We reason by cases on the structure of c.

Case 1. $c = \mathbf{skip}$. In this case (Hyp) is $\dfrac{}{\langle \mathbf{skip}, \sigma_0\rangle \to \sigma_1} \wedge \exists \widetilde{d}\, \dfrac{\widetilde{d}}{\langle \mathbf{skip}, \sigma_0\rangle \to \sigma_1'}$. Since both σ_1 and σ_1' are equal to σ_0, we get $\sigma_1 = \sigma_1'$.

Case 2. $c = X := a$. In this case (Hyp) is $\dfrac{\dfrac{d_1}{\langle a, \sigma_0\rangle \to m}}{\langle c, \sigma_0\rangle \to \sigma_1} \wedge \exists \widetilde{d}_1\, \dfrac{\dfrac{\widetilde{d}_1}{\langle a, \sigma_0\rangle \to \widetilde{m}}}{\langle c, \sigma_0\rangle \to \sigma_1'}$, where $\sigma_1 = \sigma_0[m/X]$ and $\sigma_1' = \sigma_0[\widetilde{m}/X]$. Since the evaluation of the arithmetic expressions is deterministic (see Section 2.1 on page 118), we get $m = \widetilde{m}$ and, thus, $\sigma_1 = \sigma_1'$.

Case 3. $c = c_1; c_2$. In this case (Hyp) is:

$$\frac{\dfrac{d_1}{\langle c_1, \sigma_0\rangle \to \sigma'} \quad \dfrac{d_2}{\langle c_2, \sigma'\rangle \to \sigma_1}}{\langle c_1; c_2, \sigma_0\rangle \to \sigma_1} \wedge \exists \widetilde{d}_1, \widetilde{d}_2\, \frac{\dfrac{\widetilde{d}_1}{\langle c_1, \sigma_0\rangle \to \sigma^*} \quad \dfrac{\widetilde{d}_2}{\langle c_2, \sigma^*\rangle \to \sigma_1'}}{\langle c_1; c_2, \sigma_0\rangle \to \sigma_1'}$$

and the derivation d is such that $\dfrac{d}{\langle c_1; c_2, \sigma_0\rangle \to \sigma_1}$. By induction, since $d_1 \prec^+ d$ and $d_2 \prec^+ d$, we have $P(d_1)$ and $P(d_2)$. By $P(d_1)$ we have that:

$$\forall c_1, \sigma_0, \sigma', \sigma^*,\ \frac{d_1}{\langle c_1, \sigma_0\rangle \to \sigma'} \wedge \exists \widetilde{d_1}\ \frac{\widetilde{d_1}}{\langle c_1, \sigma_0\rangle \to \sigma^*} \Rightarrow \sigma' = \sigma^*.$$

Thus, by (*Hyp*) we get $\sigma' = \sigma^*$. Analogously by $P(d_2)$ we have that (note that we have written σ^*, instead of σ'):

$$\forall c_2, \sigma^*, \sigma_1, \sigma_1',\ \frac{d_2}{\langle c_2, \sigma^*\rangle \to \sigma_1} \wedge \exists \widetilde{d_2}\ \frac{\widetilde{d_2}}{\langle c_2, \sigma^*\rangle \to \sigma_1'} \Rightarrow \sigma_1 = \sigma_1'$$

which by (*Hyp*) gives us $\sigma_1 = \sigma_1'$.

Case 4. $c =$ **if** b **then** c_1 **else** c_2. Let us consider the case in which the evaluation of b gives us **true**, that is, $\langle b, \sigma_0\rangle \to$ **true**. Recall that the evaluation of the boolean expressions is deterministic (see Section 2.2 on page 118). In this case (*Hyp*) is:

$$\frac{\dfrac{d_1}{\langle b, \sigma_0\rangle \to \textbf{true}}\quad \dfrac{d_2}{\langle c_1, \sigma_0\rangle \to \sigma_1}}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \sigma_0\rangle \to \sigma_1} \wedge \exists \widetilde{d_1},\ \widetilde{d_2}\ \frac{\dfrac{\widetilde{d_1}}{\langle b, \sigma_0\rangle \to \textbf{true}}\quad \dfrac{\widetilde{d_2}}{\langle c_1, \sigma_0\rangle \to \sigma_1'}}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \sigma_0\rangle \to \sigma_1'}$$

and the derivation d is such that $\dfrac{d}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \sigma_0\rangle \to \sigma_1}$. By induction, since $d_2 \prec^+ d$, we have $P(d_2)$, that is, $\sigma_1 = \sigma_1'$.

Analogously when $\langle b, \sigma_0\rangle \to$ **false**.

Case 5. $c =$ **while** b **do** c_1. Let us consider the case $\langle b, \sigma_0\rangle \to$ **false**. In this case (*Hyp*) is:

$$\frac{\dfrac{d_1}{\langle b, \sigma_0\rangle \to \textbf{false}}}{\langle \textbf{while } b \textbf{ do } c_1, \sigma_0\rangle \to \sigma_1} \wedge \exists \widetilde{d_1}\ \frac{\dfrac{\widetilde{d_1}}{\langle b, \sigma_0\rangle \to \textbf{false}}}{\langle \textbf{while } b \textbf{ do } c_1, \sigma_0\rangle \to \sigma_1'}$$

and we get that $\sigma_1 = \sigma_1'$ because both are equal to σ_0 (as in the case in which c is **skip**).

Now let us consider the case $\langle b, \sigma_0\rangle \to$ **true**. In this case (*Hyp*) is:

$$\frac{\dfrac{d_1}{\langle b, \sigma_0\rangle \to \textbf{true}}\quad \dfrac{d_2}{\langle c_1, \sigma_0\rangle \to \sigma'}\quad \dfrac{d_3}{\langle \textbf{while } b \textbf{ do } c_1, \sigma'\rangle \to \sigma_1}}{\langle \textbf{while } b \textbf{ do } c_1, \sigma_0\rangle \to \sigma_1} \wedge$$

$$\exists \widetilde{d_1}, \widetilde{d_2}, \widetilde{d_3}\ \frac{\dfrac{\widetilde{d_1}}{\langle b, \sigma_0\rangle \to \textbf{true}}\quad \dfrac{\widetilde{d_2}}{\langle c_1, \sigma_0\rangle \to \sigma^*}\quad \dfrac{\widetilde{d_3}}{\langle \textbf{while } b \textbf{ do } c_1, \sigma^*\rangle \to \sigma_1'}}{\langle \textbf{while } b \textbf{ do } c_1, \sigma_0\rangle \to \sigma_1'}$$

Since $d_2 \prec^+ d$ we have $P(d_2)$. Thus, $\sigma' = \sigma^*$. We get that (note that we have written σ^*, instead of σ'):

$$\frac{d_3}{\langle \textbf{while } b \textbf{ do } c_1, \sigma^*\rangle \to \sigma_1} \wedge \exists \widetilde{d_3}\ \frac{\widetilde{d_3}}{\langle \textbf{while } b \textbf{ do } c_1, \sigma^*\rangle \to \sigma_1'}$$

Since $d_3 \prec^+ d$ we have $P(d_3)$. Thus, $\sigma_1 = \sigma_1'$. This concludes the proof. □

We also have the following fact.

FACT 2.3. We have that: $\forall c \in \mathbf{Com}$, $\forall \sigma, \sigma' \in \mathit{State}$, it is not the case that $\langle \mathbf{while\ true\ do}\ c,\ \sigma\rangle \to \sigma'$.

PROOF. By absurdum. We assume the negation of the fact to be shown, that is, $\exists c \in \mathbf{Com}$, $\exists \sigma, \sigma' \in \mathit{State}$, $\langle \mathbf{while\ true\ do}\ c,\ \sigma\rangle \to \sigma'$. Consider the *minimal* derivation d such that $\dfrac{d}{\langle \mathbf{while\ true\ do}\ c,\ \sigma\rangle \to \sigma'}$. The derivation d is of the form:

$$\dfrac{\dfrac{d_1}{\langle \mathbf{true}, \sigma\rangle \to \mathbf{true}} \qquad \dfrac{d_2}{\langle \mathbf{skip},\ \sigma\rangle \to \sigma} \qquad \dfrac{d_3}{\langle \mathbf{while\ true\ do}\ c,\ \sigma\rangle \to \sigma'}}{\langle \mathbf{while\ true\ do}\ c,\ \sigma\rangle \to \sigma'}$$

which contains a proper subderivation d_3 of $\langle \mathbf{while\ true\ do}\ c,\ \sigma\rangle \to \sigma'$.

This is a contradiction because we have assumed that d is the minimal derivation of $\langle \mathbf{while\ true\ do}\ c,\ \sigma\rangle \to \sigma'$. □

3. Denotational Semantics of the Imperative Language IMP

Let us consider the language IMP of Section 2 on page 118. For that language IMP we consider the following semantic domains.

(i) N is the set of integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$.

(ii) $N_\perp$ is the flat cpo whose underlying set is $N \cup \{\perp\}$ with $\perp \notin N$, such that for all distinct $i, j \in N \cup \{\perp\}$, we have that $i \sqsubseteq j$ iff $i = \perp$.

(iii) *State* is the set of functions from **Loc** to N, each of these function is called a *state*. *State* is a cpo in the sense that every element is related only to itself in the partial order of the cpo.

(iv) $\mathit{State}_\perp$ is the flat cpo of the functions whose underlying set is $\mathit{State} \cup \{\perp\}$. The bottom element $\perp$ of this cpo is the function $\lambda X \in \mathbf{Loc}.\ \perp$ which for any given location X, returns the bottom element $\perp$ in $N_\perp$. Every state $\sigma \in \mathit{State}_\perp$ different from $\lambda X \in \mathbf{Loc}.\ \perp$, maps every location X to an element of N.

In what follows we will feel free to use the symbol $\perp$ to denote either the bottom element of $\mathit{State}_\perp$ or the bottom element of $N_\perp$. The context will disambiguate between these two uses.

(v) State^k is the set of all sequences of states of length k, for any $k \geq 0$. The concatenation of sequences is denoted by '$\cdot$'. Given two sets A and B of sequences, by $A \cdot B$, we denote the set $\{s_1 \cdot s_2 \mid s_1 \in A \text{ and } s_2 \in B\}$. For instance, $\mathit{State}^k \cdot \mathit{State}_\perp$ is the set of all sequences of states of length $k{+}1$ whose last element is a state in the set *State* or the state $\perp$.

3.1. Denotational Semantics of Arithmetic Expressions.

The semantic function $[\![\_]\!]$ for arithmetic expressions is a function from $\mathbf{Aexp} \times \mathit{State}$ to N. As usual, the arithmetic expressions in **Aexp** is written inside the fat square brackets $[\![\_]\!]$ and the state in *State* is written to the right of those brackets.

Given a state $\sigma \in \mathit{State}$, the semantics of an arithmetic expression $a \in \mathbf{Aexp}$ is defined by structural induction as follows:

$$\begin{array}{lcl} [\![n]\!]\,\sigma & = & n \\ [\![X]\!]\,\sigma & = & \sigma(X) \\ [\![a_1\ \mathbf{op}\ a_2]\!]\,\sigma & = & [\![a_1]\!]\,\sigma\ \ op\ \ [\![a_2]\!]\,\sigma \end{array}$$

where $op\colon N\times N \to N$ is the semantic operation corresponding to $\mathbf{op} \in \{+, -, \times\}$.

As usual, in the above semantic equations, the metavariables $n \in N$, $X \in \mathbf{Loc}$, and $a_1, a_2 \in \mathbf{Aexp}$ are assumed to be universally quantified at the front.

Note that in defining the operational semantics of arithmetic expressions, we assume that the given state σ is different from $\perp$.

3.2. Denotational Semantics of Boolean Expressions.

The semantic function $[\![\_]\!]$ for boolean expressions is a function from $\mathbf{Bexp} \times \mathit{State}_\perp$ to $\{\mathit{true}, \mathit{false}\}$.

Given a state $\sigma \in \mathit{State}_\perp$, the semantics of a boolean expression $b \in \mathbf{Bexp}$ is defined as follows:

- for $\sigma = \perp$,

$$[\![b]\!]\,\perp \;= \mathit{true}$$

- for $\sigma \neq \perp$ (the definition is by structural induction according to the following equations):

$$\begin{array}{lcl}
[\![\mathbf{true}]\!]\,\sigma & = & \mathit{true} \\
[\![\mathbf{false}]\!]\,\sigma & = & \mathit{false} \\
[\![a_1\ \mathbf{rop}\ a_2]\!]\,\sigma & = & [\![a_1]\!]\,\sigma\ \mathit{rop}\ [\![a_2]\!]\,\sigma \\
[\![\neg b]\!]\,\sigma & = & \mathit{not}\ ([\![b]\!]\,\sigma) \\
[\![b_1 \vee b_2]\!]\,\sigma & = & [\![b_1]\!]\,\sigma\ \mathit{or}\ [\![b_2]\!]\,\sigma \\
[\![b_1 \wedge b_2]\!]\,\sigma & = & [\![b_1]\!]\,\sigma\ \mathit{and}\ [\![b_2]\!]\,\sigma \\
[\![b_1 \Rightarrow b_2]\!]\,\sigma & = & [\![b_1]\!]\,\sigma\ \mathit{implies}\ [\![b_2]\!]\,\sigma
\end{array}$$

where $\mathit{rop} : N\times N \to \{\mathit{true}, \mathit{false}\}$ is the semantic relational operator corresponding to $\mathbf{rop} \in \{<, \leq, =, \geq, >\}$.

As usual, in the above semantic equations the metavariables $a_1, a_2 \in \mathbf{Aexp}$, and $b, b_1, b_2 \in \mathbf{Bexp}$ are assumed to be universally quantified at the front. The semantic operators *not* ($\neg$), *or* ($\vee$), *and* ($\wedge$), and *implies* ($\Rightarrow$) are those of the Propositional Calculus.

3.3. Denotational Semantics of Commands.

The semantic function $[\![\_]\!]$ for commands is a function from $\mathbf{Com} \times \mathit{State}_\perp$ to $\mathit{State}_\perp$.

Given a state $\sigma \in \mathit{State}_\perp$, the semantics of a command $c \in \mathbf{Com}$ is defined as follows:

- for $\sigma = \perp$,

$$[\![c]\!]\,\perp \;= \perp \quad \text{(that is, the semantic function } [\![\_]\!] \text{ is a strict function)}$$

- for $\sigma \neq \perp$ (the definition is by structural induction according to the following equations):

$$\begin{array}{lcll}
[\![\mathbf{skip}]\!]\,\sigma & = & \sigma & \\
[\![X := a]\!]\,\sigma & = & \sigma[([\![a]\!]\,\sigma)/X] & \\
[\![c_1; c_2]\!]\,\sigma & = & [\![c_2]\!]\,([\![c_1]\!]\,\sigma) & (\dagger 1) \\
[\![\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!]\,\sigma & = & \mathit{cond}([\![b]\!]\,\sigma,\ [\![c_1]\!]\,\sigma,\ [\![c_2]\!]\,\sigma) & (\dagger 2) \\
[\![\mathbf{while}\ b\ \mathbf{do}\ c_0]\!]\,\sigma & = & \mathit{cond}([\![b]\!]\,\sigma,\ [\![\mathbf{while}\ b\ \mathbf{do}\ c_0]\!]\,([\![c_0]\!]\,\sigma),\ \sigma) &
\end{array}$$

Recall that *cond* is a continuous function from $\{true, false\} \times State_\perp \times State_\perp$ to $State_\perp$.

The meaning of the semantic function for commands is the minimal fixpoint of the continuous functional defined by the above semantic equations. In particular, by recalling Kleene Theorem, we have that:

$[\![\mathbf{while}\, b\, \mathbf{do}\, c_0]\!] = (\lambda f.\lambda\sigma.\ cond([\![b]\!]\sigma,\ f([\![c_0]\!]\sigma),\ \sigma))\ [\![\mathbf{while}\, b\, \mathbf{do}\, c_0]\!]$

that is,

$[\![\mathbf{while}\, b\, \mathbf{do}\, c_0]\!] = \bigsqcup_{n\geq 0} \tau^n(\perp)$

where: (i) $\tau =_{def} \lambda f.\lambda\sigma.\ cond([\![b]\!]\sigma,\ f([\![c_0]\!]\sigma),\ \sigma)$ is a continuous functional from the cpo $[State_\perp \to State_\perp]$ to the cpo $[State_\perp \to State_\perp]$, and (ii) $\perp$ in the term $\tau^n(\perp)$ is the function $\lambda\sigma.\perp$, which is the bottom element of the cpo $[State_\perp \to State_\perp]$.

Given a command c and a state $\sigma \in State$ (thus, in particular, $\sigma \neq \perp$),
(i) if $[\![c]\!]\,\sigma = \perp$ then we say that *starting from state* σ, *the command* c *diverges* (or *does not terminate*), and
(ii) if $[\![c]\!]\,\sigma \neq \perp$ then we say that *starting from state* σ, *the command* c *converges* (or *terminates*).

From the definition of the semantic function $[\![\_]\!]$ for commands it follows that for all commands c, for all states $\sigma \in State_\perp$,

if *either* $\sigma = \perp$ *or* the command c does not terminate starting from a state σ different from $\perp$,
then $[\![c]\!]\sigma = \perp$.

Since a command c may not terminate, the state σ in an expression of the form $[\![c]\!]\,\sigma$ may be $\perp$ (see the above Equation (†1)) and this fact may require the evaluation of a boolean expression in the state σ which is $\lambda X \in \mathbf{Loc}.\perp$ (see the above Equation (†2)).

REMARK 3.1. [**Alternative Semantics of Boolean Expressions**] We could have defined the semantic function $[\![\_]\!]$ for boolean expressions to be a function from $\mathbf{Bexp} \times State_\perp$ to $\{true, false\}_\perp$ defined as follows:
- for $\sigma = \perp$,

$$[\![b]\!]\perp \ = \perp$$

- for $\sigma \neq \perp$ (the definition is by structural induction according to the following equations):

$$\begin{array}{lcl}
[\![\mathbf{true}]\!]\,\sigma & = & \lfloor true \rfloor \\
[\![\mathbf{false}]\!]\,\sigma & = & \lfloor false \rfloor \\
[\![a_1\ \mathbf{rop}\ a_2]\!]\,\sigma & = & \lfloor [\![a_1]\!]\,\sigma\ rop\ [\![a_2]\!]\,\sigma \rfloor \\
[\![\neg b]\!]\,\sigma & = & not_\perp\,([\![b]\!]\,\sigma) \\
[\![b_1 \vee b_2]\!]\,\sigma & = & [\![b_1]\!]\,\sigma\ \ or_\perp\ \ [\![b_2]\!]\,\sigma \\
[\![b_1 \wedge b_2]\!]\,\sigma & = & [\![b_1]\!]\,\sigma\ \ and_\perp\ \ [\![b_2]\!]\,\sigma \\
[\![b_1 \Rightarrow b_2]\!]\,\sigma & = & [\![b_1]\!]\,\sigma\ \ implies_\perp\ \ [\![b_2]\!]\,\sigma
\end{array}$$

where $rop : N \times N \to \{true, false\}$ is the semantic relational operator corresponding to $\mathbf{rop} \in \{<, \leq, =, \geq, >\}$. The operators $not_\perp$, $or_\perp$, $and_\perp$, $implies_\perp$ are the strict extensions of the corresponding operators of the Propositional Calculus. For instance, $or_\perp$ is a function from $T_\perp \times T_\perp$ to $T_\perp$, where $T_\perp$ denotes the cpo $\{true, false\}_\perp$.

Then, when defining the semantics of the commands, instead of the continuous function *cond* from $\{true, false\} \times State_\perp \times State_\perp$ to $State_\perp$, we should have used the function

$$\lambda b, \sigma_1, \sigma_2.\, b \rightarrow \sigma_1 \,|\, \sigma_2$$

from $\{true, false\}_\perp \times State_\perp \times State_\perp$ to $State_\perp$, which satisfies the following equations:

$$\begin{array}{rcl} \perp \rightarrow \sigma_1 \,|\, \sigma_2 & = & \perp \\ \lfloor true \rfloor \rightarrow \sigma_1 \,|\, \sigma_2 & = & \sigma_1 \\ \lfloor false \rfloor \rightarrow \sigma_1 \,|\, \sigma_2 & = & \sigma_2 \end{array}$$

The function $\lambda b, \sigma_1, \sigma_2.\, b \rightarrow \sigma_1 \,|\, \sigma_2$ is a continuous function.

This alternative definition of the semantics of the boolean expressions determines a semantics of commands which is equivalent to the one we have given in Section 3.3 on page 123. □

NOTE 3.2. We use the same symbol $[\![\_]\!]$ for denoting the semantic function for (i) arithmetic expressions, (ii) boolean expressions, and (iii) commands. The context will disambiguate these different uses of the symbol $[\![\_]\!]$.

4. Assertions, Hoare Triples, and Weakest Preconditions

We begin this section by introducing some syntactic domains. In particular, we will introduce the following two sets that extend, respectively, the sets **Aexpv** and **Bexp** which we have defined in Section 1 on page 117:

(1) the set **Aexpv** of the *arithmetic expressions with integer variables*, which is a superset of the set **Aexp** of the Arithmetic Expressions (see page 117), that includes also expressions constructed by using the integer variables belonging to the set **Intvar**, and

(2) the set **Assn** of the *assertions*, which is a superset of the set **Bexp** of the Boolean Expressions (defined on page 117) that includes also formulas quantified over the integer variables in **Intvar**. These quantified formulas will be required for denoting the so called invariants (see, for instance, the formula I in the program for the Ackermann function on page 148 with the quantifier $\exists k$ and $\exists n_1, \ldots, n_k$).

We consider the following syntactic domains.

(i) The set of the integer numbers $N = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$.

(ii) The set **Intvar** of the *integer variables*. The variable i ranges over **Intvar**.

(iii) The set **Aexpv** of the *arithmetic expressions with integer variables* defined as follows:

$$a \ ::= n \mid X \mid a_1 \ \mathbf{op} \ a_2 \mid i$$

where $a, a_1, a_2 \in \mathbf{Aexpv}$, $n \in N$, $X \in \mathbf{Loc}$, $\mathbf{op} \in \{+, -, \times\}$, and $i \in \mathbf{Intvar}$.

(iv) The set **Assn** of the *assertions* defined as follows:

$$A \ ::= \mathbf{true} \mid \mathbf{false} \mid a_1 \, \mathbf{rop} \, a_2 \mid \neg A \mid A_1 \vee A_2 \mid A_1 \wedge A_2 \mid A_1 \Rightarrow A_2 \mid \forall i.\, A \mid \exists i.\, A$$

where $A, A_1, A_2 \in \mathbf{Assn}$, $a_1, a_2 \in \mathbf{Aexpv}$, $\mathbf{rop} \in \{<, \leq, =, \geq, >\}$, and $i \in \mathbf{Intvar}$.

(v) The set **Com** of the *commands* defined as follows:

$$c ::= \textbf{skip} \mid X := a \mid c_1; c_2 \mid \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{while } b \textbf{ do } c_0$$

where $c, c_0, c_1, c_2 \in \textbf{Com}$, $X \in \textbf{Loc}$, $a \in \textbf{Aexp}$ (the definition of the set **Aexp** of the Arithmetic Expressions is given on page 117), and $b \in \textbf{Bexp}$ (the definition of the set **Bexp** of the Boolean Expressions is given on page 117). Note that in assignments of the form $X := a$, we have that a belongs to **Aexp**, rather than **Aexpv**.

We have that: (i) $\textbf{Aexp} \subseteq \textbf{Aexpv}$ (the inclusion is strict because integer variables do not belong to the set **Aexp** of the Arithmetic Expressions), and (ii) $\textbf{Bexp} \subseteq \textbf{Assn}$ (the inclusion is strict because integer variables and quantified formulas do not occur in any boolean expression of **Bexp**).

Let an *interpretation* I be a function from **Intvar** to N.

Given any expression *exp*, by $exp[a/X]$ we denote the result of replacing in *exp* all occurrences of X by a. If *exp* is an assertion, then by $exp[a/X]$ we denote the result of replacing all *free* occurrences of X by a [**12**].

4.1. Semantics of Arithmetic Expressions with Integer Variables.

The semantic function $[\![\_]\!]$ for arithmetic expressions with integer variables is a function from $\textbf{Aexpv} \times (\textbf{Intvar} \to N) \times \mathit{State}$ to N.

Given an interpretation $I \in \textbf{Intvar} \to N$ and a state $\sigma \in \mathit{State}$, the semantics of an arithmetic expression with integer variables $a \in \textbf{Aexpv}$ is defined by structural induction, according to the following equations:

$$\begin{array}{lcl} [\![n]\!]\, I\, \sigma & = & n \\ [\![X]\!]\, I\, \sigma & = & \sigma(X) \\ [\![a_1 \textbf{ op } a_2]\!]\, I\, \sigma & = & [\![a_1]\!]\, I\, \sigma \;\; op \;\; [\![a_2]\!]\, I\, \sigma \\ [\![i]\!]\, I\, \sigma & = & I(i) \end{array}$$

where $op : N \times N \to N$ is the semantic operation corresponding to $\textbf{op} \in \{+, -, \times\}$.

In the above semantic equations, the metavariables $n \in N$, $i \in \textbf{Intvar}$, $X \in \textbf{Loc}$, and $a_1, a_2 \in \textbf{Aexpv}$ are assumed to be universally quantified at the front.

We have the following lemmata.

LEMMA 4.1. [**Substitution of Integer Variables in Arithmetic Expressions**] For all arithmetic expressions $a \in \textbf{Aexpv}$, for all integers $n \in N$, for all integer variables i, for all interpretations $I \in \textbf{Intvar} \to N$, and for all states $\sigma \in \mathit{State}$, we have that:

$[\![a[n/i]]\!]\, I\, \sigma = [\![a]\!]\, I[n/i]\, \sigma$.

PROOF. By induction on the structure of a. □

As usual, $I[n/i]$ denotes the interpretation I' which is equal to I except that $I'(i) = n$.

LEMMA 4.2. [**Substitution of Subexpressions in Arithmetic Expressions**] For all interpretations $I \in \textbf{Intvar} \to N$, for all states $\sigma \in \mathit{State}$, for all arithmetic expressions with integer variables $a_1, a_2 \in \textbf{Aexpv}$, for all locations $X \in \textbf{Loc}$, we have that:

$[\![a_1[a_2/X]]\!] I\, \sigma = [\![a_1]\!] I\, \sigma[[\![a_2]\!] I\, \sigma/X]$.

PROOF. By induction on the structure of a_1. □

Recall that the substitution a_2/X applied to the arithmetic expression a_1, replaces *all* occurrences of X in a_1 by the arithmetic expression a_2. As usual, $\sigma[v/X]$ denotes a state σ' which is equal to σ except that $\sigma'(X) = v$.

4.2. Semantics of Assertions.

Given an interpretation $I \in \mathbf{Intvar} \to N$, a state $\sigma \in \mathit{State}_\perp$, and an assertion $A \in \mathbf{Assn}$, we say that the assertion A *is true for* (or *in*) *the interpretation* I and the state σ, and we write $I, \sigma \models A$, iff

either $\sigma = \perp$ (thus, we have that $I, \perp \models A$)

or $\sigma \neq \perp$ and $I, \sigma \models A$ can be derived by using the following axiom and deduction rules (which are given by structural induction):

$$
\begin{array}{lll}
I, \sigma \models \mathbf{true} & & \\
I, \sigma \models a_1 \ \mathbf{rop}\ a_2 & \text{if} & [\![a_1]\!]\, I\, \sigma \ \mathit{rop}\ [\![a_2]\!]\, I\, \sigma \\
I, \sigma \models \neg A & \text{if} & \mathit{not}\ (I, \sigma \models A) \\
I, \sigma \models A_1 \vee A_2 & \text{if} & I, \sigma \models A_1 \ \ \mathit{or} \ \ I, \sigma \models A_2 \\
I, \sigma \models A_1 \wedge A_2 & \text{if} & I, \sigma \models A_1 \ \ \mathit{and} \ \ I, \sigma \models A_2 \\
I, \sigma \models A_1 \Rightarrow A_2 & \text{if} & I, \sigma \models A_1 \ \ \mathit{implies} \ \ I, \sigma \models A_2 \\
I, \sigma \models \forall i.A & \text{if} & \mathit{for\ all}\ n \in N,\ I[n/i], \sigma \models A \\
I, \sigma \models \exists i.A & \text{if} & \mathit{there\ exists}\ n \in N,\ I[n/i], \sigma \models A
\end{array}
$$

where $\mathit{rop} : N \times N \to \{\mathit{true}, \mathit{false}\}$ is the semantic operator corresponding to syntactic operator $\mathbf{rop} \in \{<, \leq, =, \geq, >\}$.

In the above rules the metavariables $a_1, a_2 \in \mathbf{Aexpv}$ and $A, A_1, A_2 \in \mathbf{Assn}$ are assumed to be universally quantified at the front.

Note that integer variables may occur in assertions because integer variables may occur in arithmetic expressions with variables and arithmetic expressions with variables may occur in assertions.

LEMMA 4.3. For all boolean expressions $b \in \mathbf{Bexp}$, for all states $\sigma \in \mathit{State}_\perp$, we have that:

$[\![b]\!]\, \sigma = \mathit{true}$ iff for all interpretations I, $I, \sigma \models b$, and

$[\![b]\!]\, \sigma = \mathit{false}$ iff for all interpretations I, $\mathit{not}\ (I, \sigma \models b)$ (that is, $I, \sigma \models \neg b$).

PROOF. By induction on the structure of b. Recall that integer variables do not occur in boolean expressions. □

LEMMA 4.4. [**Substitution of Arithmetic Expressions in Assertions**] For all interpretations $I \in \mathbf{Intvar} \to N$, for all states $\sigma \in \mathit{State}_\perp$, for all arithmetic expressions with integer variables $a \in \mathbf{Aexpv}$, for all assertions $B \in \mathbf{Assn}$, for all locations $X \in \mathbf{Loc}$, we have that:

if $\sigma \neq \perp$ then $I, \sigma \models B[a/X]$ iff $I, \sigma[[\![a]\!] I \sigma / X] \models B$.

PROOF. By induction on the structure of B. Recall that if $\sigma = \perp$ then $I, \sigma \models B$ holds for every assertion B. □

Recall that the substitution a/X applied to the assertion B, replaces *all free* occurrences of X in B by a.

4.3. The Calculus of Hoare Triples.

Now we introduce an axiomatic system for establishing a ternary relation, subset of $\mathbf{Assn} \times \mathbf{Com} \times \mathbf{Assn}$, denoted $\vdash \{A\}\, c\, \{B\}$, for any given assertions $A, B \in \mathbf{Assn}$ and command $c \in \mathbf{Com}$. A triple of the form $\{A\}\, c\, \{B\}$ is said to be a *Hoare triple*.

DEFINITION 4.5. [**Hoare Calculus. Derivability of Hoare Triple**s] Given the assertions A and B and the command c, we say that the triple $\{A\}\, c\, \{B\}$ is *derivable* in the Hoare Calculus, and we write $\vdash \{A\}\, c\, \{B\}$, iff there is a proof (that is, a finite derivation) of $\{A\}\, c\, \{B\}$ by using the following axiom $H1$ and inference rules $H2$–$H6$:

$(H1)$ $\quad \{A\}\ \mathbf{skip}\ \{A\}$

$(H2)$ $\quad \dfrac{\models A \Rightarrow B[a/X]}{\{A\}\ X{:=}a\ \{B\}}$

$(H3)$ $\quad \dfrac{\{A\}\ c_1\ \{B\} \qquad \{B\}\ c_2\ \{C\}}{\{A\}\ \ c_1\,;c_2\ \ \{C\}}$

$(H4)$ $\quad \dfrac{\{A \wedge b\}\ c_1\ \{B\} \qquad \{A \wedge \neg b\}\ c_2\ \{B\}}{\{A\}\ \ \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \ \{B\}}$

$(H5)$ $\quad \dfrac{\{A \wedge b\}\ c\ \{A\}}{\{A\}\ \ \mathbf{while}\ b\ \mathbf{do}\ c\ \ \{A \wedge \neg b\}}$

$(H6)$ $\quad \dfrac{\models A \Rightarrow A' \qquad \{A'\}\ c\ \{B'\} \qquad \models B' \Rightarrow B}{\{A\}\ c\ \{B\}}$

In rule $H5$ the assertion A is said to be an *invariant* of the **while-do** loop.

A Hoare triple $\{A\}\, c\, \{B\}$ which is derivable using the Hoare Calculus, is said to be a *partially correct triple* (the reasons for this terminology we will given below).

REMARK 4.6. In the above rules $H2$ and $H6$, given any two assertions A and B, a premise of the form $\models A \Rightarrow B$ means that for all interpretations $I \in \mathbf{Intvar} \to N$, for all states $\sigma \in \mathit{State}_\perp$, the implication $A \Rightarrow B$ holds in the theory of Integer Arithmetics. We will not formally define this theory here. It will be enough to say that: (i) Integer Arithmetics is the theory of the integers with the usual arithmetic operators $+$, $-$, and $\times$, and the usual relational operators $<$, $\leq$, $=$, $\geq$, and $>$, and (ii) if we restrict Integer Arithmetics to the non-negative integers we get Peano Arithmetics. As a consequence, there is no computable decision procedure that given any assertion A, checks whether or not $\models A$ holds in Integer Arithmetics.

DEFINITION 4.7. [**Validity of Hoare Triples**] Given the assertions $A, B \in \mathbf{Assn}$, the command $c \in \mathbf{Com}$, the interpretation $I \in \mathbf{Intvar} \to N$, the state $\sigma \in \mathit{State}_\perp$, we say that the triple $\{A\}\, c\, \{B\}$ *holds in the interpretation* I *and the state* σ, and we write $I, \sigma \models \{A\}\, c\, \{B\}$, if we have that:

$I, \sigma \models A$ implies $I, [\![c]\!]\sigma \models B$.

We say that a triple $\{A\}\, c\, \{B\}$ is *valid*, or *holds*, and we write $\models \{A\}\, c\, \{B\}$, iff for all interpretations $I \in \mathbf{Intvar} \to N$, for all states $\sigma \in \mathit{State}_\perp$, we have that $I, \sigma \models A$ implies $I, [\![c]\!]\sigma \models B$.

When the triple $\{A\}\, c\, \{B\}$ is valid, we say that the command c is *partially correct* with respect to the *precondition* A and the *postcondition* B. This correctness is said to be *partial*, because we have that $\models \{A\}\, c\, \{B\}$ holds iff starting from a state σ where the assertion A holds, we have that: *either* (i) the command c does not terminate, that is, $[\![c]\!]\sigma = \perp$, *or* (ii) the command c terminates and the assertion B holds after the execution of the command c, that is, B holds in the state $[\![c]\!]\sigma$. Thus, the term 'partial correctness' refers to the fact that the command c is not guaranteed to terminate.

We say that the command c is *totally correct* with respect to the precondition A and the postcondition B iff starting from a state where the assertion A holds, (i) the command c terminates, and (ii) the assertion B holds after the execution of the command c. That is, the command c is totally correct with respect to the precondition A and the postcondition B iff for all interpretations I, for all states $\sigma \in \mathit{State}$, $I, \sigma \models A$ implies $[\![c]\!]\sigma \neq \perp$ and $I, [\![c]\!]\sigma \models B$. Thus, the term 'total correctness' refers to the fact that the command c is guaranteed to terminate.

The following theorem tells us that the partially correct triples which the Hoare rules derive, are valid triples.

THEOREM 4.8. [**Soundness Theorem**] For all assertions A and B in **Assn**, for all commands c in **Com**, we have that $\vdash \{A\}\, c\, \{B\}$ implies $\models \{A\}\, c\, \{B\}$.

PROOF. When proving that for every interpretation $I \in \mathbf{Intvar} \to N$, state $\sigma \in \mathit{State}_\perp$, assertions $A, B \in \mathbf{Assn}$, and command $c \in \mathbf{Com}$, it is the case that $I, \sigma \models A$ implies $I, [\![c]\!]\sigma \models B$, we may assume without loss of generality that $\sigma \neq \perp$, because: (i) for every command c, we have that $[\![c]\!]\perp = \perp$, and (ii) for every interpretation I, assertion A, we have that $I, \perp \models A$ holds.

The proof of the theorem is done by rule induction.

(*Rule* $H1$). We have to show that $\models \{A\}\, \mathbf{skip}\, \{A\}$ holds. This follows from the fact that for every interpretation I, state σ, $I, \sigma \models A$ implies $I, [\![\mathbf{skip}]\!]\sigma \models A$, because $[\![\mathbf{skip}]\!]\sigma = \sigma$.

(*Rule* $H2$). Assume $\vdash \{A\}\, X := a\, \{B\}$. By rule $H2$ we get $\models A \Rightarrow B[a/X]$, that is,

for all interpretations I, for all states σ,

$$I, \sigma \models A \text{ implies } I, \sigma \models B[a/X]. \tag{2.1}$$

By Lemma 4.4 on page 127 from (2.1) we get:

for all interpretations I, for all states σ,

$$I, \sigma \models A \text{ implies } I, \sigma[[\![a]\!]I\sigma/X] \models B. \tag{2.2}$$

By definition of $[\![X := a]\!]\sigma$ from (2.2) we get:

for all interpretations I, for all states σ,

$I, \sigma \models A$ implies $I, [\![X := a]\!]\sigma \models B$, that is, $\models \{A\}\, X := a\, \{B\}$.

(*Rule H3*). Assume $\vdash \{A\}\, c_1; c_2\, \{C\}$. By rule $H3$ for some assertion B we get: $\vdash \{A\}\, c_1\, \{B\}$ and $\vdash \{B\}\, c_2\, \{C\}$. By rule induction, we have that:

$$\models \{A\}\, c_1\, \{B\} \text{ and} \tag{3.1}$$

$$\models \{B\}\, c_2\, \{C\}. \tag{3.2}$$

Thus, we have:

for all interpretations I, for all states σ,

$$I, \sigma \models A \text{ implies } I, [\![c_1]\!]\sigma \models B, \quad \text{and} \tag{3.1*}$$

for all interpretations I, for all states σ,

$$I, \sigma \models B \text{ implies } I, [\![c_2]\!]\sigma \models C. \tag{3.2*}$$

By (3.1*) and (3.2*) we get: for all interpretations I, for all states σ,

$I, \sigma \models A$ implies $I, [\![c_2]\!]([\![c_1]\!]\sigma) \models C$, that is, $I, [\![c_1; c_2]\!]\sigma \models C$.

Thus, we get $\models \{A\}\, c_1; c_2\, \{C\}$.

(*Rule H4*). Assume $\vdash \{A\}\, \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\, \{B\}$. By rule $H4$ we get: $\vdash \{A \wedge b\}\, c_1\, \{B\}$ and $\vdash \{A \wedge \neg b\}\, c_2\, \{B\}$. By rule induction we have that:

$$\models \{A \wedge b\}\, c_1\, \{B\} \text{ and} \tag{4.1}$$

$$\models \{A \wedge \neg b\}\, c_2\, \{B\}. \tag{4.2}$$

Thus,

for all interpretations I, for all states σ,

$$I, \sigma \models A \wedge b \text{ implies } I, [\![c_1]\!]\sigma \models B, \quad \text{and} \tag{4.1*}$$

for all interpretations I, for all states σ,

$$I, \sigma \models A \wedge \neg b \text{ implies } I, [\![c_2]\!]\sigma \models B. \tag{4.2*}$$

By (4.1*) and (4.2*) we get: for all interpretations I, for all states σ,

$I, \sigma \models A$ implies $I, [\![\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!]\sigma \models B$.

Thus, we get $\models \{A\}\, \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\, \{B\}$.

(*Rule H5*). Assume $\vdash \{A\}\, \mathbf{while}\ b\ \mathbf{do}\ c\, \{A \wedge \neg b\}$. By rule $H5$ we get $\vdash \{A \wedge b\}\, c\, \{A\}$. By rule induction we have that:

$$\models \{A \wedge b\}\, c\, \{A\}. \tag{5.1}$$

We have to show that $\models \{A\}\, \mathbf{while}\ b\ \mathbf{do}\ c\, \{A \wedge \neg b\}$, that is, for all interpretations I, for all states σ, $I, \sigma \models A$ implies $I, [\![\mathbf{while}\ b\ \mathbf{do}\ c]\!]\sigma \models A \wedge \neg b$.

By definition of $[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!]\sigma$ (see page 124), we have that:

$[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] = \bigsqcup_{n \geq 0} \tau^n(\bot)$

where: (i) $\tau =_{def} \lambda f.\lambda\sigma.\, cond([\![b]\!]\sigma,\, f([\![c]\!]\sigma),\, \sigma)$, and (ii) $\bot$ is the function $\lambda\sigma.\bot$ of the cpo $[State_\bot \to State_\bot]$.

Now, let us consider the property P: $(State_\bot \to State_\bot) \to \{true, false\}$ defined as follows:

$P(\tau^n(\bot)) =_{def}$ for all interpretations I, for all states $\sigma \in State_\bot$,

$$I, \sigma \models A \ \text{ implies } \ I, \tau^n(\bot)\sigma \models A \wedge \neg b.$$

We have that $\models \{A\}\,\textbf{while}\ b\ \textbf{do}\ c\,\{A \wedge \neg b\}$ holds and, thus, we conclude the proof for the case of Rule $H5$, if we prove that:

for all $n \geq 0$, $P(\tau^n(\bot))$ holds, and $\quad (P1)$

if for all $n \geq 0$, $P(\tau^n(\bot))$ holds, then $P(\bigsqcup_{n\geq 0} \tau^n(\bot))$ holds. $\quad (P2)$

Let us prove properties $(P1)$ and $(P2)$. The proof of $(P2)$ is required because in order to use Scott induction, the property P should be inclusive.

Proof of $(P1)$. By induction on n.

(*Basis*) Since $\tau^0(\bot) = \bot$, we have to show that for all interpretations I, for all states $\sigma \in State_\bot$, $I, \sigma \models A$ implies $I, \bot \models A \wedge \neg b$. This is obvious, because for all assertions φ, we have that $I, \bot \models \varphi$ holds.

(*Step*) Assume $P(\tau^n(\bot))$. We show $P(\tau^{n+1}(\bot))$ as follows. Take any interpretation I and any state σ and assume

$$I, \sigma \models A. \quad (5.2)$$

From $P(\tau^n(\bot))$ and (5.2) we get $I, \tau^n(\bot)\sigma \models A \wedge \neg b$. We have to show that $I, \tau^{n+1}(\bot)\sigma \models A \wedge \neg b$, that is, $I, \mathit{cond}(\llbracket b \rrbracket\sigma,\ \tau^n(\bot)(\llbracket c \rrbracket\sigma),\ \sigma) \models A \wedge \neg b$. Now there are two cases:

Case $(P1.1)$: $\llbracket b \rrbracket\sigma = \mathit{false}$, and

Case $(P1.2)$: $\llbracket b \rrbracket\sigma = \mathit{true}$.

Case $(P1.1)$. Since $\llbracket b \rrbracket\sigma = \mathit{false}$, we have that

$$I, \sigma \models \neg b. \quad (5.3)$$

We have to show that $I, \sigma \models A \wedge \neg b$. This follows from (5.2) and (5.3).

Case $(P1.2)$. Since $\llbracket b \rrbracket\sigma = \mathit{true}$, we have that

$$I, \sigma \models b. \quad (5.4)$$

We have to show that $I, \tau^n(\bot)(\llbracket c \rrbracket\sigma) \models A \wedge \neg b$. From (5.2) and (5.4) we get

$$I, \sigma \models A \wedge b. \quad (5.5)$$

From (5.5) and (5.1) we get

$$I, \llbracket c \rrbracket\sigma \models A. \quad (5.6)$$

From (5.6) and the inductive hypothesis $P(\tau^n(\bot))$, we get $I, \tau^n(\bot)(\llbracket c \rrbracket\sigma) \models A \wedge \neg b$.

Proof of $(P2)$.

Assume that for all $n \geq 0$, for all interpretations I, for all states $\sigma \in State_\bot$, $I, \sigma \models A$ implies $I, \tau^n(\bot)\sigma \models A \wedge \neg b$. In order to show that $P(\bigsqcup_{n\geq 0} \tau^n(\bot))$ holds, we take any interpretation I and any state σ such that $I, \sigma \models A$ holds and we have to show that I, $(\bigsqcup_{n\geq 0} \tau^n(\bot))\sigma \models A \wedge \neg b$ holds. Now there are two cases:

Case $(P2.1)$: $(\bigsqcup_{n\geq 0} \tau^n(\bot))\sigma = \bot$, and

Case $(P2.2)$: there exists a state σ' such that $(\bigsqcup_{n\geq 0} \tau^n(\bot))\sigma = \sigma' \neq \bot$.

Case $(P2.1)$. In this case we have that I, $(\bigsqcup_{n\geq 0} \tau^n(\bot))\sigma \models A \wedge \neg b$ is equivalent to $I, \bot \models A \wedge \neg b$ and thus, it holds by definition of $\models$.

Case $(P2.2)$. In this case, since τ is a continuous function from $State_\bot$ to $State_\bot$ (recall that, in particular, *cond* is continuous), and $State_\bot$ is a flat cpo, we have that there exists $m > 0$ such that $\tau^m(\bot)\sigma = \sigma'$. Thus, I, $(\bigsqcup_{n\geq 0} \tau^n(\bot))\sigma \models A \wedge \neg b$ follows from the fact that for all $n \geq 0$, $P(\tau^n(\bot))$ holds, and $P(\tau^m(\bot))$ means that for all

interpretations I, for all states $\sigma \in \mathit{State}_\perp$, $I, \sigma \models A$ implies $I, \tau^m(\perp)\sigma \models A \wedge \neg b$ which in this Case ($P2.2$) is equal to $I, \sigma' \models A \wedge \neg b$.

This concludes the proof for the case of rule $H5$.

(*Rule* $H6$). Assume $\vdash \{A\}\, c\, \{B\}$. By rule $H6$ we get:

$$\models A \Rightarrow A', \tag{6.1}$$

$$\vdash \{A'\}\, c\, \{B'\}, \text{ and} \tag{6.2}$$

$$\models B' \Rightarrow B. \tag{6.3}$$

By rule induction from (6.2) we have:

$$\models \{A'\}\, c\, \{B'\}. \tag{6.2.1}$$

From (6.1), (6.2.1), and (6.3), we have that for all interpretations I, for all states σ,

$$I, \sigma \models A \text{ implies } I, \sigma \models A', \tag{6.1.1}$$

$$I, \sigma \models A' \text{ implies } I, [\![c]\!]\sigma \models B', \text{ and} \tag{6.2.2}$$

$$I, \sigma \models B' \text{ implies } I, \sigma \models B. \tag{6.3.1}$$

From (6.1.1), (6.2.2), and (6.3.1), by transitivity of implication, we have that:
for all interpretations I, for all states σ, $I, \sigma \models A$ implies $I, [\![c]\!]\sigma \models B$, that is, $\models \{A\}\, c\, \{B\}$. This concludes the proof of the theorem. □

Now we will show that, although in a weak form, also the reverse implication of the Soundness Theorem (see Theorem 4.8 on page 129) holds, that is, $\models \{A\}\, c\, \{B\}$ implies $\vdash \{A\}\, c\, \{B\}$ (see Theorem 4.24 on page 143). First we need some definitions, lemmata, and theorems.

Definition 4.9. [**Extension of an Assertion**] Given an assertion $A \in \mathbf{Assn}$ and an interpretation $I \in \mathbf{Intvar} \to N$, the *extension of the assertion A with respect to I*, denoted A^I, is the set of states: $\{\sigma \mid \sigma \in \mathit{State}_\perp \wedge I, \sigma \models A\}$. In particular, for all interpretations I, $\mathbf{false}^I = \{\perp\}$.

We have the following fact.

Fact 4.10. (i) For all assertions $A, B \in \mathbf{Assn}$, for all interpretations $I \in \mathbf{Intvar} \to N$, if $A^I = B^I$ then $\models A \Leftrightarrow B$. (ii) For all interpretations $I \in \mathbf{Intvar} \to N$, for all assertions A, we have that $\perp \in A^I$.

Proof. (i) By hypothesis we have that for all interpretations $I \in \mathbf{Intvar} \to N$, for all states $\sigma \in \mathit{State}_\perp$, $I, \sigma \models A$ iff $I, \sigma \models B$. Thus, for all I, σ, $I, \sigma \models A \Leftrightarrow B$. (ii) It follows from the fact that for all interpretations I, for all assertions A, we have that $I, \perp \models A$. □

Definition 4.11. [**Weakest Precondition and its Extension**] Given a command $c \in \mathbf{Com}$, an assertion $B \in \mathbf{Assn}$, and an interpretation $I \in \mathbf{Intvar} \to N$, the *weakest precondition of B with respect to c* is a formula, denoted $wp(c, B)$, such that its *extension with respect to the interpretation I*, denoted $wp^I(c, B)$, is the set

$$wp^I(c, B) =_{def} \{\sigma \mid \sigma \in \mathit{State}_\perp \ \wedge \ I, [\![c]\!]\sigma \models B\}.$$

Thus, $wp^I(c, B)$ is the *largest* subset of $\mathit{State}_\perp$, such that starting from any state in that largest subset, the command c *either* does not terminate *or* produces a state σ' such that $I, \sigma' \models B$.

Note that in Definition 4.11 on the facing page we said that $wp(c, B)$ is a *formula*. In Theorem 4.15 on the next page we will show that actually there is an *assertion* whose extension is equal to that of $wp(c, B)$.

The following fact follows immediately from the above Definition 4.11 on the facing page.

FACT 4.12. For all states $\sigma \in State_\perp$, for all interpretations I, for all commands c, for all assertions B, we have that $\sigma \in wp^I(c, B)$ iff $I, \sigma \models wp(c, B)$.

What we have called *weakest precondition*, sometimes in the literature is also called *weakest liberal precondition*, and the term *weakest precondition* is reserved for the similar notion which refers to total correctness, rather than partial correctness [**19**, page 101]. Our notion of weakest precondition refers to partial correctness, because in Definition 4.11 on the preceding page we allow the command c not to terminate.

Recall that the state which is obtained after a non-terminating command is the bottom element of the cpo $State_\perp$, that is, $\lambda X \in \mathbf{Loc}.\, \perp$, where $\perp \in N_\perp$.

For all interpretations $I \in \mathbf{Intvar} \to N$, for all states $\sigma \in State_\perp$, for all commands c such that $[\![c]\!]\sigma = \perp$, for all assertions B, we have that

$\sigma \in wp^I(c, B)$ iff {by Fact 4.12}

iff $I, [\![c]\!]\sigma \models B$ iff {by $[\![c]\!]\sigma = \perp$}

iff $I, \perp \models B$ iff {by the semantics of assertions (see Section 4.2 on page 127)}

iff *true*.

Now, let us consider a command c such that for all states $\sigma \in State_\perp$, $[\![c]\!]\sigma = \perp$, that is, *either* the command c does not terminate starting from σ *or* $\sigma = \perp$. Then for all interpretations I, for all assertions B, $wp(c, B)$ is *true* and the extension of $wp(c, B)$ with respect to I is the whole cpo $State_\perp$.

From Definition 4.11 on the facing page and the fact that the semantic function $[\![\_]\!]$ is strict, it follows that:
(i) for all commands c, for all assertions B, for all interpretations I, we have that

$\perp \in wp^I(c, B)$, where $\perp$ is the bottom element of the cpo $State_\perp$,

(ii) for all states $\sigma \in State$ (thus, $\sigma \neq \perp$), for all commands c, we have that c diverges starting from state σ iff $\sigma \in wp^I(c, \mathbf{false})$ iff

for all interpretations I, $I, \sigma \models wp(c, \mathbf{false})$, and

(iii) for all states $\sigma \in State$ (thus, $\sigma \neq \perp$), for all commands c, we have that c converges starting from state σ iff $\sigma \notin wp^I(c, \mathbf{false})$ iff

for all interpretations I, $I, \sigma \models \neg wp(c, \mathbf{false})$.

THEOREM 4.13. [**Properties of Weakest Preconditions**] For all assertions A and B, for all commands c,
(i) $\models \{wp(c, B)\}\, c\, \{B\}$
(ii) $\models \{A\}\, c\, \{B\}$ iff $\models A \Rightarrow wp(c, B)$.

PROOF. The proof of (i) is immediate. The proof of (ii) is as follows. We have that:

$\models \{A\}\, c\, \{B\}$ iff for all I, for all σ, if $I, \sigma \models A$ then $I, [\![c]\!]\sigma \models B$.

By the definition of the weakest precondition, we have that:

for all I, for all σ, $I, [\![c]\!]\sigma \models B$ iff $I, \sigma \models wp(c, B)$.

Thus, for all I, for all σ, if $I, \sigma \models A$ then $I, \sigma \models wp(c, B)$. Hence, $\models A \Rightarrow wp(c, B)$. □

NOTATION 4.14. In what follows and, in particular, in the proof of the following Theorem 4.15 for the case of the **while-do**, for all interpretations I, for all assertions A, we write $I \models A$ to mean that for all states σ, $I, \sigma \models A$ holds. □

THEOREM 4.15. [**Expressiveness Theorem**] Given any command $c \in \mathbf{Com}$ and any assertion $B \in \mathbf{Assn}$, there exists an assertion, call it $A(c, B)$, such that for all interpretations $I \in \mathbf{Intvar} \to N$, its extension $A^I(c, B)$ w.r.t. I is equal to the extension w.r.t. I of the weakest precondition of B w.r.t. the command c, that is, $A^I(c, B) = wp^I(c, B)$.

PROOF. In order to show that $A^I(c, B) = wp^I(c, B)$, by definition of the extension of an assertion, we have to show that for all commands $c \in \mathbf{Com}$, for all assertions $B \in \mathbf{Assn}$, for all interpretations $I \in \mathbf{Intvar} \to N$, for all states $\sigma \in State_\perp$, there exists an assertion $A(c, B)$ such that

$I, [\![c]\!]\sigma \models B$ iff $I, \sigma \models A(c, B)$.

The proof is by cases on the command c.

(*Case* $c = \mathbf{skip}$). The assertion $A(\mathbf{skip}, B)$ to be found is B itself. Indeed, for all interpretations $I \in \mathbf{Intvar} \to N$, for all states $\sigma \in State_\perp$, we have that:

$I, [\![\mathbf{skip}]\!]\sigma \models B$ iff {by definition of $[\![\_]\!]$}

iff $I, \sigma \models B$.

(*Case* $c = X := a$). The assertion $A(X := a, B)$ to be found is $B[a/X]$ (that is, B with all free occurrences of X substituted by a). Indeed, for all interpretations $I \in \mathbf{Intvar} \to N$, for all states $\sigma \in State_\perp$, we have that:

$I, [\![X := a]\!]\sigma \models B$ iff {by definition of $[\![\_]\!]$}

iff $I, \sigma[[\![a]\!]I\sigma/X] \models B$ iff {by Lemma 4.4 on page 127}

iff $I, \sigma \models B[a/X]$.

(*Case* $c = c; c_2$). The assertion $A(c_1; c_2, B)$ to be found is inductively defined as $A(c_1, A(c_2, B))$. Indeed, for all interpretations $I \in \mathbf{Intvar} \to N$, for all states $\sigma \in State_\perp$, we have that:

$I, [\![c_1; c_2]\!]\sigma \models B$ iff {by definition of $[\![\_]\!]$}

iff $I, [\![c_2]\!]([\![c_1]\!]\sigma) \models B$ iff {by structural induction}

iff $I, [\![c_1]\!]\sigma \models A(c_2, B)$ iff {by structural induction}

iff $I, \sigma \models A(c_1, A(c_2, B))$.

(*Case* $c = \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2$). The assertion $A(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, B)$ to be found is inductively defined as $(b \wedge A(c_1, B)) \vee (\neg b \wedge A(c_2, B))$. Indeed, for all interpretations $I \in \mathbf{Intvar} \to N$, for all states $\sigma \in State_\perp$, we have that:

$I, [\![\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!]\sigma \models B$ iff {by definition of $[\![\_]\!]$}

iff $([\![b]\!]\sigma \wedge I, [\![c_1]\!]\sigma \models B) \vee (\neg[\![b]\!]\sigma \wedge I, [\![c_2]\!]\sigma \models B)$ iff {by structural induction}

iff $([\![b]\!]\sigma \wedge I, \sigma \models A(c_1, B)) \vee ([\![\neg b]\!]\sigma \wedge I, \sigma \models A(c_2, B))$ iff {by definition of $\models$}

iff $I, \sigma \models (b \wedge A(c_1, B)) \vee (\neg b \wedge A(c_2, B))$.

(*Case* $c =$ **while** b **do** c_0). The assertion $wp(\textbf{while } b \textbf{ do } c_0, B)$ to be found is an assertion whose extension $wp^I(\textbf{while } b \textbf{ do } c_0, B)$ with respect to the interpretation I is such that a state $\sigma \in wp^I(\textbf{while } b \textbf{ do } c_0, B)$ iff

$$\begin{array}{lll} & \sigma = \bot & \left.\begin{array}{l}\\\end{array}\right] (\alpha) \\ \vee & \forall k \geq 0.\ \forall \sigma_0 \ldots \sigma_k \in \mathit{State}^k \cdot \mathit{State}_\bot. & \left.\begin{array}{l}\\\\\\\end{array}\right. \\ & \big[\sigma_0 = \sigma \ \wedge\ (\forall i.\, (0 \leq i < k) \Rightarrow ([\![b]\!]\sigma_i \wedge [\![c_0]\!]\sigma_i = \sigma_{i+1}))\big] & \left.\begin{array}{l}\\\end{array}\right] (\beta) \\ & \Rightarrow \big([\![b]\!]\sigma_k \ \vee\ (I, \sigma_k \models B)\big). & \left.\begin{array}{l}\\\end{array}\right. \end{array}$$

The correctness of this formula will be proved in Section 7 on page 153.

Note that in the formula $(\alpha) \vee (\beta)$ we can write $\sigma_0 = \sigma \neq \bot$, instead of $\sigma_0 = \sigma$, because if $\sigma = \bot$ then $\sigma \in wp^I(\textbf{while } b \textbf{ do } c_0, B)$ by (α).

Now, before continuing the proof, let us make the following two remarks.

REMARK 4.16. If in (β) we specify that the sequence $\sigma_0 \ldots \sigma_k$ of states belongs to $\mathit{State}_\bot^{k+1}$, rather than $\mathit{State}^k \cdot \mathit{State}_\bot$, we get a formula (β') equivalent to (β), because

(i) the semantic function $[\![\_]\!]$ is strict, that is, for all commands c, $[\![c]\!]\bot = \bot$,

(ii) for all boolean expressions b, $[\![b]\!]\bot = \mathit{true}$, and

(iii) for all interpretations I, for all assertions B, $I, \bot \models B$.

The equivalence of (β) and (β') follows from the fact that for all i, with $0 \leq i < k$, if $\sigma_i = \bot$ then for all j, with $i \leq j \leq k$, we have that $\sigma_j = \bot$. □

REMARK 4.17. If in (β) we specify that the sequence $\sigma_0 \ldots \sigma_k$ of states belongs to State^{k+1}, rather than $\mathit{State}^k \cdot \mathit{State}_\bot$, we get an equivalent formula because if $\sigma_k = \bot$ then $I, \sigma_k \models B$ holds and, thus, the implication $\Rightarrow$ holds. □

Let us continue the proof of the Expressivity Theorem.

Since the formula $(\alpha) \vee (\beta)$ is *not* an assertion, in order to prove this **while-do** case of the theorem, we have to construct an assertion which is equivalent to $(\alpha) \vee (\beta)$. We do so by first avoiding in the formula $(\alpha) \vee (\beta)$ all references to the states σ_i's. This can be done by using sequences of integers and $\bot$'s, as we now indicate. We assume that the locations occurring in the states are $X_1, \ldots, X_\ell$ and these locations are ordered in the sequence $\overline{X} = \langle X_1, \ldots, X_\ell \rangle$. Then, instead of any given state σ, we can use a sequence $\overline{s} = \langle s_1, \ldots, s_\ell \rangle$ made out of integers or $\bot$'s for denoting the values stored for the state σ in the locations $\langle X_1, \ldots, X_\ell \rangle$, so that, for $i = 1, \ldots, \ell$, we have that s_i is the value stored in the location X_i in σ.

In those sequences, if the value stored in a location is undefined, we stipulate that it is $\bot$. Thus, if the state σ is undefined, the sequence $\overline{s}$ is made out of all $\bot$'s. Note that, since the cpo $\mathit{State}_\bot$ is flat, any sequence $\overline{s}$ is made out of either all integers in N or all $\bot$'s.

Let us assume that in the state σ_k the locations $\overline{X} = \langle X_1, \ldots, X_\ell \rangle$ have the values $\overline{s}_k = \langle s_1, \ldots, s_\ell \rangle$. By Lemma 4.4 on page 127, in the formula (β) we can replace $I, \sigma_k \models B$ by $I \models B[s_1/X_1, \ldots, s_\ell/X_\ell]$, also denoted $I \models B[\overline{s}_k/\overline{X}]$. (Recall that for all interpretations I, for all assertions B, when we write $I \models B$ we mean that for all states σ, $I, \sigma \models B$.) Indeed, in $B[s_1/X_1, \ldots, s_\ell/X_\ell]$ there is no occurrence of $X_1, \ldots, X_\ell$ and thus, the value of σ is not significant.

Thus, by recalling that for all interpretations I, $\mathbf{false}^I = \{\bot\}$ and, as stated in Remark 4.17 on the previous page, the state σ_k may be assumed to be an element of $State$, rather than $State_\bot$, we get the following formula equivalent to $(\alpha)\vee(\beta)$:

$$\begin{array}{ll} & \mathbf{false} \\ \vee & \forall k{\geq}0.\ \forall \overline{s}_0 \ldots \overline{s}_k \in (N^\ell)^{k+1}. \\ & \big[\,\overline{X}{=}\overline{s}_0 \ \wedge\ (\forall i.\,(0{\leq}i{<}k) \Rightarrow (b[\overline{s}_i/\overline{X}] \wedge [\![c_0]\!]\sigma_i = \sigma_{i+1}))\big] \\ & \Rightarrow\ (\,b[\overline{s}_k/\overline{X}] \vee I \models B[\overline{s}_k/\overline{X}]\,). \end{array} \qquad (W1)$$

Note that for all i, with $0{\leq}i{<}k$, in the sequence $\overline{s}_i$ the value $\bot$ does not occur. Now we have the following three properties:

$(W1.1)$ $(\mathbf{false} \vee b)$ iff b,

$(W1.2)$ $b[\overline{s}_i/\overline{X}]$ iff $I \models b[\overline{s}_i/\overline{X}]$,

$(W1.3)$ $[\![c_0]\!]\sigma_i{=}\sigma_{i+1} \neq \bot$ with $\sigma_i \in State$ iff for every interpretation I,

$$I, \sigma_i \models \big(wp(c_0, \overline{X}{=}\overline{s}_{i+1}) \ \wedge\ \neg wp(c_0, \mathbf{false})\big)\,[\overline{s}_i/\overline{X}].$$

Point $(W1.1)$ is obvious. Point $(W1.2)$ is a consequence of the fact that no integer variable occurs in any boolean expression. Point $(W1.3)$ can be proved as follows (recall that, as noted on page 133, the command c_0 starting from the state $\sigma_i{\in}State$ converges iff for every interpretation I, $I, \sigma_i \models \neg wp(c_0, \mathbf{false})$):

$[\![c_0]\!]\sigma_i = \sigma_{i+1} \neq \bot$

iff for every interpretation I, $\sigma_i \in wp^I(c_0, \overline{X}{=}\overline{s}_{i+1}) \wedge [\![c_0]\!]\sigma_i \neq \bot$

iff for every interpretation I, $I, \sigma_i \models wp(c_0, \overline{X}{=}\overline{s}_{i+1}) \wedge I, \sigma_i \models \neg wp(c_0, \mathbf{false})$

iff for every interpretation I, $I, \sigma_i \models \big(wp(c_0, \overline{X}{=}\overline{s}_{i+1}) \wedge \neg wp(c_0, \mathbf{false})\big)$

iff for every interpretation I, $I \models \big(wp(c_0, \overline{X}{=}\overline{s}_{i+1}) \wedge \neg wp(c_0, \mathbf{false})\big)[\overline{s}_i/\overline{X}]$.

Thus, by Properties $(W1.1)$, $(W1.2)$, and $(W1.3)$, from $(W1)$ we get:

$$\begin{array}{l} \forall k{\geq}0.\ \forall \overline{s}_0 \ldots \overline{s}_k \in (N^\ell)^{k+1}. \\ \big[\,\overline{X}{=}\overline{s}_0 \wedge \forall i.\,(0{\leq}i{<}k) \Rightarrow \big(I \models \big(b \wedge wp(c_0, \overline{X}{=}\overline{s}_{i+1}) \wedge \neg wp(c_0, \mathbf{false})\big)[\overline{s}_i/\overline{X}]\big)\big] \\ \Rightarrow I \models (b \vee B)[\overline{s}_k/\overline{X}]. \end{array} \qquad (W2)$$

Note that by structural induction the formulas $wp(c_0, \overline{X} = \overline{s}_{i+1})$ and $\neg wp(c_0, \mathbf{false})$ can be expressed as assertions because c_0 is a subcommand of the command c.

Note also that in the formula $(W2)$ we first compute the assertions $wp(c_0, \overline{X} = \overline{s}_{i+1})$ and $\neg wp(c_0, \mathbf{false})$, and then we apply the substitution $\overline{s}_i/\overline{X}$. To apply first the substitution $\overline{s}_i/\overline{X}$ and then to compute the weakest preconditions is not correct. Indeed, in particular, the formula $wp(c_0,\ \overline{s}_i{=}\overline{s}_{i+1})$ does not make sense.

Now let us consider the following Table 1 where, for $i = 0, \ldots, k$, we have indicated the values stored in the locations $X_1, \ldots, X_\ell$ for each state σ_i, for $i = 0, \ldots, k$, for $j{=}1, \ldots, \ell$.

state	locations X_1	...	X_ℓ
σ_0	$\overline{s}_0 = \langle s_{01}$	...	$s_{0\ell}\rangle$
σ_1	$\overline{s}_1 = \langle s_{11}$	...	$s_{1\ell}\rangle$
$\vdots$	$\vdots$	$\vdots$	
σ_k	$\overline{s}_k = \langle s_{k1}$	...	$s_{k\ell}\rangle$

TABLE 1. A sequence of states and the associated sequence of values stored in the locations $X_1, \ldots, X_\ell$.

We can encode this sequence of $k+1$ sequences, each of which is of length ℓ, row by row into a single sequence of the form:

$$\langle s_{01}, \ldots, s_{0\ell},\ s_{11}, \ldots, s_{1\ell},\ \ldots,\ s_{k1}, \ldots, s_{k\ell}\rangle$$

whose length is $\ell(k+1)$. Note that the value of ℓ is fixed for any given command **while** b **do** c_0 and assertion B.

Now by Lemma 4.19 on the following page and Lemma 4.21 on page 140, any finite sequence $x_0, \ldots, x_p$ of integers can be encoded by two natural numbers n and m such that there exists an *assertion* $\beta^{\pm}(n, m, i, x)$ which holds iff x is the i-th element of that sequence, for $i = 0, \ldots, p$.

Thus, the formula $(W2)$ can be written as the following assertion $(W3)$ where the variables $x_1, y_1, \ldots, x_\ell, y_\ell$ belong to **Intvar**.

$$\begin{array}{l}
\forall k \geq 0.\, \forall n, m \geq 0. \\
\big[\, \beta^{\pm}(n, m, 0, X_1) \wedge \ldots \wedge \beta^{\pm}(n, m, \ell{-}1, X_\ell) \wedge \\
\quad \forall i.\, (0 \leq i < k) \Rightarrow \\
\quad \big[\big(\forall x_1, \ldots, x_\ell. (\beta^{\pm}(n, m, i\ell, x_1) \wedge \ldots \wedge \beta^{\pm}(n, m, i\ell{+}\ell{-}1, x_\ell)) \\
\qquad\qquad \Rightarrow b[x_1/X_1, \ldots, x_\ell/X_\ell] \big) \\
\quad \wedge \big(\forall x_1, y_1, \ldots, x_\ell, y_\ell. \big(\beta^{\pm}(n, m, i\ell, x_1) \wedge \ldots \wedge \beta^{\pm}(n, m, i\ell{+}\ell{-}1, x_\ell) \\
\qquad\qquad\qquad \wedge \beta^{\pm}(n, m, (i{+}1)\ell, y_1) \wedge \ldots \wedge \beta^{\pm}(n, m, (i{+}1)\ell{+}\ell{-}1, y_\ell) \big) \\
\qquad\qquad\qquad \Rightarrow \big(\big(wp(c_0, (X_1 = y_1, \ldots, X_\ell = y_\ell)) \\
\qquad\qquad\qquad\qquad \wedge \neg wp(c_0, \mathbf{false}) \big) [x_1/X_1, \ldots, x_\ell/X_\ell] \big) \big) \big] \big] \\
\Rightarrow \big[\forall x_1 \ldots x_\ell. \big((\beta^{\pm}(n, m, k\ell, x_1) \wedge \ldots \wedge \beta^{\pm}(n, m, k\ell{+}\ell{-}1, x_\ell)) \\
\qquad\qquad \Rightarrow (b \vee B)[x_1/X_1, \ldots, x_\ell/X_\ell] \big) \big].
\end{array} \qquad (W3)$$

In the assertion $(W3)$ $x_1, \ldots, x_\ell$ are the values in the locations $X_1, \ldots, X_\ell$ in the state σ_i (that is, $\langle s_{i1}, \ldots, s_{i\ell}\rangle$ in Table 1), while $y_1, \ldots, y_\ell$ are the values in those locations in the state σ_{i+1} (that is, $\langle s_{i+1\,1}, \ldots, s_{i+1\,\ell}\rangle$ in Table 1).

If $\ell = 1$, that is, we have one location only, say X, then the assertion $(W3)$ reduces to the following assertion $(W3.1)$ where we have written: (i) x, instead of x_1, (ii) y, instead of y_1, and (iii) X, instead of X_1.

$$
\begin{array}{l}
\forall k \geq 0.\, \forall n, m \geq 0. \\
\big[\, \beta^{\pm}(n, m, 0, X) \wedge \\
\quad \forall i.\, (0 \leq i < k) \Rightarrow \big((\forall x.(\beta^{\pm}(n, m, i, x) \Rightarrow b[x/X])) \wedge \\
\qquad\qquad (\forall x, y.((\beta^{\pm}(n, m, i, x) \wedge \beta^{\pm}(n, m, i+1, y)) \\
\qquad\qquad\qquad \Rightarrow (wp(c_0, X = y) \wedge \neg wp(c_0, \mathbf{false}))[x/X])) \big) \,\big] \\
\Rightarrow \big[\forall x.(\beta^{\pm}(n, m, k, x) \Rightarrow (b \vee B)[x/X])\big].
\end{array}
\qquad (W3.1)
$$

This concludes the proof of the Expressiveness Theorem. □

REMARK 4.18. In the proof of the Expressiveness Theorem (see page 134) we made use of assertions with quantified integer variables and this is the reason why we have introduced the set **Intvar** of the integer variables.

Now we prove Lemma 4.19 and Lemma 4.21 that we have used in the proof of the Expressiveness Theorem.

In what follows, for all natural numbers x, n, p, we will write the formula

$x = n \ \mathbf{mod}\ p$

as an abbreviation for the assertion

$\exists h.\, h \geq 0 \ \wedge\ x + h \times p = n \ \wedge\ 0 \leq x < p$

where h is a natural number, or equivalently,

$\exists h.\, h \geq 0 \ \wedge\ x + h \times p = n \ \wedge\ h \times p \leq n \ \wedge\ (h+1) \times p > n$

where h is a natural number. Note that in these assertions we have used no other arithmetic operation besides $+$ and $\times$.

LEMMA 4.19. [**The Gödel β Predicate**] For all natural numbers n, m, i, x, let the assertion $\beta(n, m, i, x)$, called the *Gödel beta predicate*, be the equality formula $x = n \ \mathbf{mod}\ (1+(1+i)\times m)$, that is, for all natural numbers n, m, i, x,

$\beta(n, m, i, x) =_{def} \exists h.\, h \geq 0 \ \wedge\ x = n - h \times (1+(1+i)\times m) \ \wedge\ 0 \leq x < (1+(1+i)\times m).$

Any sequence $\langle n_0, \ldots, n_k \rangle$ of $k+1$ natural numbers can be encoded by two natural numbers m and n such that for $i = 0, \ldots, k$, $x = n_i$, that is, x is the i-th element of that sequence, iff $\beta(n, m, i, x)$ holds.

PROOF. Let us first define the natural numbers m and n for any given sequence $\langle n_0, \ldots, n_k \rangle$ of $k+1$ natural numbers. By definition, the number m is the factorial of the maximum number in the set $\{n_0, \ldots, n_k, k\}$. In order to define the number n we first define the following $3(k+1)$ values:

for $i = 0, \ldots, k$,

$p_i = 1 + (i+1) \times m$

$c_i = (p_0 \times \ldots \times p_k)/p_i$

$d_i =$ the unique number such that: (i) $0 \leq d_i < p_i$, and (ii) the integer division of $(c_i d_i)$ by p_i, denoted $(c_i d_i)\ \mathbf{div}\ p_i$, has remainder 1, that is there exists an integer q_i such that $c_i d_i = q_i p_i + 1$.

Note that the definition of d_i is well-formed because it can be shown that there is a unique value d_i which satisfies Conditions (i) and (ii).

Then we define n to be $\sum_{i=0}^{k} n_i\, c_i\, d_i$.

We leave it to the reader to check that for $i = 0, \ldots, k$, for any given sequence $\langle n_0, \ldots, n_k \rangle$ of $k+1$ natural numbers, the values of m and n defined as we have indicated above, are such that $\beta(n, m, i, x)$ holds iff x is the i-th element of the sequence $\langle n_0, \ldots, n_k \rangle$.

Note also that $\forall i, j,\ 0 \leq i, j \leq k,$ if $i \neq j$ then $\gcd(p_i, p_j) = 1$. □

The following example clarifies the constructions we have indicated in the proof of Lemma 4.19 on the preceding page.

EXAMPLE 4.20. Let us consider, for instance, the sequence $\langle 1, 3, 2 \rangle$. It is encoded by the numbers $m = 6$ and $n = 7127$, as we now show. The given sequence has 3 numbers and, thus, $k = 2$ and $m = (max\{1, 3, 2, 2\})! = 6$.

$p_0 = 1+1\times 6 = 7$. $p_1 = 1+2\times 6 = 13$. $p_2 = 1+3\times 6 = 19$.

$c_0 = 13\times 19 = 247$. $c_1 = 7\times 19 = 133$. $c_2 = 7\times 13 = 91$.

Now we list the values of:

- $r_0(n)$ (that is, the remainder of $(c_0 \times n)$ **div** p_0, which is $(247 \times n)$ **div** 7), for $1 \leq n < p_0$,
- $r_1(n)$ (that is, the remainder of $(c_1 \times n)$ **div** p_1, which is $(133 \times n)$ **div** 13), for $1 \leq n < p_1$, and
- $r_2(n)$ (that is, the remainder of $(c_2 \times n)$ **div** p_2, which is $(91 \times n)$ **div** 19), for $1 \leq n < p_2$.

n:	1	2	3	**4**	5	6
$r_0(n)$:	2	4	6	**1**	3	5

n:	1	2	3	4	5	6	7	8	**9**	10	11	12
$r_1(n)$:	3	6	9	12	3	5	8	11	**1**	4	7	10

n:	1	2	3	4	5	6	7	8	9	10	11	12	13	**14**	15	16	17	18
$r_2(n)$:	15	11	7	3	18	14	10	6	2	17	13	9	5	**1**	16	12	8	4

Within boxes we have singled out the values of n such that $r_i(n) = 1$, for $i = 0, 1, 2$.

Note that for $i = 0, 1, 2$, the function $\lambda n.r_i(n)$ defines a permutation of the sequence $\langle 1, 2, \ldots, p_i - 1 \rangle$. Thus, we have that:

$d_0 = 4$ (indeed, if we divide 247×4 by 7 we get the remainder 1),

$d_1 = 9$ (indeed, if we divide 133×9 by 13 we get the remainder 1), and

$d_2 = 14$ (indeed, if we divide 91×14 by 19 we get the remainder 1).

Then $n = 1\times 247\times 4 + 3\times 133\times 9 + 2\times 91\times 14 = 7127$. We have that:

for $i = 0$: $\beta(7127, 6, 0, 1)$ holds because 7127 **mod** $(1+(1+0)\times 6) = 1$,

for $i = 1$: $\beta(7127, 6, 1, 3)$ holds because 7127 **mod** $(1+(1+1)\times 6) = 3$, and

for $i = 2$: $\beta(7127, 6, 2, 2)$ holds because 7127 **mod** $(1+(1+2)\times 6) = 2$.

We can encode any finite sequence of natural numbers by a unique natural number using powers of prime numbers. For instance, the sequence $\langle 1, 3, 2 \rangle$ can be encoded

by $2^1 3^3 5^2$. However, this encoding cannot be realized by using $-$, $+$, and $\times$ only. In particular, in fact, in the language of assertions we cannot express the primitive recursion schema. Indeed, if it were the case, we could have reduced multiplication to addition and Peano Arithmetics (which is an undecidable theory) would have been reducible to Presburger Arithmetics (which is a decidable theory).

LEMMA 4.21. [**The $\beta^{\pm}$ Predicate**] A sequence of $k+1$ *integer* numbers can be encoded by two *natural* numbers m and n so that for $i = 0, \ldots k$, the i-th element of the sequence is the integer number x iff the following assertion $\beta^{\pm}(n, m, i, x)$ holds:

$$\beta^{\pm}(n, m, i, x) =_{def} \exists y.\, \big(\beta(n, m, i, y) \ \wedge \\ y \geq 0 \wedge \exists z.\big(z \geq 0 \wedge (y = 2z \Rightarrow x = z) \wedge (y = 2z-1 \Rightarrow x = -z)\big)\big)$$

where, as indicated in Lemma 4.19 on page 138, the formula $\beta(n, m, i, y)$ stands for

$$\exists h.\, h \geq 0 \ \wedge \ y + h \times (1 + (1+i) \times m) = n \ \wedge \ 0 \leq y < (1 + (1+i) \times m).$$

PROOF. Any sequence $\sigma = \langle n_0, \ldots, n_k \rangle$ of integer numbers is encoded by the values of m and n which are constructed, as indicated in the proof of Lemma 4.19 on page 138, starting from the sequence of natural numbers which is obtained from σ by replacing, for $i = 0, \ldots, k$, each integer number n_i by the natural number, denoted n_i^+, defined as follows:

$n_i^+ =$ if $n_i \geq 0$ then $2\, n_i$ else $(-2\, n_i) - 1$,

that is,

for n_i =	0	-1	1	-2	2	-3	3	-4	4	-5	5	...		
we have: n_i^+ =	0	1	2	3	4	5	6	7	8	9	10	...		□

Now we illustrate the fact that if we choose a particular language of assertions and a particular language of commands, it may be the case that the chosen language of assertions is *not* expressive w.r.t. the chosen language of commands.

Let us consider the language of assertions which consists of the following three assertions only: (i) $X < 0$, (ii) $X = 0$, and (iii) $X > 0$. In particular, for instance, the assertion $X = 0 \vee X > 0$ is not in the language. We have that this language of assertions is *not* expressive w.r.t. the language that consists of the command $X := X+1$ only. Indeed, $wp(c,\, X > 0)$ is $X \geq 0$, and it is not in the language of assertions we have considered.

Let us consider the language of assertions which consists of the assertions **true** and **false** only. This language of assertions is expressive w.r.t. the language **Com** of commands we have considered in Section 1 on page 117. Indeed, for all commands c, we have that:
(i) $wp(c, \mathbf{true})$ is **true** because both $\models \{\mathbf{true}\}\, c\, \{\mathbf{true}\}$ and $\models \{\mathbf{false}\}\, c\, \{\mathbf{true}\}$ hold, and
(ii) the value of $wp(c, \mathbf{false})$ depends on c.
(ii.1) For all commands c such that for all $\sigma \in \mathit{State}$, $[\![c]\!]\sigma \neq \bot$ (that is, c terminates starting from σ), we have that $wp(c, \mathbf{false})$ is **false**, because $\not\models \{\mathbf{false}\}\, c\, \{\mathbf{false}\}$ holds and $\not\models \{\mathbf{true}\}\, c\, \{\mathbf{false}\}$ does not hold.
(ii.2) For all commands c such that for all $\sigma \in \mathit{State}$, $[\![c]\!]\sigma = \bot$ (that is, c does not terminate starting from σ), we have that $wp(c, \mathbf{false})$ is **true**, because both $\models \{\mathbf{false}\}\, c\, \{\mathbf{false}\}$ and $\not\models \{\mathbf{true}\}\, c\, \{\mathbf{false}\}$ hold.

We have the following theorems.

THEOREM 4.22. **[Uniqueness of Weakest Preconditions]** Given a command $c \in \mathbf{Com}$ and an assertion $B \in \mathbf{Assn}$, we have that the assertion $wp(c, B)$ is unique up to logical equivalence.

PROOF. Let us consider a command $c \in \mathbf{Com}$ and an assertion $B \in \mathbf{Assn}$. Let us also assume that there are two formulas A_1 and A_2 which enjoy the properties of the weakest precondition of B w.r.t. c. In particular, for A_1 we have that:

$\models \{A_1\}\, c\, \{B\}$, and ($\alpha$.1)

for all assertions A and B, for all commands c, $\models \{A\}\, c\, \{B\}$ iff $\models A \Rightarrow A_1$. ($\alpha$.2)

Similarly, for A_2 we have that:

$\models \{A_2\}\, c\, \{B\}$, and ($\beta$.1)

for all assertions A and B, for all commands c, $\models \{A\}\, c\, \{B\}$ iff $\models A \Rightarrow A_2$. ($\beta$.2)

From (α.1) and (β.2) we have that $\models A_1 \Rightarrow A_2$, and from ($\alpha$.2) and ($\beta$.1) we have that $\models A_2 \Rightarrow A_1$. Thus, $\models A_2 \Leftrightarrow A_1$. □

LEMMA 4.23. **[Weakest Precondition Lemma]** For all commands $c \in \mathbf{Com}$, assertions $B \in \mathbf{Assn}$, if $A(c, B)$ is an assertion such that for all interpretations I, $A^I(c, B) = wp^I(c, B)$, then $\vdash \{A(c, B)\}\, c\, \{B\}$.

In particular, we have that $\vdash \{wp(c, B)\}\, c\, \{B\}$.

PROOF. We proceed by cases on the command c as in the proof of the Expressiveness Theorem 4.15 on page 134.

(*Case* $c = \mathbf{skip}$). By Theorem 4.15 we have that $B^I = wp^I(\mathbf{skip}, B)$. Since by hypothesis, $A^I(\mathbf{skip}, B) = wp^I(\mathbf{skip}, B)$, we get by transitivity, $A^I(\mathbf{skip}, B) = B^I$. By Fact 4.10 on page 132 we have:

$\models A(\mathbf{skip}, B) \Leftrightarrow B$. (1.1)

From (1.1) and $\vdash \{B\}\, \mathbf{skip}\, \{B\}$ (see rule $H1$), by the consequence rule $H6$, we get $\vdash \{A(\mathbf{skip}, B)\}\, \mathbf{skip}\, \{B\}$.

(*Case* $c = X := a$). By Theorem 4.15 we have that $(B[a/X])^I = wp^I(X := a, B)$. (Recall that $B[a/X]$ denotes the assertion B where all free occurrences of X are substituted by a.) Since by hypothesis, $A^I(X := a, B) = wp^I(X := a, B)$, we get by transitivity, $A^I(X := a,\ B) = (B[a/X])^I$. By Fact 4.10 we have:

$\models A(X := a, B) \Leftrightarrow B[a/X]$. (2.1)

From (2.1) and $\vdash \{B[a/X]\}\, X := a\, \{B\}$ (see rule $H2$), by the consequence rule $H6$, we get $\vdash \{A(X := a, B)\}\, X := a\, \{B\}$.

(*Case* $c = c_1; c_2$). By Theorem 4.15 we have that $A^I(c_1, A(c_2, B)) = wp^I(c_1; c_2, B)$. Since by hypothesis, $A^I(c_1; c_2, B) = wp^I(c_1; c_2, B)$, by transitivity, we get that $A^I(c_1; c_2, B) = A^I(c_1, A(c_2, B))$. By Fact 4.10 on page 132 we have:

$\models A(c_1; c_2, B) \Leftrightarrow A(c_1, A(c_2, B))$. (3.1)

By structural induction hypothesis, we have:

$\vdash \{A(c_2, B)\}\, c_2\, \{B\}$, where $A^I(c_2, B) = wp^I(c_2, B)$, and (3.2)

$\vdash \{A(c_1, A(c_2, B))\}\ c_1\, \{A(c_2, B)\}$, (3.3)

where $A^I(c_1, A(c_2, B)) = wp^I(c_1; c_2, B) = wp^I(c_1, A(c_2, B))$

From (3.1), (3.2), (3.3), rule $H3$, and rule $H6$, we get $\vdash \{A(c_1; c_2, B)\}\, c_1; c_2\, \{B\}$.

(*Case* $c =$ **if** b **then** c_1 **else** c_2). By Theorem 4.15 we have that:

$\big((b \wedge A(c_1, B)) \vee (\neg b \wedge A(c_2, B))\big)^I = wp^I(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2,\ B)$.

Since by hypothesis,
$A^I(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, B) = wp^I(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, B)$, we get, by transitivity,
$A^I(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, B) = \big((b \wedge A(c_1, B)) \vee (\neg b \wedge A(c_2, B))\big)^I$. By Fact on page 132 we have:

$$\models A(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, B) \Leftrightarrow (b \wedge A(c_1, B)) \vee (\neg b \wedge A(c_2, B)). \tag{4.1}$$

By structural induction hypothesis, we have:

$$\vdash \{A(c_1, B)\}\, c_1\, \{B\}, \text{ and} \tag{4.2}$$

$$\vdash \{A(c_2, B)\}\, c_2\, \{B\}. \tag{4.3}$$

Since $\models (b \wedge A(c_1, B)) \Rightarrow A(c_1, B)$ and $\models (\neg b \wedge A(c_2, B)) \Rightarrow A(c_2, B)$, from (4.1) and (4.2), by rule $H6$ we get, respectively:

$$\vdash \{b \wedge A(c_1, B)\}\, c_1\, \{B\}, \text{ and} \tag{4.2.1}$$

$$\vdash \{\neg b \wedge A(c_2, B)\}\, c_2\, \{B\}. \tag{4.3.1}$$

Let F stand for $(b \wedge A(c_1, B)) \vee (\neg b \wedge A(c_2, B))$. We have that: $(b \wedge F) \Leftrightarrow (b \wedge A(c_1, B))$ and $(\neg b \wedge F) \Leftrightarrow (\neg b \wedge A(c_2, B))$. Thus, from (4.2.1) and (4.3.1) we get, respectively:

$$\vdash \{b \wedge F\}\, c_1\, \{B\}, \text{ and} \tag{4.2.2}$$

$$\vdash \{\neg b \wedge F\}\, c_2\, \{B\}. \tag{4.3.2}$$

From (4.1), (4.2.2), (4.3.2), rule $H4$, and rule $H6$, we get:

$$\vdash \{A(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, B)\}\, \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2\, \{B\}.$$

(*Case* $c =$ **while** b **do** c_0). Let W denote the assertion $A(\textbf{while } b \textbf{ do } c_0, B)$. Since by hypothesis, $W^I = wp^I(\textbf{while } b \textbf{ do } c_0, B)$ we have that $\models W \Leftrightarrow wp(\textbf{while } b \textbf{ do } c_0, B)$. Now we show that:

$$\models \{W \wedge b\}\, c_0\, \{W\}, \text{ and} \tag{5.1}$$

$$\models (W \wedge \neg b) \Rightarrow B. \tag{5.2}$$

In order to prove Property (5.1) we need to show that for all I, σ, $I, \sigma \models W \wedge b$ implies $I, [\![c_0]\!]\sigma \models W$. Take any I and σ. Assume $I, \sigma \models W \wedge b$. We have that:

$$I, \sigma \models W \text{ and} \tag{5.1.1}$$

$$I, \sigma \models b. \tag{5.1.2}$$

From (5.1.1) $I, \sigma \models W$, by $\models W \Leftrightarrow wp(\textbf{while } b \textbf{ do } c_0, B)$, we get:

$$I, \sigma \models wp(\textbf{while } b \textbf{ do } c_0, B).$$

Hence, $I, [\![\textbf{while } b \textbf{ do } c_0]\!]\sigma \models B$. By the property of the function $[\![\_]\!]$ for commands, we have:

$$I, [\![\textbf{if } b \textbf{ then } (c_0; \textbf{while } b \textbf{ do } c_0) \textbf{ else skip}]\!]\sigma \models B. \tag{5.1.3}$$

From (5.1.3), since by (5.1.2), $I, \sigma \models b$ holds, we have:

$I, [\![c_0; \textbf{while } b \textbf{ do } c_0]\!]\sigma \models B$, that is, $I, [\![\textbf{while } b \textbf{ do } c_0]\!]([\![c_0]\!]\sigma) \models B$.

Therefore, $I, [\![c_0]\!]\sigma \models wp(\textbf{while } b \textbf{ do } c_0, B)$, and thus, $I, [\![c_0]\!]\sigma \models W$.

Now we prove Property (5.2). Take any I and σ. Assume $I, \sigma \models W \wedge \neg b$. We have that:

$I, \sigma \models W$ and (5.2.1)

$I, \sigma \models \neg b$. (5.2.2)

We have to show that $I, \sigma \models B$.

From (5.2.1) we get $I, \sigma \models wp(\textbf{while } b \textbf{ do } c_0, B)$. Thus, by definition of weakest precondition, we also get $I, [\![\textbf{while } b \textbf{ do } c_0]\!]\sigma \models B$. By the property of the function $[\![\_]\!]$ for commands, we have:

$I, [\![\textbf{if } b \textbf{ then } (c_0; \textbf{while } b \textbf{ do } c_0) \textbf{ else skip}]\!]\sigma \models B$. (5.2.3)

From (5.2.3), since by (5.2.2), $I, \sigma \models \neg b$, we get $I, [\![\textbf{skip}]\!]\sigma \models B$. Thus, $I, \sigma \models B$.

This completes the proofs of Properties (5.1) and (5.2).

Now, from (5.1) by definition of weakest precondition, we have that:

$\models (W \wedge b) \Rightarrow wp(c_0, W)$. (5.3)

By induction hypothesis (that is, by assuming that this Lemma 4.23 holds for c_0, which is a subcommand of the command **while** b **do** c_0) from (5.3) we get:

$\vdash \{wp(c_0, W)\}\, c_0\, \{W\}$. (5.4)

Thus by (5.3) and (5.4), by rule $H6$ we get:

$\vdash \{W \wedge b\}\, c_0\, \{W\}$. (5.5)

From (5.5), by rule $H5$ we get:

$\vdash \{W\}\, \textbf{while } b \textbf{ do } c_0\, \{W \wedge \neg b\}$. (5.6)

From (5.6), by (5.2) and rule $H6$, we get: $\vdash \{W\}\, \textbf{while } b \textbf{ do } c_0\, \{B\}$. □

THEOREM 4.24. **[Relative Completeness Theorem]** Given a procedure (which is necessarily not Turing computable) which for all assertions φ, checks whether or not $\models \varphi$ holds in Integer Arithmetics, we have that for all Hoare triples $\{A\}\, c\, \{B\}$, if $\models \{A\}\, c\, \{B\}$ then $\vdash \{A\}\, c\, \{B\}$.

PROOF. Let us assume $\models \{A\}\, c\, \{B\}$. By Lemma 4.23 on page 141 we get: (i) $\vdash \{wp(c, B)\}\, c\, \{B\}$. Thus, from $\models \{A\}\, c\, \{B\}$ and a property of the weakest precondition (see Theorem 4.13 (ii) on page 133), we get: (ii) $\models A \Rightarrow wp(c, B)$. Thus, from (i) and (ii), by the consequence rule $H6$ we get: $\vdash \{A\}\, c\, \{B\}$.

The procedure for checking for all assertions φ, whether or not $\models \varphi$ holds in Integer Arithmetics, is necessary for proving the premises of rules $H2$ and $H6$ of Hoare Calculus. □

THEOREM 4.25. **[Gödel Incompleteness Theorem]** The set $\{A \mid A \in \textbf{Assn}$ and $\models A$ holds in Integer Arithmetics$\}$ is not recursively enumerable.

PROOF. Let us consider the set $C\uparrow$ of commands which diverge starting from the state where all arithmetic variables are bound to 0. Formally,

$C\uparrow =_{def} \{c \mid \models wp(c, \textbf{false})$ and all locations in c are bound to $0\}$.

Note that: (i) there is nothing peculiar in our choice to have all locations bound to 0 because, by using assignments, we can bound every location to any value we desire, and (ii) having all locations bound to values, we may ask ourselves given any interpretation I, any state σ, any assertion A, whether or not $I, \sigma \models A$ holds.

We know that $C\uparrow$ is *not* recursively enumerable (r.e., for short), because the complement of $C\uparrow$ is r.e. and not recursive (recall Post Theorem which states that if a set and its complement are both r.e. then they are both recursive).

Now, let us assume by absurdum that $\{A \mid A \in \mathbf{Assn} \text{ and } \models A\}$ is r.e. Thus, we have a semidecision procedure that, in particular, tells us given any command c, where all locations are bound to 0, whether or not $\models wp(c, \mathbf{false})$ holds. Thus, $C\uparrow$ is r.e. and we get a contradiction. □

If we restrict the assertions by assuming that the set **Aop** of arithmetic operators (see page 125) is $\{+, -\}$, then we have that the set $\{A \mid A \in \mathbf{Assn} \text{ and } \models A\}$ is recursive. Analogously, if we take **Aop** to be $\{\times\}$, instead of $\{+, -\}$. These results follow from Presburger's result of the decidability of the first order theory of the natural numbers with the addition operation only.

If we restrict the assertions by avoiding quantifiers (and keeping **Aop** to be $\{+, -, \times\}$), that is, if we consider the set **Bexp** of the boolean expressions (see page 117), instead of the set **Assn**, we have that the set $\{b \mid b \in \mathbf{Bexp} \text{ and } \models b\}$ is not r.e. As we will see below, this result is a consequence of the following Matijasevic Theorem (see Theorem 4.26).

Recall that:
(i) for every arithmetic expression $a \in \mathbf{Aexp}$ (see page 117), for every state $\sigma \in State_\perp$ which assigns integers to locations (see page 118), we have that $\sigma \models a = 0$ iff $[\![a]\!]\sigma = 0$ (see page 126), and

(ii) the value of $[\![a]\!]\sigma$ does not depend on any interpretation I (that assigns integers to integer variables of **Intvar**), because neither quantifiers nor integer variables of **Intvar** occur in the boolean expressions of **Bexp** (see pages 117 and 125).

THEOREM 4.26. [**Matijasevic Theorem**] Let us consider the set **Aexp** of the arithmetic expressions. The set $\{a = 0 \mid a \in \mathbf{Aexp} \text{ and } \sigma \models a = 0 \text{ for some state } \sigma\}$ is not recursive and it is recursively enumerable.

As a consequence of Matijasevic Theorem and Post Theorem, we get that the set $\{\neg(a = 0) \mid a \in \mathbf{Aexp} \text{ and for all states } \sigma,\ \sigma \models \neg(a = 0)\}$ of boolean expressions in **Bexp** is not recursively enumerable.

Since $\neg(a = 0)$ is a particular boolean expression, we also get that the set $\{b \mid b \in \mathbf{Bexp} \text{ and } \models b\}$ is not recursively enumerable.

Now we present a proof system for deciding whether or not an equation e, where $a_1, a_2 \in \mathbf{Aexp}$ (see Figure 1 on the next page).

If an equation e can be derived by our proof system we write $\vdash e$. If an equation e of the form $a_1 = a_2$ is valid (that is, e holds for every state which assigns integers to the locations occurring in a_1 or a_2) we write $\models e$.

The proof system we will present is *sound* and *complete* in the sense that: (i) it derives only valid equations, and (ii) every valid equation has a derivation. Indeed, for every equation e of the form $a_1 = a_2$, where $a_1, a_2 \in \mathbf{Aexp}$, we have that $\vdash e$ iff $\models e$ [**19**, page 350].

In Figure 1 we assume that: (i) **op** ranges over $\{+, -, \times\}$, (ii) n ranges over the integers N, and (iii) a, possibly with subscripts, ranges over **Aexp**.

Note that when proving equations, a context-rule holds because we have the left and right congruence rule.

(reflexivity)	$a = a$
(symmetry)	$\dfrac{a_1 = a_2}{a_2 = a_1}$
(transitivity)	$\dfrac{a_1 = a_2 \qquad a_2 = a_3}{a_1 = a_3}$
(left and right congruence)	$\dfrac{a_1 = a_2}{a \ \mathbf{op}\ a_1 = a \ \mathbf{op}\ a_2} \qquad \dfrac{a_1 = a_2}{a_1 \ \mathbf{op}\ a = a_2 \ \mathbf{op}\ a}$
(associativity for $+$)	$(a_1{+}a_2){+}a_3 = a_1{+}(a_2{+}a_3)$
(associativity for $\times$)	$(a_1{\times}a_2){\times}a_3 = a_1{\times}(a_2{\times}a_3)$
(commutativity for $+$)	$a_1{+}a_2 = a_2{+}a_1$
(commutativity for $\times$)	$a_1{\times}a_2 = a_2{\times}a_1$
(distributivity)	$a_1{\times}(a_2{+}a_3) = (a_1{\times}a_2){+}(a_1{\times}a_3)$
(identity for $+$)	$0{+}a = a$
(identity for $\times$)	$1{\times}a = a$
(inverse for $+$)	$a{-}a = 0$
(negative numbers)	$-n = (-1){\times}n$
(subtraction)	$a_1{-}a_2 = a_1{+}((-1){\times}a_2)$
(successor)	$1{+}1 = 2, \quad 1{+}2 = 3, \quad 1{+}3 = 4, \ \ldots$

FIGURE 1. Proof system for equations in Integer Arithmetics. All variables are universally quantified in front.

In the proof system of Figure 1 all sums and products are computed by successive applications of the successor rule. For instance, the product 2×2 is computed by the following sequence of equalities:

$$2 \times 2 = (1+1) \times (1+1) = (1 \times 1) + (1 \times 1) + (1 \times 1) + (1 \times 1) = 1+1+1+1 = 4.$$

EXERCISE 4.27. By using the proof system of Figure 1, show that for all integer numbers a, b, and c, we have that:

(i) $4 + 3 = 7$, (ii) $-a + a = 0$,
(iii) $0 - a = -a$, (iv) $0 = -0$,
(v) $-(-a) = a$,
(vi) $a{-}b = c$ iff $a = b{+}c$ (that is, every subtraction can be reduced to a sum),
(vii) $-2 \times 3 = -6$, (viii) $0 \times a = 0$, and (ix) $(-1) \times (-1) = 1$. □

5. Proving Simple Programs Correct Using Hoare Triples

In this section we present the correctness proofs for some programs. Here and in what follows, we will free to refer to 'a command' also as 'a program'.

5.1. Summing up the elements of an array.

We first consider the following program *ArraySum* for summing up the n elements $A[0], \ldots A[n-1]$ of a given array A.

$\{n \geq 0\}$ Program: *ArraySum*

$i := 0;$
$sum := 0;$

$I \equiv \{0 \leq i \leq n \ \wedge \ sum = \sum_{j=0}^{i-1} A[j]\}$

while $i < n$ **do** $sum := sum + A[i];$
$i := i + 1;$
od

$\{sum = \sum_{j=0}^{n-1} A[j]\}$

We first show that the assertion I, which is the invariant of the **while-do** loop, is established upon initialization. We have to show the following implication:

$$n \geq 0 \ \Rightarrow \ \big(0 \leq i \leq n \ \wedge \ sum = \textstyle\sum_{j=0}^{i-1} A[j]\big)\,[0/i,\, 0/sum],$$

that is,

$$n \geq 0 \ \Rightarrow \ \big(0 \leq 0 \leq n \ \wedge \ 0 = \textstyle\sum_{j=0}^{-1} A[j]\big).$$

This implication is obvious. Then we show that the invariant I is kept during every execution of the body of the loop because we have that:

$$\big(0 \leq i \leq n \ \wedge \ sum = \textstyle\sum_{j=0}^{i-1} A[j]\big) \ \wedge \ (i < n) \ \Rightarrow$$
$$\big(0 \leq i \leq n \ \wedge \ sum = \textstyle\sum_{j=0}^{i-1} A[j]\big)\,[i+1/i, \ sum+A[i]/sum],$$

that is,

$$\big(0 \leq i \leq n \ \wedge \ sum = \textstyle\sum_{j=0}^{i-1} A[j]\big) \ \wedge \ (i < n) \ \Rightarrow$$
$$\big(0 \leq i+1 \leq n \ \wedge \ sum+A[i] = \textstyle\sum_{j=0}^{i} A[j]\big).$$

Also this implication is obvious.

Upon termination of the loop we get that the postcondition holds, because of the following implication holds:

$$\big(0 \leq i \leq n \ \wedge \ sum = \textstyle\sum_{j=0}^{i-1} A[j]\big) \ \wedge \ (i \not< n) \ \Rightarrow \ sum = \textstyle\sum_{j=0}^{n-1} A[j].$$

The termination of the program holds because at each execution of the body of the loop the value of $n-i$ decreases by one unit and cannot become negative. Thus, we have established the total correctness of the given program for summing up the elements of a given array.

5.2. Computing the power of a number.

Now we present a program for computing the n-th power of a given number x. The algorithm is based on the following equations where: (i) *even* and *odd* are primitive predicates over the natural numbers with the obvious meaning, and (ii) **div** 2 denotes the integer division by 2 (for instance, $7\,\mathbf{div}\,2$ returns 3, and $8\,\mathbf{div}\,2$ returns 4).

$$\begin{array}{ll} x^n = 1 & \text{if } n = 0 \\ x^n = (x^{n/2})^2 & \text{if } even(n) \\ x^n = x \times x^{n-1} & \text{if } odd(n) \end{array}$$

We have the following imperative program.

$\{n \geq 0\}$ Program: *Fast Exponentiation*

$k := n;\quad y := 1;\quad z := x;$

$I \equiv \{y \times z^k = x^n \ \wedge\ k \geq 0\}$

while $k \neq 0$ **do** **if** $odd(k)$ **then begin** $k := k-1;\quad y := y \times z$ **end**

$L \equiv \{y \times z^k = x^n \ \wedge\ k \geq 0 \ \wedge\ even(k)\}$

$k := k\,\mathbf{div}\,2;\quad z := z \times z;$

od

$\{y = x^n\}$

Partial correctness of our *Fast Exponentiation* program is established by the following four implications whose easy proofs are left to the reader.

The invariant I is established upon initialization because the following implication holds:

$$n \geq 0 \ \Rightarrow\ \big(y \times z^k = x^n \ \wedge\ k \geq 0\big)\,[n/k,\ 1/y,\ x/z],$$

that is,

$$n \geq 0 \ \Rightarrow\ \big(x^n = x^n \ \wedge\ n \geq 0\big).$$

The invariant I is kept during every execution of the body of the loop, because the following two implications hold:

$$\big(y \times z^k = x^n \ \wedge\ k \geq 0\big) \ \wedge\ (k \neq 0) \ \wedge\ odd(k) \ \Rightarrow$$
$$\big(y \times z^k = x^n \ \wedge\ k \geq 0 \ \wedge\ even(k)\big)\,[k-1/k,\ y \times z/y],$$

and

$$\big(y \times z^k = x^n \ \wedge\ k \geq 0 \ \wedge\ even(k)\big) \ \Rightarrow$$
$$\big(y \times z^k = x^n \ \wedge\ k \geq 0\big)\,[k\,\mathbf{div}\,2/k,\ z \times z/z].$$

Upon termination of the loop we get the postcondition because the following implication holds:

$$\big(y \times z^k = x^n \ \wedge\ k \geq 0\big) \ \wedge\ (k = 0) \ \Rightarrow\ (y = x^n).$$

The termination of the program is shown by the fact that the value of k is decreased at every execution of the loop body. This concludes the proof of the total correctness of our Fast Exponentiation program.

5.3. Computing Ackermann function.

Ackermann function is defined by the following equations.

$$\begin{array}{lll} Ack(0,n) & = n{+}1 & \text{for any } n \geq 0 \\ Ack(m{+}1,0) & = Ack(m{-}1,1) & \text{for any } m \geq 0 \\ Ack(m{+}1,n{+}1) & = Ack(m,\ Ack(m{+}1,n)) & \text{for any } m,n \geq 0 \end{array}$$

We may compute Ackermann function by using the following imperative program.

$\{m \geq 0\ \wedge n \geq 0\}$ Program: *Ackermann Function*

$stack := push(m,\ emptystack);\quad res := n;$

$I\ \equiv\ \{stack = emptystack\ \Rightarrow\ Ack(m,n) = res$
$\wedge\ stack \neq emptystack\ \Rightarrow\ \exists k, \exists n_1, \ldots n_k,$
$stack = push(n_1, \ldots, push(n_k,\ emptystack)\ldots)$
$\wedge\ Ack(m,n) = Ack(n_k, \ldots, Ack(n_1, res)\ldots)\}$

while $stack \neq emptystack$ **do**
$a := top(stack);\quad stack := pop(stack);$
if $a{=}0$ **then** $res := res{+}1$ **else**
if $res{=}0$ **then begin** $stack := push(a{-}1,\ stack);\ \ res := 1$ **end else**
else begin $stack := push(a,\ push(a{-}1,\ stack));\ \ res := res{-}1$ **end**
od

$\{res = Ack(m,n)\}$

We leave to the reader the proof of the partial correctness of this program which is done by checking that: (i) the invariant I is established upon initialization, (ii) the invariant I is kept during every execution of the body of the loop, and (iii) the invariant I establishes the postcondition $res = Ack(m,n)$ upon termination of the loop.

The termination of our program for Ackermann function can be shown by proving that at every execution of the loop body the measure μ which we now define, decreases according to a suitable ordering. For any stack with the k values $n_1, n_2, n_3, \ldots, n_{k-1}, n_k$, in this order (being n_1 the top of the stack), for any value of res, the function μ returns a multiset of pairs of integers, as follows:

$$\mu(push(n_1, push(n_2, push(n_3, \ldots, push(n_{k-1}, push(n_k,\ emptystack))\ldots))),\ res)$$
$$=_{def}\ \{\langle n_1, res\rangle, \langle n_2{+}1, 0\rangle, \langle n_3{+}1, 0\rangle, \ldots, \langle n_{k-1}{+}1, 0\rangle, \langle n_k{+}1, 0\rangle\}$$

Below we will prove (see Cases 1, 2, and 3) that at every execution of the loop body the measure μ decreases according to the ordering that we now define.

Let us consider the lexicographic order $>_{lex}$ on pairs of natural numbers which is defined as follows: for any natural numbers a_1, b_1, a_2, and b_2, we have that: $\langle a_1, b_1\rangle >_{lex} \langle a_2, b_2\rangle$ iff either $a_1 > a_2$ or ($a_1 = a_2$ and $b_1 > b_2$).

Let us also consider the multiset order $\gg_{lex}$ on finite multisets of pairs of natural numbers which is defined as follows: for any two finite multisets X and Y of pairs of natural numbers we have that: $X \gg_{lex} Y$ iff there exist the multisets A, B, and C

such that $X = A \uplus C$ and $Y = B \uplus C$ and $\forall q \in B$, $\exists p \in A$, $p >_{lex} q$, where $\uplus$ denotes the disjoint union of multisets (see also Definition 9.10 on page 42).

Case 1: $a=0$. We have that:

$\{\langle a, res\rangle, \langle n_2{+}1, 0\rangle, \langle n_3{+}1, 0\rangle, \ldots\} \gg_{lex} \{\langle n_2, res{+}1\rangle, \langle n_3{+}1, 0\rangle, \ldots\}$

because: (i) the pair $\langle a, res\rangle$ belongs to the multiset on the left hand side and it does not belong to the multiset on the right hand side, (ii) $\langle n_2{+}1, 0\rangle >_{lex} \langle n_2, res{+}1\rangle$, and (iii) all other pairs of the two multisets are equal.

Case 2: $a>0 \wedge res=0$. We have that:

$\{\langle a, res\rangle, \langle n_2{+}1, 0\rangle, \langle n_3{+}1, 0\rangle, \ldots\} \gg_{lex} \{\langle a{-}1, 1\rangle, \langle n_2{+}1, 0\rangle, \langle n_3{+}1, 0\rangle, \ldots\}$

because: (i) $\langle a, res\rangle >_{lex} \langle a{-}1, 1\rangle$, and (ii) all other pairs of the two multisets are equal.

Case 3: $a>0 \wedge res>0$. We have that:

$\{\langle a, res\rangle, \langle n_2{+}1, 0\rangle, \langle n_3{+}1, 0\rangle, \ldots\} \gg_{lex} \{\langle a, res{-}1\rangle, \langle a, 0\rangle, \langle n_2{+}1, 0\rangle, \langle n_3{+}1, 0\rangle, \ldots\}$

because: (i) $\langle a, res\rangle >_{lex} \langle a, res{-}1\rangle$, (ii) $\langle a, res\rangle >_{lex} \langle a, 0\rangle$ (recall that in this Case 3 $res>0$), and (iii) all other pairs of the two multisets are equal.

Since, as it is well known, the order $>$ is *well-founded* (that is, has no infinite descending sequences of elements of the form: $a_1 > a_2 > a_3 > \ldots$) iff the order $\gg_{lex}$ is well-founded, there is no an infinite sequence of executions of the loop body, and this concludes the proof of the total correctness of the given program for computing the Ackermann function.

5.4. Computing a linear recursive schema.

Let us consider the following recursively defined function $h : N \to D$, where: (i) N is the set of integers, (ii) D is a given set, (iii) a and b are two constants in D, and (iv) c is a function from D to D.

$$\begin{array}{ll} h(0) & = a \\ h(1) & = b \\ h(n{+}2) & = c(h(n)) \quad \text{for any } n \geq 0 \end{array}$$

We may compute the value of $h(n)$ for any argument $n \geq 0$ by using the following imperative program. Recall that: (i) by $\mathrm{x}\,\mathbf{div}\,2$ we denote the integer division of x by 2, so that, for instance, $7\,\mathbf{div}\,2 = 3$ and $6\,\mathbf{div}\,2 = 3$ (in general, when performing the integer division we discard the remainder, if it is different from 0), and (ii) $c^0(n) = n$ and $c^{m+1}(n) = c(c^m(n))$. We assume that $even(n)$ is true iff n is an even natural number.

$\{K \geq 0\}$ Program: *Linear Recursive Schema*

$n := K;$

$\{n \geq 0 \wedge n = K\}$

if $even(n)$ **then** $res := a$ **else begin** $res := b$; $n := n{-}1$ **end**;

$I \equiv \{n \geq 0 \wedge c^{n\,\mathbf{div}\,2}(res) = h(K) \wedge even(n)\}$

while $n>1$ **do** $n := n{-}2$; $res := c(res)$ **od**

$\{res = h(K)\}$

The partial correctness of this imperative program follows from the following four implications:

(1) $(n \geq 0 \wedge n = K \wedge even(n)) \Rightarrow (n \geq 0 \wedge c^{n \,\mathbf{div}\, 2}(a) = h(K) \wedge even(n))$

(2) $(n \geq 0 \wedge n = K \wedge \neg even(n)) \Rightarrow (n \geq 0 \wedge c^{(n-1) \,\mathbf{div}\, 2}(b) = h(K) \wedge even(n-1))$

(3) $(n \geq 0 \wedge c^{n \,\mathbf{div}\, 2}(res) = h(K) \wedge even(n) \wedge n > 1)$
$\Rightarrow (n-2 \geq 0 \wedge c^{(n-2) \,\mathbf{div}\, 2}(c(res)) = h(K) \wedge even(n-2))$

(4) $(n \geq 0 \wedge c^{n \,\mathbf{div}\, 2}(res) = h(K) \wedge even(n) \wedge n \leq 1) \Rightarrow res = h(K)$

The proofs of (1) and (2) can be done by induction on K.

The termination of the program follows from the fact that at each execution of the loop body the value of n decreases and that value cannot go below 0.

By taking different values for a, b, and c, we get different instantiations of the function h. Indeed,

(i) by taking $a = 0$, $b = 0$, and $c = \lambda n.n+1$, we get the function which for any $n \geq 0$, computes the integer division of n by 2,

(ii) by taking $a = true$, $b = false$, $c = \lambda x.\, not(x)$, we get the function which for any input $n \geq 0$, tells us whether or not n is even, that is, $h(n)$ is *true* iff $even(n)$ holds,

(iii) by taking $a = false$, $b = true$, $c = \lambda x.\, not(x)$, we get the function which for any input $n \geq 0$, tells us whether or not n is odd, that is, $h(n)$ is *false* iff $even(n)$ holds, and

(iv) by taking $a = \varepsilon$, $b = 1$, and $c = \lambda w.\, 10w$, we get the function which for any input $n \geq 0$, returns the word $(10)^{n \,\mathbf{div}\, 2}$ if $even(n)$ holds, and $(10)^{n \,\mathbf{div}\, 2}1$ if $even(n)$ does not hold.

5.5. Dividing integers using binary arithmetics.

Given the integer numbers $P \geq 0$ and $Q > 0$, we may compute the quotient q and the remainder r such that $P = Q \times q + r$ with $0 \leq r < Q$ by using the following program.

$\{P \geq 0 \wedge Q > 0\}$ Program: *Binary Division*

$r := P;\ \ m := Q;\ \ k := 0;\ \ q := 0;$

$I_1 \equiv \{P \geq 0 \wedge Q > 0 \wedge r = P \wedge m = Q \times 2^k \wedge k \geq 0 \wedge q = 0\}$

while $r \geq m$ **do** $m := m \times 2;\ \ k := k+1$ **od**;

$I_2 \equiv \{P \geq 0 \wedge Q > 0 \wedge P = Q \times q + r \wedge 0 \leq r < m \wedge m = Q \times 2^k \wedge k \geq 0 \wedge q \geq 0\}$

while $m \neq Q$ **do** $m := m \,\mathbf{div}\, 2;\ \ k := k-1;$

$I_3 \equiv \{P \geq 0 \wedge Q > 0 \wedge P = Q \times q + r \wedge 0 \leq r < 2 \times m \wedge m = Q \times 2^k \wedge k \geq 0 \wedge q \geq 0\}$

if $r \geq m$ **then begin** $r := r - m;\ \ q := q + 2^k$ **end**;

od

$\{P = Q \times q + r \wedge 0 \leq r < Q\}$

I_1 and I_2 are the invariants of the two while-do loops and I_3 is the assertion which holds after the execution of $k := k-1$. Note that when the statement $m := m \,\mathbf{div}\, 2$

is executed then m is an even number, because: (i) $m \neq Q$, and (ii) by I_2, we have that $m = Q \times 2^k$ (indeed, it can be shown that if $m \neq Q \wedge k \geq 0$, then $k \geq 1$).

Here is a trace of the values of r (initialized to P), m (initialized to Q), k (initialized to 0), and q (initialized to 0), while computing $P = 32$ divided by $Q = 6$. This division returns the quotient $q = 5$ and the remainder $r = 2$.

step	r	m	k	q
0	$P = 32$	$Q = 6$	0	0
1		12	1	
2		24	2	
3		48	3	
4	8	24	2	4 $(=0+2^2)$
5		12	1	
6	2	6	0	5 $(=0+2^2+2^0)$

The partial correctness of our program for binary division follows from the following implications:

1. $P \geq 0 \wedge Q > 0 \Rightarrow I_1[P/r,\ Q/m,\ 0/k,\ 0/q]$
2. $I_1 \wedge r \geq m \Rightarrow I_1[(m \times 2)/m,\ (k+1)/k]$
3. $I_1 \wedge r < m \Rightarrow I_2$
4. $I_2 \wedge m \neq Q \Rightarrow I_3[(m\ \mathbf{div}\ 2)/m,\ (k-1)/k]$
5. $I_3 \wedge r \geq m \Rightarrow I_2[(r-m)/r,\ (q+2^k)/q]$
6. $I_2 \wedge m = Q \Rightarrow P = Q \times q + r \wedge 0 \leq r < Q$

In the above implications, as usual,

(i) the generic formula $I[v/x]$ denotes I, where all occurrences of x have been replaced by v, and

(ii) $I[v_1/x_1, \ldots, v_n/x_n]$ is an abbreviation for $(\ldots(I[v_1/x_1])\ldots[v_n/x_n])$. In particular, $I_3[(m\ \mathbf{div}\ 2)/m,\ (k-1)/k]$ denotes I_3 where all occurrences of m have been replaced by $m\ \mathbf{div}\ 2$ and all occurrences of k have been replaced by $k-1$.

We leave it to the reader to prove those implications. In one of those proofs the following property is required: if $m \neq Q \wedge k \geq 0$ then $k \geq 1$.

The termination of the program *Binary Division* is a consequence of the following two facts:

(i) for the first while-do loop, we have that at each execution of the body, m increases and, eventually, it becomes larger than r, and

(ii) for the second while-do loop, we have that at each execution of the body, k is decreased by one unit until it becomes 0 and, thus, m becomes equal to Q (because $I_2 \Rightarrow m = Q \times 2^k$).

6. Verification Conditions on Annotated Commands

In this section we present a language of *annotated commands*, called **Acom**, and through those annotations, also called *verification conditions*, one can verify that any given annotated command is partially correct w.r.t. given assertions. Here are the annotated commands (see also page 126, where we have defined the set **Com** of commands).

Annotated Commands

$$c, c_0, c_1, c_2 \text{ over } \mathbf{Acom} \qquad c ::= \mathbf{skip} \mid X := a \mid c_1;\, X := a \mid c_1; \{D\}\, c_2 \mid \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \mid \mathbf{while}\ b\ \mathbf{do}\ \{D\}\, c_0$$

where $a \in \mathbf{Aexp}$, $b \in \mathbf{Bexp}$, and $D \in \mathbf{Assn}$.

In the case of the **while-do** command the assertion D is called the *invariant* of the **while-do**.

Now we present the so called *proof obligations*, which allow us to show the partial correctness of commands with respect to given preconditions and postconditions. A proof obligation is presented as a deductive rule of the form $\frac{Y_1 \quad \ldots \quad Y_n}{X}$ which tells us that in order to show X we need to show $Y_1, \ldots, Y_n$.

$$(H1) \quad \frac{\models A \Rightarrow B}{\{A\}\ \mathbf{skip}\ \{B\}}$$

$$(H2) \quad \frac{\models A \Rightarrow B[a/X]}{\{A\}\ X{:=}a\ \{B\}}$$

$$(H3.2) \quad \frac{\{A\}\ c_1\ \{B[a/X]\}}{\{A\}\ c_1\,; X{:=}a\ \{B\}}$$

$$(H3.2) \quad \frac{\{A\}\ c_1\ \{D\} \qquad \{D\}\ c_2\ \{B\}}{\{A\}\ c_1\,; \{D\}\ c_2\ \{B\}}$$

$$(H4) \quad \frac{\{A \wedge b\}\ c_1\ \{B\} \qquad \{A \wedge \neg b\}\ c_2\ \{B\}}{\{A\}\ \mathbf{if}\ b\ \mathbf{then}\ c_1\, \mathbf{else}\ c_2\ \{B\}}$$

$$(H5) \quad \frac{\models A \Rightarrow D \qquad \{D \wedge b\}\ c_0\ \{D\} \qquad \models (D \wedge \neg b) \Rightarrow B}{\{A\}\ \mathbf{while}\ b\ \mathbf{do}\ \{D\}\ c_0\ \{B\}} \quad (D \text{ is the invariant})$$

These proof obligations are derived from the rules of the Hoare Calculus (see Section 4.3 on page 128).

The three premises of rule $(H5)$ are called *initialization*, *looping*, and *exit*, respectively.

The proof obligations $(H1)$–$(H5)$ are *sufficient conditions*. However, they are not necessary conditions. Indeed, let us consider the following Hoare triple:

$$\{\mathbf{true}\}\ \mathbf{while\ false\ do}\ \{\mathbf{false}\}\ \mathbf{skip}\ \{\mathbf{true}\} \qquad (\alpha)$$

where **false** is the invariant of the **while-do** command. Since for any command c, the triple $\{\mathbf{true}\}\, c\, \{\mathbf{true}\}$ holds, we have that (α) is a valid triple.

However, the initialization condition $\models \mathbf{true} \Rightarrow \mathbf{false}$ does not hold and, thus, the invariant **false** does not allow us to establish the validity of the triple (α).

Note that, since the calculus of the Hoare triples is undecidable, there is no algorithm that for any given command c and assertions A and B, (i) terminates, and (ii) constructs for all **while-do** commands occurring in c, the associated invariants which allow us to establish the validity of the triple $\{A\}\, c\, \{B\}$.

Actually, the calculus of the Hoare triples is not even semidecidable, because the triple $\{\mathbf{true}\}\, c\, \{\mathbf{false}\}$ holds iff the command c does not terminate, and non-termination is undecidable and not semidecidable.

7. Semantics of While-Do Loops

In what follows we give two definitions (see Definition 7.1 below and Definition 7.6 on page 156) of the semantics of the **while-do** loops and we show that they are equivalent.

In Lemma 7.3 on the following page we show that these definitions are also equivalent to the semantic equations of Section 3.3 on page 123.

DEFINITION 7.1. [**Denotational Semantics of while-do**] We stipulate that for all boolean expressions $b \in \mathbf{Bexp}$, for all commands $c_0 \in \mathbf{Com}$, and for all states $\sigma, \sigma' \in \mathit{State}_\perp$,

$[\![\mathbf{while}\ b\ \mathbf{do}\ c_0]\!]\sigma = \sigma'$ iff

$$\begin{array}{lll}
 & \sigma \neq \perp \wedge [\![b]\!]\sigma = \mathit{false} \wedge \sigma = \sigma' & (1)\\
\vee & \exists k{>}0.\ \exists \sigma_0 \ldots \sigma_k \in \mathit{State}^k \cdot \mathit{State}_\perp. & (2)\\
 & \quad \sigma_0 = \sigma \wedge \sigma_k = \sigma' \wedge \Big(\forall i,\ 0{\leq} i{<}k.\ ([\![b]\!]\sigma_i = \mathit{true} \wedge [\![c_0]\!]\sigma_i = \sigma_{i+1})\Big) \wedge & \\
 & \qquad ((\ [\![b]\!]\sigma_k = \mathit{false} \wedge \sigma_k \neq \perp) & (2.1)\\
 & \quad \vee \quad \sigma_k = \perp) & (2.2)\\
\vee & \sigma = \sigma' = \perp & (3)\\
\vee & \sigma' = \perp \wedge & \\
 & \forall k{\geq}0.\ \forall \sigma_0 \ldots \sigma_k \in \mathit{State}^{k+1}. & (4)\\
 & \quad \sigma_0 = \sigma \wedge \Big(\forall i,\ 0{\leq} i{<}k.\ ([\![b]\!]\sigma_i = \mathit{true} \wedge [\![c_0]\!]\sigma_i = \sigma_{i+1})\Big) &
\end{array}$$

REMARK 7.2. Cases (1), (2.1), (2.2), (3), and (4) of this Definition 7.1 are *mutually exclusive* and *exhaustive* and, thus, Definition 7.1 defines a function, called $[\![\mathbf{while}\ b\ \mathbf{do}\ c_0]\!]$, from $\mathit{State}_\perp$ to $\mathit{State}_\perp$.

Note that in Case (2) we have that $k{\neq}0$. Indeed, if $k{=}0$ then Case (2.1) reduces to Case (1), and Case (2.2) reduces to Case (3). Thus, if we do *not* insist on having mutually exclusive cases, in Case (2) we may write '$\exists k{\geq}0$', instead of '$\exists k{>}0$'.

The initial state σ is equal to $\bot$ in Case (3), while it is different from $\bot$ in Cases (1), (2.1), (2.2), and (4).

Case (1) holds when the body of the **while-do** command is never executed and, thus, the **while-do** command is equivalent to **skip**.

Case (2.1) holds when the body of the **while-do** command is executed at least once and the last execution of its body terminates.

Case (2.2) holds when the body of the **while-do** command is executed at least once and the last execution of its body does *not* terminate, that is, the execution of c_0 starting from the state σ_{k-1}, does not terminate.

Case (4) holds when the body of the **while-do** command is executed an unbounded number of times because the value of boolean expression b is always *true*, that is, for all $i \geq 0$, (1) the value of b in the state σ_i is *true*, and (2) every execution of c_0 starting from the state σ_i, terminates leading to the state σ_{i+1}. □

LEMMA 7.3. For all boolean expressions $b \in \mathbf{Bexp}$, for all commands $c_0 \in \mathbf{Com}$, and for all states $\sigma, \sigma' \in \mathit{State}_\bot$, we have that $[\![\mathbf{while}\ b\ \mathbf{do}\ c_0]\!]\sigma = \sigma'$ holds according to the semantic equations given in Section 3.3 on page 123 iff $[\![\mathbf{while}\ b\ \mathbf{do}\ c_0]\!]\sigma = \sigma'$ holds according to Definition 7.1 on the preceding page.

PROOF. The proof is done by cases considering the possible values of σ (whether $\sigma = \bot$ or $\sigma \neq \bot$) and b (whether $[\![b]\!]\sigma = \mathit{true}$ or $[\![b]\!]\sigma = \mathit{false}$).

(Case 1) Assume that $\sigma \neq \bot$ and $[\![b]\!]\sigma = \mathit{false}$. According to the semantic equations, $\sigma' = [\![\mathbf{while}\, b\, \mathbf{do}\, c_0]\!]\sigma = \sigma$. We also have that $\sigma' = \sigma$ according to Case 1 of Definition 7.1.

(Case 2.1) Assume that $\sigma \neq \bot$ and there exists $k > 0$ such that for all i, with $0 \leq i < k$, $[\![b]\!]([\![c_0]\!]^i\sigma) = \mathit{true}$ and $[\![b]\!]([\![c_0]\!]^k\sigma) = \mathit{false}$ and $[\![c_0]\!]^k\sigma = \sigma'$. According to the semantic equations, $\sigma' = [\![\mathbf{while}\, b\, \mathbf{do}\, c_0]\!]\sigma = (\bigsqcup_{i\geq 0} \tau^i(\lambda\sigma.\bot))\sigma$ where:

$\tau =_{def} \lambda f.\lambda\sigma.\mathit{cond}([\![b]\!]\sigma,\ f([\![c_0]\!]\sigma),\ \sigma)$.

We have that:

$\tau^0(\lambda\sigma.\bot)\sigma = \bot$

$\tau^1(\lambda\sigma.\bot)\sigma = \mathit{cond}([\![b]\!]\sigma, \bot, \sigma)$

$\tau^2(\lambda\sigma.\bot)\sigma = \mathit{cond}([\![b]\!]\sigma,\ \mathit{cond}([\![b]\!]([\![c_0]\!]\sigma), \bot, [\![c_0]\!]\sigma)\,, \sigma)$

$\tau^3(\lambda\sigma.\bot)\sigma = \mathit{cond}([\![b]\!]\sigma,\ \mathit{cond}([\![b]\!]([\![c_0]\!]\sigma), \mathit{cond}([\![b]\!]([\![c_0]\!]^2\sigma), \bot, [\![c_0]\!]^2\sigma), [\![c_0]\!]\sigma)\,, \sigma)$

...

In order to highlight the iterative structure of the above right hand sides, let us rewrite them by using the following abbreviations:

(i) (a, b, c), instead of $\mathit{cond}(a, b, c)$,
(ii) b, instead of $[\![b]\!]$,
(iii) for all $i > 0$, $i\sigma$, instead of $[\![c_0]\!]^i\sigma$, and
(iv) σ, instead of $[\![c_0]\!]^0\sigma$.

Thus, for instance, $[\![b]\!]\sigma$ is rewritten as $b\sigma$, and for $i > 0$, $[\![b]\!]([\![c_0]\!]^i\sigma)$ is rewritten as $bi\sigma$. We get:

$$
\begin{array}{lcccccccccc}
\tau^0(\lambda\sigma.\bot)\sigma & = & & & & & \bot & & & & \\
\tau^1(\lambda\sigma.\bot)\sigma & = & & & & (b\sigma, & \bot, & \sigma) & & & \\
\tau^2(\lambda\sigma.\bot)\sigma & = & & & (b\sigma, & (b1\sigma, & \bot, & 1\sigma), & \sigma) & & \\
\tau^3(\lambda\sigma.\bot)\sigma & = & & (b\sigma, & (b1\sigma, & (b2\sigma, & \bot, & 2\sigma), & 1\sigma), & \sigma) & \\
\quad\ldots & & & & & & \ldots & & & & \\
\tau^{i+1}(\lambda\sigma.\bot)\sigma & = & (b\sigma, & (b1\sigma, & (b2\sigma, & (\ldots, & (bi\sigma, & \bot, & i\sigma), & \ldots), & 2\sigma), & 1\sigma), & \sigma) \\
\quad\ldots & & & & & & \ldots & & & & \\
\end{array}
$$

By using the same abbreviations, the hypotheses holding in our case are rewritten as follows: there exists $k>0$ such that for all i, with $0\leq i<k$, $bi\sigma = \mathit{true}$ and $bk\sigma = \mathit{false}$ and $k\sigma = \sigma'$.

By looking at the above values of $\tau^i(\lambda\sigma.\bot)\sigma$, for $i \geq 0$, one can easily show by induction that, for all $k > 0$, for all i, with $i > k$, $\tau^i(\lambda\sigma.\bot)\sigma = k\sigma$, that is, $\tau^i(\lambda\sigma.\bot)\sigma = [\![c_0]\!]^k\sigma$. Therefore, we get the following table:

$i =$	0	1	$\ldots$	$k-1$	k	$k+1$	$k+2$	$\ldots$
$[\![b]\!]([\![c_0]\!]^i\sigma) =$	*true*	*true*	$\ldots$	*true*	*false*			
$\tau^i(\lambda\sigma.\bot)\sigma =$	$\bot$	$\bot$	$\ldots$	$\bot$	$\bot$	$[\![c_0]\!]^k\sigma$	$[\![c_0]\!]^k\sigma$	$\ldots$

Thus, $\sigma' = (\bigsqcup_{i\geq 0}\tau^i(\lambda\sigma.\bot))\sigma = [\![c_0]\!]^k\sigma$.

We also have that $\sigma' = [\![c_0]\!]^k\sigma$ according to Case 2.1 of Definition 7.1.

(Case 2.2) Assume that $\sigma \neq \bot$ and there exists $k > 0$ such that for all i, with $0 \leq i < k$, $[\![b]\!]([\![c_0]\!]^i\sigma) = \mathit{true}$ and $[\![c_0]\!]^k\sigma = \bot$. According to the semantic equations, $\sigma' = [\![\mathbf{while}\, b\, \mathbf{do}\, c_0]\!]\sigma = (\bigsqcup_{i\geq 0}\tau^i(\lambda\sigma.\bot))\sigma$. The proof is similar to that of Case 2.1. However, in this case, since $[\![c_0]\!]^k\sigma = \bot$, we have that: (1) for all $i \geq k$, $[\![b]\!]([\![c_0]\!]^i\sigma) = \mathit{true}$ (recall that for all b, $[\![b]\!]\bot = \mathit{true}$), and (2) for all $i > k$, $\tau^i(\lambda\sigma.\bot)\sigma = \bot$ (recall that for all commands c, $[\![c]\!]\bot = \bot$).

Thus, we have the following table:

$i =$	0	1	$\ldots$	$k-1$	k	$k+1$	$k+2$	$\ldots$
$[\![b]\!]([\![c_0]\!]^i\sigma) =$	*true*	*true*	$\ldots$	*true*	*true*	*true*	*true*	$\ldots$
$\tau^i(\lambda\sigma.\bot)\sigma =$	$\bot$	$\bot$	$\ldots$	$\bot$	$\bot$	$\bot$	$\bot$	$\ldots$

Thus, $\sigma' = (\bigsqcup_{i\geq 0}\tau^i(\lambda\sigma.\bot))\sigma = \bot$.

We also have that $\sigma' = \bot$ according to Case 2.2 of Definition 7.1.

(Case 3) Assume that $\sigma = \bot$. According to the semantic equations, we have that $\sigma' = [\![\mathbf{while}\, b\, \mathbf{do}\, c_0]\!]\sigma = \bot$, because $[\![c]\!]\bot = \bot$ for all commands c.

We also have that $\sigma' = \bot$ according to Case 3 of Definition 7.1.

(Case 4) Assume that $\sigma \neq \bot$ and for all $i \geq 0$, $[\![b]\!]([\![c_0]\!]^i\sigma) = \mathit{true}$ and $[\![c_0]\!]^i\sigma \neq \bot$. According to the semantic equations, $\sigma' = [\![\mathbf{while}\, b\, \mathbf{do}\, c_0]\!]\sigma = (\bigsqcup_{i\geq 0} \tau^i(\lambda\sigma.\bot))\sigma$.

Let us show, by induction on i, that for all $i \geq 0$, $\tau^i(\lambda\sigma.\bot) = \lambda\sigma.\bot$.
The basis case is obvious. The inductive step is as follows.

$\tau^{i+1}(\lambda\sigma.\bot) = \tau(\tau^i(\lambda\sigma.\bot)) =$ {by induction hypothesis} $=$

$= \tau(\lambda\sigma.\bot) = \lambda\sigma.\mathit{cond}([\![b]\!]\sigma, \bot, \sigma) =$ {because for all $i \geq 0$, $[\![b]\!]([\![c_0]\!]^i\sigma) = \mathit{true}$} $=$

$= \lambda\sigma.\bot$.

Now, since for all $i \geq 0$, $\tau^i(\lambda\sigma.\bot) = \lambda\sigma.\bot$, we get that $\sigma' = \bot$.

We also have that $\sigma' = \bot$ according to Case 4 of Definition 7.1. □

REMARK 7.4. We have assumed that the semantics function $[\![\_]\!]$ for commands is a *total* function from $\mathbf{Com} \times \mathit{State}_\bot$ to $\mathit{State}_\bot$. We could have assumed that it were a *partial* function from $\mathbf{Com} \times \mathit{State}$ to State. In that case, in order to be consistent with the semantic equations given in Section 3.3, in Definition 7.1 on page 153 we should have limited ourselves to Cases (1) and (2.1) only, because in those cases $\sigma' \neq \bot$. In the other cases, that is, Cases (2.2), (3), and (4), $[\![\mathbf{while}\ b\ \mathbf{do}\ c_0]\!]\sigma$ should have been undefined and, indeed, we have that $\sigma' = \bot$. □

REMARK 7.5. Since the semantic function for commands is strict and for every boolean expression b, $[\![b]\!]\bot = \mathit{true}$, we have that Case (2.2) and Case (3) of Definition 7.1 on page 153 are the instances for $0 < k = m$ and $0 = k = m$, respectively, of the more general case in which there exist: (1) two integers k and m, with $0 \leq k \leq m$, and (2) a sequence of states $\sigma_0 \ldots \sigma_{k-1}\sigma_k \ldots \sigma_m \in \mathit{State}^k \cdot \mathit{State}_\bot^{m-k+1}$ such that: (2.1) for $i = 0, \ldots, k-1$, we have that $\sigma_i \neq \bot$, and (2.2) for $i = k, \ldots, m$, we have that $\sigma_i = \bot$. □

DEFINITION 7.6. [**Weakest Precondition of while-do**] We stipulate that for all states $\sigma \in \mathit{State}_\bot$, for all interpretations $I \in \mathbf{Intvar} \to N$, for all boolean expressions $b \in \mathbf{Bexp}$, for all commands $c_0 \in \mathbf{Com}$, and for all assertions $B \in \mathbf{Assn}$,

$\sigma \in wp^I(\mathbf{while}\ b\ \mathbf{do}\ c_0,\ B)$ iff

$$\begin{array}{lll} & \sigma = \bot & \left.\begin{array}{l}\\\end{array}\right] (\alpha) \\ \vee & \forall k \geq 0.\, \forall \sigma_0 \ldots \sigma_k \in \mathit{State}^k \cdot \mathit{State}_\bot.\ (\beta_\Rightarrow) & \left.\begin{array}{l}\\\end{array}\right] (\beta) \end{array}$$

where the formula $(\beta_\Rightarrow)$ is the following implication:

$$\begin{array}{l} \Big[\, \sigma_0 = \sigma \ \wedge\ \Big(\forall i,\, 0 \leq i < k.\, ([\![b]\!]\sigma_i = \mathit{true} \wedge [\![c_0]\!]\sigma_i = \sigma_{i+1})\Big)\, \Big] \\ \Rightarrow\ (\ [\![b]\!]\sigma_k = \mathit{true}\ \vee\ I, \sigma_k \models B\ ) \end{array}$$

The following two theorems, Theorem 7.7 on the next page and Theorem 7.9 on page 158, show that Definition 7.1 on page 153 and Definition 7.6 are equivalent in the sense that:
for all boolean expressions $b \in \mathbf{Bexp}$, for all commands $c_0 \in \mathbf{Com}$, for all states $\sigma \in \mathit{State}_\bot$, for all interpretations $I \in \mathbf{Intvar} \to N$, for all assertions $B \in \mathbf{Assn}$,

$\sigma \in wp^I(\mathbf{while}\ b\ \mathbf{do}\ c_0,\ B)$

iff there exists $\sigma' \in \mathit{State}_\bot$ such that $[\![\mathbf{while}\ b\ \mathbf{do}\ c_0]\!]\sigma = \sigma' \ \wedge\ I, \sigma' \models B$.

THEOREM 7.7. [**Equivalence of Semantics of while-do. Part 1**] For all boolean expressions $b \in \mathbf{Bexp}$, for all commands $c_0 \in \mathbf{Com}$, for all states $\sigma, \sigma' \in State_\perp$, for all interpretations $I \in \mathbf{Intvar} \to N$, for all assertions $B \in \mathbf{Assn}$,

if $[\![\mathbf{while}\ b\ \mathbf{do}\ c_0]\!]\sigma = \sigma' \ \wedge\ I, \sigma' \models B$

then $\sigma \in wp^I(\mathbf{while}\ b\ \mathbf{do}\ c_0,\ B)$.

PROOF. By cases according to Definition 7.1.

(*Case* 1). Since $[\![b]\!]\sigma = \mathit{false}$ and $\sigma_0 = \sigma$, we have that $[\![b]\!]\sigma_0 = \mathit{false}$. For any $k \geq 0$ and any sequence $\sigma_0 \ldots \sigma_k$, the implication $(\beta_\Rightarrow)$ holds because its premise is false. Thus, (β) holds and $\sigma \in wp^I(\mathbf{while}\ b\ \mathbf{do}\ c_0,\ B)$.

(*Case* 2). It is enough to show that $\forall k \geq 0.\, \forall \sigma_0 \ldots \sigma_k \in State^k \cdot State_\perp.\, (\beta_\Rightarrow)$. In order to do so, let us take any $k \geq 0$ and any sequence $\sigma_\beta =_{def} \sigma_0 \ldots \sigma_k \in State^k \cdot State_\perp$ such that $\sigma_0 = \sigma$ and $\forall i, 0 \leq i < k.\, ([\![b]\!]\sigma_i = \mathit{true} \wedge [\![c_0]\!]\sigma_i = \sigma_{i+1})$.

For this case we look at Cases (2.1) and (2.2) separately.

(*Case* 2.1). In this case, by hypothesis, there exists a sequence $\sigma_\alpha =_{def} \sigma_0 \ldots \sigma_s \in State^s \cdot State_\perp$ such that $s > 0$ and $\sigma_0 = \sigma$ and $\sigma_s = \sigma'$ and $\forall i, 0 \leq i < k.\, ([\![b]\!]\sigma_i = \mathit{true} \wedge [\![c_0]\!]\sigma_i = \sigma_{i+1})$ and $[\![b]\!]\sigma_s = \mathit{false}$ and $\sigma_s \neq \perp$.

Let m be the minimum between k and s. Since the semantic function $[\![\_]\!]$ for commands is indeed a function, we have that for $j = 0, \ldots, m$, the prefixes of length j of the sequences σ_α and σ_β are equal. Now, we have that:

(2.1.1) for $0 \leq k < s$, $(\beta_\Rightarrow)$ holds for σ_β because $[\![b]\!]\sigma_k = \mathit{true}$,

(2.1.2) for $k = s$, $(\beta_\Rightarrow)$ holds for σ_β because $\sigma_s = \sigma'$ and $[\![\mathbf{while}\ b\ \mathbf{do}\ c_0]\!]\sigma = \sigma_s \neq \perp$, and $I, \sigma' \models B$ (by hypothesis), and

(2.1.3) for $k > s$, $(\beta_\Rightarrow)$ holds for σ_β because $[\![b]\!]\sigma_s = \mathit{false}$ and thus, the premise of $(\beta_\Rightarrow)$ is false.

Thus, (β) holds and $\sigma \in wp^I(\mathbf{while}\ b\ \mathbf{do}\ c_0,\ B)$.

(*Case* 2.2) is similar to Case (2.1). In this case, by hypothesis, there exists a sequence $\sigma_\alpha =_{def} \sigma_0 \ldots \sigma_s \in State^s \cdot State_\perp$ such that $s > 0$ and $\sigma_0 = \sigma$ and $\sigma_s = \sigma'$ and $\forall i, 0 \leq i < k.\, ([\![b]\!]\sigma_i = \mathit{true} \wedge [\![c_0]\!]\sigma_i = \sigma_{i+1})$ and $\sigma_s = \perp$.

Let m be the minimum between k and s. Since the semantic function $[\![\_]\!]$ for commands is indeed a function, we have that for $j = 0, \ldots, m$, the prefixes of length j of the sequences σ_α and σ_β are equal. Now, we have that:

(2.2.1) for $0 \leq k < s$, $(\beta_\Rightarrow)$ holds for σ_β because $[\![b]\!]\sigma_k = \mathit{true}$,

(2.2.2) for $k = s$, $(\beta_\Rightarrow)$ holds for σ_β because $\sigma_s = \sigma' = \perp$ and $[\![\mathbf{while}\ b\ \mathbf{do}\ c_0]\!]\sigma = \sigma_s = \perp$, and $I, \perp \models B$ (by definition of $\models$), and

(2.2.3) for $k > s$, $(\beta_\Rightarrow)$ holds for σ_β because: (i) $\sigma_s = \perp$, (ii) for all $j \geq s$, $\sigma_j = \perp$ (because for all $j \geq s$, $[\![c_0]\!]\sigma_j = \sigma_{j+1}$ and the semantic function $[\![\_]\!]$ for commands is a strict function), and (iii) $I, \perp \models B$ (by definition of $\models$) and, thus, the conclusion of $(\beta_\Rightarrow)$ holds.

Thus, (β) holds and $\sigma \in wp^I(\mathbf{while}\ b\ \mathbf{do}\ c_0,\ B)$.

(*Case* 3). In this case $\sigma = \sigma' = \perp$ and for any b, c_0, and B, we have that $\perp \in wp^I(\mathbf{while}\ b\ \mathbf{do}\ c_0,\ B)$.

(*Case* 4). In this case, by hypothesis, we have that:

$\forall k \geq 0.\, \forall \sigma_0 \ldots \sigma_k \in \mathit{State}^{k+1}.\ \sigma_0 = \sigma \wedge \forall i \geq 0.\, (\llbracket b \rrbracket \sigma_i = \mathit{true} \wedge \llbracket c_0 \rrbracket \sigma_i = \sigma_{i+1}).$ (†)

It is enough to show that $\forall k \geq 0.\, \forall \sigma_0 \ldots \sigma_k \in \mathit{State}^k \cdot \mathit{State}_\perp.\ (\beta_\Rightarrow)$. In order to do so, let us take any $k \geq 0$ and any sequence $\sigma_0 \ldots \sigma_k \in \mathit{State}^k \cdot \mathit{State}_\perp$ such that $\sigma_0 = \sigma$ and $\forall i, 0 \leq i < k.\, (\llbracket b \rrbracket \sigma_i = \mathit{true} \wedge \llbracket c_0 \rrbracket \sigma_i = \sigma_{i+1})$. Since (†) holds and the semantic function $\llbracket \_ \rrbracket$ for commands is indeed a function, we have that $\sigma_k \in \mathit{State}$. By (†) we also have that $\llbracket b \rrbracket \sigma_k = \mathit{true}$. Thus, the conclusion of $(\beta_\Rightarrow)$ holds. □

REMARK 7.8. The above Theorem 7.7 can also be stated as follows:
for all interpretations $I \in \mathbf{Intvar} \to N$, for all states $\sigma \in \mathit{State}_\perp$, for all assertions $B \in \mathbf{Assn}$,

if $\llbracket \mathbf{while}\ b\ \mathbf{do}\ c_0 \rrbracket \sigma \in B^I$ then $\sigma \in wp^I(\mathbf{while}\ b\ \mathbf{do}\ c_0,\ B)$.

Indeed, we have that:

$\sigma \in wp^I(\mathbf{while}\ b\ \mathbf{do}\ c_0,\ B)$ iff {by Fact 4.12 on page 133}
iff $I, \sigma \models wp(\mathbf{while}\ b\ \mathbf{do}\ c_0, B)$ iff {by Definition 4.11 on page 132}
iff I, $\llbracket \mathbf{while}\ b\ \mathbf{do}\ c_0 \rrbracket \sigma \models B$. □

Note that, in contrast to the statement of Theorem 7.7 on the previous page, in this new, equivalent statement of the theorem there is no reference to the state σ'.

THEOREM 7.9. [**Equivalence of Semantics of while-do. Part 2**] For all boolean expressions $b \in \mathbf{Bexp}$, for all commands $c_0 \in \mathbf{Com}$, for all states $\sigma \in \mathit{State}_\perp$, for all interpretations $I \in \mathbf{Intvar} \to N$, for all assertions $B \in \mathbf{Assn}$,

if $\sigma \in wp^I(\mathbf{while}\ b\ \mathbf{do}\ c_0,\ B)$
then there exists $\sigma' \in \mathit{State}_\perp$ such that $\llbracket \mathbf{while}\ b\ \mathbf{do}\ c_0 \rrbracket \sigma = \sigma' \ \wedge\ I, \sigma' \models B$.

PROOF. By cases according to Definition 7.6.
(*Case* α). Assume $\sigma = \perp$. Since the semantics function $\llbracket \_ \rrbracket$ for commands is a strict function, we have that $\llbracket \mathbf{while}\ b\ \mathbf{do}\ c_0 \rrbracket \perp = \perp$. (This case corresponds to Case (3) of Definition 7.1.) We have that $I, \perp \models B$. Thus, there exists a state σ', namely $\perp$, such that $\llbracket \mathbf{while}\ b\ \mathbf{do}\ c_0 \rrbracket \sigma = \sigma'$ and $I, \sigma' \models B$.
(*Case* β). Assume (β) of Definition 7.6.

Let us consider two cases: Case $\beta.1$ and Case $\overline{\beta.1}$.
(*Case* $\beta.1$). We assume (β) of Definition 7.6 and we also assume that:

$$\exists k \geq 0.\, \exists \sigma_0 \ldots \sigma_k \in \mathit{State}^k \cdot \mathit{State}_\perp.$$
$$\sigma_0 = \sigma \ \wedge\ \Big(\forall i,\ 0 \leq i < k.\ (\llbracket b \rrbracket \sigma_i = \mathit{true} \wedge \llbracket c_0 \rrbracket \sigma_i = \sigma_{i+1})\Big) \qquad (\beta.1)$$
$$\wedge\ \llbracket b \rrbracket \sigma_k = \mathit{false} \wedge I, \sigma_k \models B.$$

Let us consider two subcases: Subcase $\beta.1.1$ and Subcase $\beta.1.2$.
(*Subcase* $\beta.1.1$). Assume that in Condition $(\beta.1)$ we have that $k = 0$. Thus, we have that:
$\sigma_0 = \sigma \ \wedge\ \llbracket b \rrbracket \sigma_0 = \mathit{false} \wedge I, \sigma_0 \models B$. (This case corresponds to Case (1) of Definition 7.1.) We also have that $\llbracket \mathbf{while}\ b\ \mathbf{do}\ c_0 \rrbracket \sigma = \sigma_0$, because $\llbracket b \rrbracket \sigma_0 = \mathit{false}$ and $\sigma_0 = \sigma$. Thus, there exists a state σ', namely σ_0, such that $\llbracket \mathbf{while}\ b\ \mathbf{do}\ c_0 \rrbracket \sigma = \sigma'$ and $I, \sigma' \models B$.

(*Subcase* β.1.2). Assume that in Condition (β.1) we have that $k>0$. Thus, Condition (β.1) becomes:

$\exists k>0.\, \exists \sigma_0 \dots \sigma_k \in State^k \cdot State_\perp.$

$\sigma_0 = \sigma \wedge \Big(\forall i,\, 0 \leq i < k.\ (\llbracket b \rrbracket \sigma_i = true \wedge \llbracket c_0 \rrbracket \sigma_i = \sigma_{i+1})\Big)$ (β.1*)

$\wedge\ \llbracket b \rrbracket \sigma_k = false \wedge I, \sigma_k \models B.$

By (β) we have that $\llbracket b \rrbracket \sigma_k = true \vee I, \sigma_k \models B$. Thus, from ($\beta$.1*) we get: $I, \sigma_k \models B$. (This case corresponds to Case (2.1) of Definition 7.1 because $\llbracket b \rrbracket \sigma_k = false$ and, thus, $\sigma_k \neq \perp$.) Since by (β.1*) we have that for all i, with $0 \leq i < k$, $\llbracket b \rrbracket \sigma_i = true$ and $\llbracket b \rrbracket \sigma_k = false$, by Definition 7.1 on page 153 we get that $\llbracket \textbf{while } b \textbf{ do } c_0 \rrbracket \sigma = \sigma_k$. Thus, there exists a state σ', namely σ_k, such that $\llbracket \textbf{while } b \textbf{ do } c_0 \rrbracket \sigma = \sigma'$ and $I, \sigma' \models B$.

(*Case* $\overline{\beta.1}$). In this case we assume (β) of Definition 7.6 and we also assume that:

$\forall k \geq 0.\, \forall \sigma_0 \dots \sigma_k \in State^k \cdot State_\perp.$

$\Big[\ \sigma_0 = \sigma \wedge \Big(\forall i,\, 0 \leq i < k.\ (\llbracket b \rrbracket \sigma_i = true \wedge \llbracket c_0 \rrbracket \sigma_i = \sigma_{i+1})\Big)\ \Big]$ ($\overline{\beta.1}$)

$\Rightarrow (\llbracket b \rrbracket \sigma_k = true \ \vee\ I, \sigma_k \not\models B).$

Note that Condition ($\overline{\beta.1}$) is the negation of Condition (β.1). By (β) and ($\overline{\beta.1}$) we get:

$\forall k \geq 0.\, \forall \sigma_0 \dots \sigma_k \in State^k \cdot State_\perp.$

$\Big[\ \sigma_0 = \sigma \wedge \Big(\forall i,\, 0 \leq i < k.\ (\llbracket b \rrbracket \sigma_i = true \wedge \llbracket c_0 \rrbracket \sigma_i = \sigma_{i+1})\Big)\ \Big]$ ($\beta \wedge \overline{\beta.1}$)

$\Rightarrow \llbracket b \rrbracket \sigma_k = true.$

Let us consider two subcases: Subcase ($\overline{\beta.1.1}$) and Subcase ($\overline{\beta.1.2}$).

(*Subcase* $\overline{\beta.1.1}$). Let us assume that for all $k \geq 0$ and for all sequences $\sigma_0 \dots \sigma_k \in State^k \cdot State_\perp$, if $\sigma_0 = \sigma \wedge \Big(\forall i,\, 0 \leq i < k.\ (\llbracket b \rrbracket \sigma_i = true \wedge \llbracket c_0 \rrbracket \sigma_i = \sigma_{i+1})\Big)$ then $\sigma_k \neq \perp$. (This case corresponds to Case (4) of Definition 7.1.) By Definition 7.1 we have that $\llbracket \textbf{while } b \textbf{ do } c_0 \rrbracket \sigma = \perp$. We also have that $I, \perp \models B$ holds. Thus, there exists a state σ', namely $\perp$, such that $\llbracket \textbf{while } b \textbf{ do } c_0 \rrbracket \sigma = \sigma'$ and $I, \sigma' \models B$.

(*Subcase* $\overline{\beta.1.2}$). Let us assume that there exists $k \geq 0$ and there exists a sequence $\sigma_0 \dots \sigma_k \in State^k \cdot State_\perp$ for which we have that:

$\sigma_0 = \sigma \wedge \Big(\forall i,\, 0 \leq i < k.\ (\llbracket b \rrbracket \sigma_i = true \wedge \llbracket c_0 \rrbracket \sigma_i = \sigma_{i+1})\Big) \wedge \sigma_k = \perp.$

(This case corresponds to Case (2.2) of Definition 7.1.) For that sequence we have that:

$\sigma_0 = \sigma \wedge \Big(\forall i,\, 0 \leq i < k.\ \Rightarrow (\llbracket b \rrbracket \sigma_i = true \wedge \llbracket c_0 \rrbracket \sigma_i = \sigma_{i+1})\Big) \wedge \llbracket b \rrbracket \sigma_k = true.$

By Definition 7.1 we have that $\llbracket \textbf{while } b \textbf{ do } c_0 \rrbracket \sigma = \sigma_k = \perp$. We also have that $I, \sigma_k \models B$ holds because $\sigma_k = \perp$. Thus, there exists a state σ', namely $\perp$, such that $\llbracket \textbf{while } b \textbf{ do } c_0 \rrbracket \sigma = \sigma'$ and $I, \sigma' \models B$. $\square$

8. Nondeterministic Computations

In this section we present the language of the guarded commands introduced by E. D. Dijkstra **[3]** and we present its operational semantics in the style of the so called *big-step semantics*. In this kind of operational semantics a given command c and a given state σ are related to the state which we get, so to speak, at the end of the execution of c starting from σ. The guarded commands allow nondeterministic computations. We have the following new syntactic categories (which are mutually recursively defined) besides the arithmetic expressions **Aexp** and the boolean expressions **Bexp** (defined in Section 1 on page 117):

- Commands

  c ranges over **Com** $\quad c ::= \mathbf{skip} \mid \mathbf{abort} \mid X := a \mid c_1; c_2 \mid \mathbf{if}\ gc\ \mathbf{fi} \mid \mathbf{do}\ gc\ \mathbf{od}$

- Guarded Commands

  gc ranges over **Gcom** $\quad gc ::= b \rightarrow c \mid gc_1 [\!] gc_2$

where a ranges over the arithmetic expressions **Aexp** and b ranges over the boolean expressions **Bexp**.

Note that the set **Com** of commands we consider here, is different from the set **Com** of commands we considered in Section 1 on page 117.

The operator $[\!]$ is assumed to be associative and commutative.

The operational semantics given below specifies: (i) the execution of commands, and (ii) the execution of guarded commands.

8.1. Operational Semantics of Nondeterministic Commands.

The execution of commands is given as a subset of $\mathbf{Com} \times \mathit{State} \times \mathit{State}$. A triple in $\mathbf{Com} \times \mathit{State} \times \mathit{State}$ is written as $\langle c, \sigma\rangle \rightarrow \gamma$ and specifies that the command c from the state σ produces the new state γ. The axiom and the inference rules defining the execution of commands are as follows.

(1.1) $\langle \mathbf{skip}, \sigma\rangle \rightarrow \sigma$

(1.2) $$\frac{\langle a,\sigma\rangle \rightarrow n}{\langle X{:=}a, \sigma\rangle \rightarrow \sigma[n/X]}$$

(1.3) $$\frac{\langle c_1,\sigma\rangle \rightarrow \sigma_1 \qquad \langle c_2,\sigma_1\rangle \rightarrow \sigma_2}{\langle c_1; c_2, \sigma\rangle \rightarrow \sigma_2}$$

(1.4) $$\frac{\langle gc,\sigma\rangle \rightarrow \langle c,\sigma\rangle \qquad \langle c,\sigma\rangle \rightarrow \sigma'}{\langle \mathbf{if}\ gc\ \mathbf{fi}, \sigma\rangle \rightarrow \sigma'}$$

(1.5) $$\frac{\langle gc,\sigma\rangle \rightarrow \mathbf{fail}}{\langle \mathbf{do}\ gc\ \mathbf{od}, \sigma\rangle \rightarrow \sigma}$$

(1.6) $$\frac{\langle gc,\sigma\rangle \rightarrow \langle c,\sigma\rangle \qquad \langle c; \mathbf{do}\ gc\ \mathbf{od}, \sigma\rangle \rightarrow \sigma'}{\langle \mathbf{do}\ gc\ \mathbf{od}, \sigma\rangle \rightarrow \sigma'}$$

Let us informally explain the rules for the **if** ... **fi** and **do** ... **od** commands.

Starting from the state σ, the execution of $\mathbf{if}\ gc_1 [\!] \ldots [\!] gc_n\ \mathbf{fi}$ is performed as follows.

(i) We first choose in a nondeterministic way a guarded command, say gc_i, among $\{gc_1, \ldots, gc_n\}$, such that its guard b_i evaluates to **true**.
(ii) Let gc_i be $b_i \to c_i$. Then we execute the command c_i starting from σ.
(iii) If no guarded command among $\{gc_1, \ldots, gc_n\}$ has a guard which evaluates to **true**, the execution of the **if** $gc_1 [\!] \ldots [\!] gc_n$ **fi** command is aborted, that is, there is no state γ such that $\langle$**if** $gc_1 [\!] \ldots [\!] gc_n$ **fi**, $\sigma\rangle \to \gamma$. Moreover, the execution of the whole program is aborted, that is, there is no state γ such that $\langle c, \sigma\rangle \to \gamma$, where c is the command which includes the command **if** $gc_1 [\!] \ldots [\!] gc_n$ **fi**.

Starting from the state σ, the execution of **do** $gc_1 [\!] \ldots [\!] gc_n$ **od** is performed as follows.
(i) We first choose in a nondeterministic way a guarded command, say gc_i, among $\{gc_1, \ldots, gc_n\}$, such that its guard b_i evaluates to **true**.
(ii) Let gc_i be $b_i \to c_i$. Then we execute the command c_i starting from σ and we go to Step (i), that is, we execute again the command **do** $gc_1 [\!] \ldots [\!] gc_n$ **od** starting from the state γ, if any, such that $\langle c_i, \sigma\rangle \to \gamma$.
(iii) If no guarded command among $\{gc_1, \ldots, gc_n\}$, has a guard which evaluates to **true**, then the execution of the **do** ... **od** command is terminated.

Note that we gave no rules for the command **abort**. Thus, for every state $\sigma \in$ *State* and every $\gamma \in$ *State*, no triple $\langle$**abort**$, \sigma, \gamma\rangle$ exists in the subset of **Com** $\times$ *State* $\times$ *State* which denotes the operational semantics.

8.2. Operational Semantics of Guarded Commands.

The execution of guarded commands is given as a subset of **Gcom** $\times$ *State* $\times$ ((**Com** $\times$ *State*) $\cup$ {**fail**}). A triple in **Gcom** $\times$ *State* $\times$ ((**Com** $\times$ *State*) $\cup$ {**fail**}) is written as $\langle gc, \sigma\rangle \to \gamma$ and specifies that the guarded command gc from the state σ produces a value of γ which is either a new $\langle$command, state$\rangle$ pair or the value **fail**. The inference rules defining the execution of guarded commands are as follows.

$$(2.1)\ \frac{\langle b,\sigma\rangle \to \textbf{true}}{\langle b \to c, \sigma\rangle \to \langle c,\sigma\rangle} \qquad (2.2)\ \frac{\langle b,\sigma\rangle \to \textbf{false}}{\langle b \to c, \sigma\rangle \to \textbf{fail}}$$

$$(2.3)\ \frac{\langle gc_1,\sigma\rangle \to \langle c,\sigma\rangle}{\langle gc_1 [\!] gc_2, \sigma\rangle \to \langle c,\sigma\rangle} \qquad (2.4)\ \frac{\langle gc_2,\sigma\rangle \to \langle c,\sigma\rangle}{\langle gc_1 [\!] gc_2, \sigma\rangle \to \langle c,\sigma\rangle}$$

$$(2.5)\ \frac{\langle gc_1, \sigma\rangle \to \textbf{fail} \quad \langle gc_2, \sigma\rangle \to \textbf{fail}}{\langle gc_1 [\!] gc_2, \sigma\rangle \to \textbf{fail}}$$

Note that when evaluating the guard of a guarded command, the state does *not* change. This fact is not exploited in the semantic rules given by Winskel **[19]**.

Instead of the above rules (2.3) and (2.4), we can equivalently use the following two rules:

$$(2.3^*)\ \frac{\langle b \to c,\sigma\rangle \to \langle c,\sigma\rangle}{\langle b \to c [\!] gc_2, \sigma\rangle \to \langle c,\sigma\rangle} \qquad (2.4^*)\ \frac{\langle b \to c,\sigma\rangle \to \langle c,\sigma\rangle}{\langle gc_1 [\!] b \to c, \sigma\rangle \to \langle c,\sigma\rangle}$$

Note also that for all b, c, σ, gc_1, gc_2, we have that:
(i) $\langle b \to c, \sigma\rangle \to \langle c, \sigma\rangle$ holds iff $\langle b, \sigma\rangle \to$ **true**, and

(ii) $\langle gc_1 [\!] gc_2, \sigma\rangle \rightarrow \langle c, \sigma\rangle$ holds iff there exists $i \in \{1,2\}$ such that gc_i is $b \rightarrow c$ and $\langle b, \sigma\rangle \rightarrow \mathbf{true}$.

Here is Euclid's algorithm for computing the greatest common divisor of M and N using guarded commands (we assume that $M>0$ and $N>0$):

$\{n=N>0 \wedge m=M>0\}$
do $m \geq n \wedge n > 0 \rightarrow m := m-n$
$[\!]$ $n \geq m \wedge m > 0 \rightarrow n := n-m$
od
$\{gcd(N,M) = \mathit{if}\ n=0\ \mathit{then}\ m\ \mathit{else}\ n\}$

The assertions between curly brackets at the beginning and at the end of the above **do** ... **od** command relate the value of the integer variables before and after the execution of that command.

9. Owicki-Gries Calculus for Parallel Programs

In this section we present the Owicki-Gries rules for proving properties of imperative parallel programs. They are constructed using commands c of the following form:

$$c ::= \mathbf{skip} \mid X := a \mid c_1; c_2 \mid \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \mid \mathbf{while}\ b\ \mathbf{do}\ c_0 \mid c_1\ \mathbf{par}\ c_2$$

where a is any arithmetic expression in **Aexp** and b is any boolean expression in **Bexp** (see Section 1 on page 117). The Owicki-Gries rules are like the Hoare rules for partial correctness (see page 128), but they use decorated Hoare triples, as we now explain.

A *decorated Hoare triple* $\{A\}\, c\, \{B\} \mid {}^{R}_{W}$ is a Hoare triple with two sets R and W, called the *upper decoration* and the *lower decoration*, respectively. The set R is the set of locations which the command c reads and does not write, and W is the set of the locations that the command c writes. Thus, if there is a location X that c reads and writes (this occurs, for instance, in the command $X := X+1$), then $X \notin R$ and $X \in W$.

By definition, (i) every location occurring in c belongs to either R or W, and (ii) for every decorated triple $\{A\}\, c\, \{B\} \mid {}^{R}_{W}$, we have that $R \cap W = \{\}$.

Given an expression e by $Locs(e)$ we denote the set of locations occurring in e.

We say that a decorated triple $\{A\}\, c\, \{B\} \mid {}^{R}_{W}$ is *derivable* in the Owicki-Gries Calculus, and we write $\vdash \{A\}\, c\, \{B\} \mid {}^{R}_{W}$, iff there is a proof (that is, a finite derivation) of $\{A\}\, c\, \{B\} \mid {}^{R}_{W}$ by using the following axiom and inference rules.

$(OG1)$ $\quad \{A\}\ \mathbf{skip}\ \{A\} \mid {}^{\{\}}_{\{\}}$

$$(OG2) \quad \frac{\models A \Rightarrow B[a/X]}{\{A\}\ X{:=}a\ \{B\} \mid {}^{Locs(a)-\{X\}}_{\{X\}}}$$

$$(OG3)\quad \frac{\{A\}\ c_1\ \{B\} \mid \begin{smallmatrix} R1 \\ W1 \end{smallmatrix} \qquad \{B\}\ c_2\ \{C\} \mid \begin{smallmatrix} R2 \\ W2 \end{smallmatrix}}{\{A\}\ \ c_1\,;c_2\ \ \{C\} \mid \begin{smallmatrix} (R1\cup R2)-(W1\cup W2) \\ W1\cup W2 \end{smallmatrix}}$$

$$(OG4)\quad \frac{\{A\wedge b\}\ c_1\ \{B\} \mid \begin{smallmatrix} R1 \\ W1 \end{smallmatrix} \qquad \{A\wedge \neg b\}\ c_2\ \{B\} \mid \begin{smallmatrix} R2 \\ W2 \end{smallmatrix}}{\{A\}\ \ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \ \{B\} \mid \begin{smallmatrix} (Locs(b)\cup R1\cup R2)-(W1\cup W2) \\ W1\cup W2 \end{smallmatrix}}$$

$$(OG5)\quad \frac{\{A\wedge b\}\ c\ \{A\} \mid \begin{smallmatrix} R \\ W \end{smallmatrix}}{\{A\}\ \ \textbf{while}\ b\ \textbf{do}\ c\ \ \{A\wedge\neg b\} \mid \begin{smallmatrix} (Locs(b)\cup R)-W \\ W \end{smallmatrix}}$$

$$(OG6)\quad \frac{\{A1\}\ c_1\ \{B1\} \mid \begin{smallmatrix} R1 \\ W1 \end{smallmatrix} \qquad \{B2\}\ c_2\ \{C2\} \mid \begin{smallmatrix} R2 \\ W2 \end{smallmatrix}}{\{A1\wedge A2\}\ \ c_1\ \textbf{par}\ c_2\ \ \{B1\wedge B2\} \mid \begin{smallmatrix} R1\cup R2 \\ W1\cup W2 \end{smallmatrix}} \quad \left|\begin{array}{l}\text{assume } R1\cap W2 = \\ = R2\cap W1 = \\ = W1\cap W2 = \{\}\end{array}\right.$$

$$(OG7)\quad \frac{\models A\Rightarrow A' \qquad \{A'\}\ c\ \{B'\} \mid \begin{smallmatrix} R \\ W \end{smallmatrix} \qquad \models B'\Rightarrow B}{\{A\}\ c\ \{B\} \mid \begin{smallmatrix} R \\ W \end{smallmatrix}}$$

With reference to rule $(OG3)$, the upper decoration should be the set of locations read by either c_1 or c_2, and not written by c_1 or c_2, that is, $(R1\cup R2)-(W1\cup W2)$. This value is equal to $(R1-W2)\cup(R2-W1)$, because $R1\cap W1 = R2\cap W2 = \{\}$.

The assumptions we have made for rule rule $(OG6)$, allow us to execute the commands c_1 and c_2 in parallel and, without knowing their relative execution speed, we have that the final values stored in the locations which are written by those commands, is uniquely determined.

We have that if $\vdash \{A\}\,c\,\{B\} \mid \begin{smallmatrix} R \\ W \end{smallmatrix}$ holds then the decorated triple $\{A\}\,c\,\{B\} \mid \begin{smallmatrix} R \\ W \end{smallmatrix}$ is *valid* and by this we mean that for all interpretations $I \in \mathbf{Intvar} \to N$, for all states $\sigma \in State_\perp$, $I,\sigma \models A$ implies $I, [\![c]\!]\sigma \models B$. As for the Hoare triples, when a decorated triple $\{A\}\,c\,\{B\} \mid \begin{smallmatrix} R \\ W \end{smallmatrix}$ is valid we write $\models \{A\}\,c\,\{B\} \mid \begin{smallmatrix} R \\ W \end{smallmatrix}$.

The rules $OG1-OG7$ of the Owicki-Gries Calculus we have presented above, can be shown to be sound and relatively complete in the same sense in which the rules $H1-H6$ of the Hoare Calculus are sound and relatively complete (see Section 4.3 on page 128, Theorem 4.8 on page 129, and Theorem 4.24 on page 143). We will not present the proofs of these facts here and the interested reader is encouraged to look at the relevant literature.

CHAPTER 6

Syntax and Semantics of First Order Functional Languages

In this chapter we consider a first order functional language, called REC, under two evaluation regimes, the call-by-value and the call-by-name regime. For each of these regimes we will provide the operational and the denotational semantics.

1. Syntax of the First Order Functional Language REC

In this section we present the syntax of the first order functional language REC. Let us first introduce the following basic sets.

(i) The set of *integers* $N = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. The variables ranging over N are: $n, m, \ldots$

(ii) The set $\{+, -, \times\}$ of the *arithmetic operators*. **op** ranges over $\{+, -, \times\}$.

(iii) The set $\mathbf{Var} = \{x, y, z, \ldots\}$ of the *integer variables* (or *variables*, for short).

(iv) The set $\mathbf{Fvar} = \{f_1, f_2, \ldots, g, h, \ldots\}$ of the *function variables* with arity $a\,(\geq 0)$.

The syntax of the REC language is defined by: (v) the sets of terms, and (vi) the set of declarations.

(v) The set **Term** of *terms* is defined as follows:

$$t \;::=\; n \mid x \mid t_1\,\mathbf{op}\,t_2 \mid \mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \mid f_i(t_1, \ldots, t_{a_i})$$

where $t, t_0, t_1, t_2, t_{a_i} \in \mathbf{Term}$, $n \in N$, $x \in \mathbf{Var}$, and $f_i \in \mathbf{Fvar}$. A term is said to be *closed* if no integer variable occurs in it.

(vi) The set of *declarations* is a set of equations, each of which is called *a declaration*. A set of declarations is represented as follows:

$$\begin{cases} f_1(x_1, \ldots, x_{a_1}) &= d_1 \\ & \ldots \\ f_k(x_1, \ldots, x_{a_k}) &= d_k \end{cases}$$

where: (1) the f_i's are distinct function variables, (2) the x_i's are integer variables, and (3) the d_i's are terms. We assume that for $i = 1, \ldots, k$, every variable occurring in d_i belongs to $\{x_1, \ldots, x_{a_i}\}$.

For instance, the factorial function is defined by the following declaration:

$$fact(x) = \mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x \times fact(x-1)$$

In order to compute the factorial of an integer, say 5, we have to evaluate the term $fact(5)$, where for all $x \geq 0$, $fact(x)$ is defined by the above declaration.

Note that in order to avoid the introduction of boolean values, we encode the value *true* by the integer 0 and the value *false* by any other integer. Thus, for instance, **if** 0 **then** 4 **else** 7 evaluates to 4, and **if** 3 **then** 4 **else** 7 evaluates to 7.

The evaluation function is defined by the semantics of the language REC which we will now introduce. Actually, we will define two different semantics which are usually called the *call-by-value* semantics and the *call-by-name* semantics, respectively.

In order to see the reason for these two semantics, let us consider the following declarations of the function f and g both of arity 1:

$$\begin{cases} f(x) = 1 \\ g(x) = g(x+1) \end{cases}$$

Now there are two options for the evaluation of the term $f(g(1))$: either the *call-by-value* regime, or the *call-by-name* regime.

(*call-by-value regime*): the evaluation of $f(g(1))$ does not return any value, because we assume that before evaluating the function declaration, we first compute the values of all the arguments (and, obviously, the evaluation of $g(1)$ does not terminate because it requires the evaluation of $g(2)$, which in turn requires the evaluation of $g(3)$, and so on), or

(*call-by-name regime*): the evaluation of $f(g(1))$ returns the value 1, because we assume that we first evaluate the declaration of the outermost function call and then we evaluate the arguments which are actually needed for the evaluation of that declaration.

The call-by-value regime defines the so called *call-by-value* semantics, while the call-by-name regime defines the so called *call-by-name* semantics.

2. Call-by-value Operational Semantics of REC

In this section we give the operational semantics of the language REC under the call-by-value regime. It will be given as a set of deduction rules which are shown in Table 1 on the next page. Those rules define the rewriting relation $\rightarrow^{va} \subseteq \mathbf{Term} \times \mathbf{Term}$. The superscript ^{va} is for denoting the call-by-value regime. For reasons of simplicity, we will also write $t_1 \rightarrow t_2$, instead of $t_1 \rightarrow^{va} t_2$, for any two terms t_1 and t_2.

Note that in the call-by-value semantics we have $f_i(t_1, \ldots, t_{a_i}) \rightarrow n$ if for some integers $n_1, \ldots, n_{a_i}$, we have that $t_1 \rightarrow n_1, \ldots, t_{a_i} \rightarrow n_{a_i}$, and $d_i[n_1/x_1, \ldots, n_{a_i}/x_{a_i}] \rightarrow n$. This rule enforces the evaluation of all the arguments before the evaluation of the function declaration.

Note also that the evaluation of the **if** t_0 **then** t_1 **else** t_2 construct first requires the evaluation of the term t_0 and then it requires the evaluation of the term t_1 or t_2, according to the value of t_0, whether or not it is 0.

REMARK 2.1. In giving the operational semantics there is no context-rule, that is, we do *not* have the following rule: for any term t, for any integer n,

$$\frac{t \rightarrow n}{C[t] \;\rightarrow\; C[n]}$$

where a *context* $C[\_]$ is any term 'with a missing subterm'.

Call-by-value Operational Semantics of the language REC.

$$n \to n$$

$$\frac{t_1 \to n_1 \quad t_2 \to n_2}{t_1\,\mathbf{op}\,t_2 \to n}$$

where $n = n_1\,op\,n_2$ and op is the semantic operation corresponding to **op**

$$\frac{t_0 \to 0 \qquad t_1 \to n_1}{\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \to n_1} \qquad \frac{t_0 \to n \quad t_2 \to n_2 \quad n{\neq}0}{\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \to n_2}$$

$$\frac{t_1 \to n_1 \quad \ldots \quad t_{a_i} \to n_{a_i} \quad d_i[n_1/x_1, \ldots, n_{a_i}/x_{a_i}] \to n}{f_i(t_1,\ldots,t_{a_i}) \to n}$$

where d_i is the right hand side of the declaration of the function f_i.

TABLE 1. Call-by-value Operational Semantics of the language REC. For simplicity, we write $t_1 \to t_2$, instead of $t_1 \to^{va} t_2$.

More formally, the contexts $C[\_]$ in the language REC are defined by the following context-free grammar:

$$\begin{aligned} C[\_] ::= {} & [\_] \mid [\_]\,\mathbf{op}\,t_2 \mid t_1\,\mathbf{op}\,[\_] \\ & \mid \mathbf{if}\ [\_]\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \mid \mathbf{if}\ t_0\ \mathbf{then}\ [\_]\ \mathbf{else}\ t_2 \mid \mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ [\_] \\ & \mid f_i([\_], \ldots, t_{a_i}) \mid \ \ldots\ \mid f_i(t_1, \ldots, [\_]) \end{aligned}$$

Given a context $C[\_]$ and a term t, by $C[t]$ we denote the context $C[\_]$ with the term t in place of the missing subterm.

REMARK 2.2. The rules we have given specify a *big-step semantics* in the sense that the binary relation $\to$ relates a given term t to its value n without specifying any intermediate term. For instance, rules such as the following ones would specify a semantics for the **if** t_0 **then** t_1 **else** t_2 construct which is not big-step semantics and is, instead, a *small-step semantics*.

$$\frac{t_0 \to 0}{\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \to t_1} \qquad \frac{t_0 \to n \qquad n{\neq}0}{\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \to t_2}$$

We have the following fact which tells us that the call-by-value operational semantics defines a function. We leave the proof to the reader. It can be done by rule induction.

FACT 2.3. [**Determinism of Call-by-value Operational Semantics**] For all term t, integers n_1 and n_2, if $t \to^{va} n_1$ and $t \to^{va} n_2$ then $n_1 = n_2$.

3. Call-by-value Denotational Semantics of REC

In this section we give the denotational semantics of the language REC under the call-by-value regime as a set of equations. These equations define a unique semantic function $[\![\_]\!]^{va}$ by structural induction.

We first introduce the semantic domains $\mathbf{Env}^{va}$ and $\mathbf{Fenv}^{va}$ which are defined as follows. Let N denote the discrete cpo of the integers and $N_\perp$ denote the flat cpo of the integers with the bottom element $\perp$.

(i) $\mathbf{Env}^{va}$ is the cpo $[\mathbf{Var} \to N]$ of the *environments*.
Thus, an environment $\rho \in \mathbf{Env}^{va}$ is a function from the discrete cpo **Var** of the integer variables to N.

(ii) $\mathbf{Fenv}^{va}$ is the cpo of *function environments*.
For a given set of k declarations whose i-th declaration defines a function with arity a_i, $\mathbf{Fenv}^{va}$ is the cpo $[N^{a_1} \to N_\perp] \times \ldots \times [N^{a_k} \to N_\perp]$. Thus, a function environment $\varphi \in \mathbf{Fenv}^{va}$ is the product of k continuous functions. For $i = 1, \ldots, k$, the i-th projection of φ, denoted φ_i, is a continuous function in the cpo $[N^{a_i} \to N_\perp]$.

The denotational semantic function under the call-by-value regime will be denoted by $[\![\_]\!]^{va}$.
For all terms t, $[\![t]\!]^{va}$ is the unique continuous function in $[\mathbf{Fenv}^{va} \to [\mathbf{Env}^{va} \to N_\perp]]$ defined by structural induction by the equations of Table 2 for any given function environment $\varphi \in \mathbf{Fenv}^{va}$ and any given environment $\rho \in \mathbf{Env}^{va}$.

When understood from the context we will avoid writing the superscript ^{va}.

Call-by-value Denotational Semantics of the language REC.

$$[\![\, n \,]\!]\, \varphi\, \rho \;=\; \lfloor n \rfloor$$
$$[\![\, x \,]\!]\, \varphi\, \rho \;=\; \lfloor \rho(x) \rfloor$$
$$[\![\, t_1 \,\mathbf{op}\, t_2 \,]\!]\, \varphi\, \rho \;=\; [\![\, t_1 \,]\!]\, \varphi\, \rho \;\; op_\perp \;\; [\![\, t_2 \,]\!]\, \varphi\, \rho$$
$$[\![\, \mathbf{if}\, t_0\, \mathbf{then}\, t_1\, \mathbf{else}\, t_2 \,]\!]\, \varphi\, \rho \;=\; \mathit{Cond}([\![\, t_0 \,]\!]\, \varphi\, \rho, [\![\, t_1 \,]\!]\, \varphi\, \rho, [\![\, t_2 \,]\!]\, \varphi\, \rho)$$
$$[\![\, f_i(t_1, \ldots, t_{a_i}) \,]\!]\, \varphi\, \rho \;=\; \mathit{let}\, v_1 \Leftarrow [\![\, t_1 \,]\!]\, \varphi\, \rho,\; \ldots,\; v_{a_i} \Leftarrow [\![\, t_{a_i} \,]\!]\, \varphi\, \rho \,\centerdot\, \varphi_i(v_1, \ldots, v_{a_i})$$

TABLE 2. Call-by-value Denotational Semantics of the language REC. For simplicity, we write $[\![\_]\!]$, instead of $[\![\_]\!]^{va}$.

Note also that, as usual in mathematics, the expressions which provide the call-by-value semantics, should be evaluated in the call-by-name regime. Thus, in particular, if the function φ is the constant function $\lambda x.1$ then $\varphi(\perp) = 1$.

For the denotational semantics we have a context-rule of the form:

if $[\![\, t_1 \,]\!]\, \varphi\, \rho = [\![\, t_2 \,]\!]\, \varphi\, \rho$
then for any context $C[\_]$ we have that $[\![\, C[t_1] \,]\!]\, \varphi\, \rho = [\![\, C[t_2] \,]\!]\, \varphi\, \rho$.

3.1. Computation of the function environment in call-by-value.

The function environment $\varphi \in \mathbf{Fenv}^{va}$ associated with a given set of declarations can be computed as the minimal fixpoint of a continuous functional as we now explain.

Let us consider the following k declarations:

$$\begin{cases} f_1(x_1,\ldots,x_{a_1}) & = & d_1 \\ & \ldots & \\ f_k(x_1,\ldots,x_{a_k}) & = & d_k \end{cases}$$

Associated with these k declarations, we have a k-tuple $\langle \delta_1,\ldots,\delta_k\rangle$ of functions belonging to the cpo $\mathbf{Fenv}^{va} = [N^{a_1} \to N_\perp] \times \ldots \times [N^{a_k} \to N_\perp]$ which is recursively defined as follows:

$$\begin{cases} \delta_1 & = & \lambda n_1.\ \ldots\ \lambda n_{a_1}.\, [\![d_1]\!]\, \langle \delta_1,\ldots,\delta_k\rangle\, \rho[n_1/x_1,\ldots,n_{a_1}/x_{a_1}] \\ & \ldots & \\ \delta_k & = & \lambda n_1.\ \ldots\ \lambda n_{a_k}.\, [\![d_k]\!]\, \langle \delta_1,\ldots,\delta_k\rangle\, \rho[n_1/x_1,\ldots,n_{a_k}/x_{a_k}] \end{cases}$$

This system of k equations can be rewritten as follows:

$$\begin{cases} \delta_1 & = & (\lambda\chi.\, \lambda n_1.\ \ldots\ \lambda n_{a_1}.\, [\![d_1]\!]\, \chi\, \rho[n_1/x_1,\ldots,n_{a_1}/x_{a_1}])\, \langle \delta_1,\ldots,\delta_k\rangle \\ & \ldots & \\ \delta_k & = & (\lambda\chi.\, \lambda n_1.\ \ldots\ \lambda n_{a_k}.\, [\![d_k]\!]\, \chi\, \rho[n_1/x_1,\ldots,n_{a_k}/x_{a_k}])\, \langle \delta_1,\ldots,\delta_k\rangle \end{cases}$$

and it defines a continuous functional τ such that

$\langle \delta_1,\ldots,\delta_k\rangle = \tau\, \langle \delta_1,\ldots,\delta_k\rangle$.

The minimal fixpoint of τ is the function environment $\varphi = \langle \varphi_1,\ldots,\varphi_k\rangle \in \mathbf{Fenv}^{va}$ associated with the given k declarations. For $i = 1,\ldots,k$, the environment φ associates the function φ_i with the function variable f_i.

The minimal fixpoint of τ exists because τ is continuous. This continuity property follows from the facts that (see Chapter 4 Section 5 starting on page 101):
(i) $\delta_1,\ldots,\delta_k$ are variables, (ii) lambda abstraction is continuous, (iii) ρ is a continuous function in $[\mathbf{Var} \to N]$, (iv) environment updating of the form $\rho[n/x]$ is a continuous operation, (v) tupling is continuous, (vi) function application is continuous, and (vii) all operators occurring in the right hand sides of the semantic equations, such as $\lfloor\_\rfloor$, $op_\perp$, *Cond*, and the *let* construct, are continuous.

EXAMPLE 3.1. [**Factorial Function in Call-by-value**] In the case of the declaration of the factorial function:

$fact(x) = \mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x \times fact(x-1)$

we have that

$\delta = (\lambda\chi.\, \lambda n \in N.\, [\![\mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times fact(x-1)]\!]\, \chi\, \rho[n/x])\, \delta$

and the value of function variable *fact* is the minimal fixpoint of the functional

$\tau =_{def} \lambda\chi.\, \lambda n \in N.\, [\![\mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times fact(x-1)]\!]\, \chi\, \rho[n/x]$.

Thus, in this case $\mathbf{Fenv} = [N \to N_\perp]$ and the function environment $\varphi \in [N \to N_\perp]$ is the function $fix(\tau) = \bigsqcup_{k\geq 0} \tau^k(\perp)$. We have that:

$\tau^0(\bot) = \lambda n{\in}N.\, \bot$

$\tau^1(\bot) = \lambda n{\in}N.\, \mathit{Cond}(\lfloor n\rfloor, \lfloor 1\rfloor, \lfloor n\rfloor \times_\bot (\tau^0(\bot))(n{-}1)) = \begin{cases} \lfloor 1\rfloor & \mathit{if}\ n{=}0 \\ \bot & \mathit{otherwise} \end{cases}$

$\tau^2(\bot) = \lambda n{\in}N.\, \mathit{Cond}(\lfloor n\rfloor, \lfloor 1\rfloor, \lfloor n\rfloor \times_\bot (\tau^1(\bot))(n{-}1)) = \begin{cases} \lfloor 1\rfloor & \mathit{if}\ n{=}0 \\ \lfloor 1\rfloor & \mathit{if}\ n{=}1 \\ \bot & \mathit{otherwise} \end{cases}$

Then, as one can prove by mathematical induction, we get that, for any $k \geq 0$,

$\tau^k(\bot) = \lambda n{\in}N. \begin{cases} \lfloor n!\rfloor & \mathit{if}\ 0{\leq}n{<}k \\ \bot & \mathit{otherwise} \end{cases}$

Thus, $\varphi = \mathit{fix}(\tau) = \bigsqcup_{k\geq 0} \tau^k(\bot) = \lambda n. \begin{cases} \lfloor n!\rfloor & \mathit{if}\ 0{\leq}n \\ \bot & \mathit{otherwise} \end{cases}$

The following table shows the computation of $\tau^k(\bot)$, for $k \geq 0$, of the functional τ and the limit point $\mathit{fix}(\tau) = \bigsqcup_{k\geq 0} \tau^k(\bot)$. This table is constructed starting from the lowest row and going upwards.

$n \in N$:	$\ldots$	-2	-1	0	1	2	3	4	$\ldots$
$\varphi =_{def} \bigsqcup_{k\geq 0} \tau^k(\bot)$	$\ldots$	$\bot$	$\bot$	$\lfloor 1\rfloor$	$\lfloor 1\rfloor$	$\lfloor 2\rfloor$	$\lfloor 6\rfloor$	$\lfloor 24\rfloor$	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\tau^4(\bot)$	$\ldots$	$\bot$	$\bot$	$\lfloor 1\rfloor$	$\lfloor 1\rfloor$	$\lfloor 2\rfloor$	$\lfloor 6\rfloor$	$\bot$	$\ldots$
$\tau^3(\bot)$	$\ldots$	$\bot$	$\bot$	$\lfloor 1\rfloor$	$\lfloor 1\rfloor$	$\lfloor 2\rfloor$	$\bot$	$\bot$	$\ldots$
$\tau^2(\bot)$	$\ldots$	$\bot$	$\bot$	$\lfloor 1\rfloor$	$\lfloor 1\rfloor$	$\bot$	$\bot$	$\bot$	$\ldots$
$\tau^1(\bot)$	$\ldots$	$\bot$	$\bot$	$\lfloor 1\rfloor$	$\bot$	$\bot$	$\bot$	$\bot$	$\ldots$
$\tau^0(\bot)$	$\ldots$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\ldots$

TABLE 3. Computation of the minimal fixpoint $\bigsqcup_{k\geq 0} \tau^k(\bot)$ relative to the functional $\tau =_{def} \lambda\chi.\, \lambda n.\, [\![\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x \times \mathit{fact}(x{-}1)]\!]\, \chi\, \rho[n/x]$ associated with the declaration of the factorial function.

As indicated on the topmost row of Table 3, we have that for all $n \geq 0$,

$\big(\bigsqcup_{k\geq 0} \tau^k(\bot)\big)(n) = \lfloor n!\rfloor$.

The call-by-value evaluation of the factorial function for the input 3 is as follows:

$[\![\mathit{fact}(3)]\!]\, \varphi\, \rho = \mathit{let}\ v \Leftarrow [\![3]\!]\, \varphi\, \rho\, .\, (\varphi\, v) = \mathit{let}\ v \Leftarrow \lfloor 3\rfloor\, .\, (\varphi\, v) = \varphi(3) = \lfloor 6\rfloor$. □

Now let us present a second example of evaluation of the call-by-value denotational semantics. In this second example we will see that the value of $\mathit{fact}(2)$ can be computed *without* computing the minimal fixpoint φ of the functional τ associated with the declaration of the function *fact* (see the topmost row of the above table). Indeed, it is enough to use the property that for all $n \in N$, the minimal fixpoint φ satisfies the following equation:

$\varphi(n) = \mathit{Cond}(\lfloor n\rfloor, \lfloor 1\rfloor, \lfloor n\rfloor \times_\bot \varphi(n{-}1))$.

Note that for every input value $n \in N$, the value of $\varphi(n) =_{def} \big(\bigsqcup_{k\geq 0} \tau^k(\bot)\big)(n)$ is obtained as the value of $(\tau^k(\bot))(n)$, for a sufficiently large value of $k\,(\geq 0)$.

EXAMPLE 3.2. [**Factorial Function in Call-by-value. Version 2**] We want to compute the value of $fact(2)$ where, as in the previous Example 3.1, the function $fact$ is defined by the following declaration:

$fact(x) = \textbf{if } x \textbf{ then } 1 \textbf{ else } x \times fact(x-1)$.

We have that:

$$\begin{array}{l}\tau\ \varphi\ x = Cond(\lfloor n\rfloor, \lfloor 1\rfloor, \lfloor n\rfloor \times_\bot (let\, v \Leftarrow \lfloor n{-}1\rfloor \centerdot \varphi(v))) = \\ \qquad = Cond(\lfloor n\rfloor, \lfloor 1\rfloor, \lfloor n\rfloor \times_\bot \varphi(n{-}1))\end{array}$$

with $\tau \in [[N \to N_\bot] \to [N \to N_\bot]$, $\varphi \in [N \to N_\bot]$, and $n \in N$.

The evaluation of $[\![fact(2)]\!]\,\varphi\,\rho$ is as follows.

$$\begin{array}{ll}[\![fact(2)]\!]\,\varphi\,\rho = let\ v \Leftarrow \lfloor 2\rfloor \centerdot \varphi(v) = & \\ \quad = \varphi(2) = & \\ \quad = (\tau\,\varphi)(2) = & \\ \quad = Cond(\lfloor 2\rfloor, \lfloor 1\rfloor, \lfloor 2\rfloor \times_\bot \varphi(2{-}1)) = & (\dagger) \\ \quad = \lfloor 2\rfloor \times_\bot \varphi(2{-}1) = & \\ \quad = \lfloor 2\rfloor \times_\bot (Cond(\lfloor 2{-}1\rfloor, \lfloor 1\rfloor, \lfloor 2{-}1\rfloor \times_\bot \varphi((2{-}1){-}1))) = & \\ \quad = \lfloor 2\rfloor \times_\bot (Cond(\lfloor 1\rfloor, \lfloor 1\rfloor, \lfloor 2{-}1\rfloor \times_\bot \varphi((2{-}1){-}1))) = & \\ \quad = \lfloor 2\rfloor \times_\bot (\lfloor 2{-}1\rfloor \times_\bot \varphi((2-1)-1)) = & \\ \quad = \lfloor 2\rfloor \times_\bot (\lfloor 2{-}1\rfloor \times_\bot Cond(\lfloor (2{-}1){-}1\rfloor, \lfloor 1\rfloor, \lfloor (2{-}1){-}1\rfloor \times_\bot \varphi(((2{-}1){-}1){-}1))) = & \\ \quad = \lfloor 2\rfloor \times_\bot (\lfloor 2{-}1\rfloor \times_\bot \lfloor 1\rfloor) = \lfloor 2\rfloor. & \end{array}$$

Since φ is a function, we have that: if $e_1 = e_2$ then $\varphi(e_1) = \varphi(e_2)$. Thus, the above derivation could have also be done as follows. Continuing from the above Expression (†):

$$\begin{array}{l}Cond(\lfloor 2\rfloor, \lfloor 1\rfloor, \lfloor 2\rfloor \times_\bot \varphi(2{-}1)) = \\ = \lfloor 2\rfloor \times_\bot \varphi(1) = \\ = \lfloor 2\rfloor \times_\bot (Cond(\lfloor 1\rfloor, \lfloor 1\rfloor, \lfloor 1\rfloor \times_\bot \varphi(0)) = \\ = \lfloor 2\rfloor \times_\bot (\lfloor 1\rfloor \times_\bot \varphi(0)) = \\ = \lfloor 2\rfloor \times_\bot (\lfloor 1\rfloor \times_\bot Cond(\lfloor 0\rfloor, \lfloor 1\rfloor, \lfloor (2{-}1){-}1\rfloor \times_\bot \varphi(-1))) = \\ = \lfloor 2\rfloor \times_\bot (\lfloor 1\rfloor \times_\bot \lfloor 1\rfloor) = \lfloor 2\rfloor. \end{array}$$ □

EXAMPLE 3.3. [**Constant Functions and Looping Functions in Call-by-value**] Let us consider the declaration:

$$\begin{cases} f(x) = 1 \\ g(x) = g(x{+}1) \end{cases}$$

In this case the function environment φ is a pair of functions

$\langle \varphi_1, \varphi_2 \rangle \in [N \to N_\bot] \times [N \to N_\bot]$,

the function variable f being associated with φ_1 and the function variable g being associated with φ_2. We have that $\langle \varphi_1, \varphi_2 \rangle$ is the minimal fixpoint of the functional

which transforms the pair $\langle \delta_1, \delta_2 \rangle$ of functions into itself, as specified by the following equation:

$$\langle \delta_1, \delta_2 \rangle = \langle \lambda\chi.\, \lambda n \in N.\, [\![1]\!]\, \chi\, \rho[n/x],\ \ \lambda\chi.\, \lambda n \in N.\, [\![g(x+1)]\!]\, \chi\, \rho[n/x] \rangle\ \langle \delta_1, \delta_2 \rangle \qquad (\dagger\dagger)$$

Now we have that:

$[\![1]\!]\, \chi\, \rho[n/x] = \lfloor 1 \rfloor$

and, if we denote by χ_2 the second component of the argument χ which is a pair, we also have that:

$[\![g(x+1)]\!]\, \chi\, \rho[n/x] =$

$= \mathit{let}\ v \Leftarrow [\![x+1]\!]\, \chi\, \rho[n/x] \centerdot \chi_2(v) =$

$= \mathit{let}\ v \Leftarrow \lfloor n \rfloor +_\perp \lfloor 1 \rfloor \centerdot \chi_2(v) =$

$= \chi_2(n+1).$

Thus, Equation $(\dagger\dagger)$ above becomes

$$\langle \delta_1, \delta_2 \rangle = \langle \lambda\chi.\, \lambda n \in N.\, \lfloor 1 \rfloor,\ \ \lambda\chi.\, \lambda n \in N.\, \chi_2(n+1) \rangle\ \langle \delta_1, \delta_2 \rangle$$

and we have that $\langle \varphi_1, \varphi_2 \rangle$ is the minimal fixpoint of the functional τ:

$$\tau =_{def} \langle \lambda\chi.\, \lambda n \in N.\, \lfloor 1 \rfloor,\ \ \lambda\chi.\, \lambda n \in N.\, \chi_2(n+1) \rangle.$$

That minimal fixpoint is computed as follows, where $n, m \in N$ and, for $i = 0, 1$, the term $(\tau^i(\perp))_2$ denotes the second component of the pair $\tau^i(\perp)$:

$\tau^0(\perp) = \langle \lambda n.\, \perp,\ \lambda n.\, \perp \rangle$

$\tau^1(\perp) = \langle \lambda n.\, \lfloor 1 \rfloor,\ \lambda n.\, ((\tau^0(\perp))_2(n+1)) \rangle =$

$\quad = \langle \lambda n.\, \lfloor 1 \rfloor,\ \lambda n.\, ((\lambda m.\perp)(n+1)) \rangle =$

$\quad = \langle \lambda n.\, \lfloor 1 \rfloor,\ \lambda n.\perp \rangle$

$\tau^2(\perp) = \langle \lambda n.\, \lfloor 1 \rfloor,\ \lambda n.\, ((\tau^1(\perp))_2(n+1)) \rangle =$

$\quad = \langle \lambda n.\, \lfloor 1 \rfloor,\ \lambda n.\, ((\lambda m.\perp)(n+1)) \rangle =$

$\quad = \langle \lambda n.\, \lfloor 1 \rfloor,\ \lambda n.\perp \rangle.$

Thus, we get that:

$$\varphi = \mathit{fix}(\tau) = \bigsqcup_{k \geq 0} \tau^k(\perp) = \langle \lambda n \in N.\, \lfloor 1 \rfloor,\ \lambda n \in N.\perp \rangle.$$

Therefore, $[\![f(g(1))]\!]\, \varphi\, \rho =$

$= \mathit{let}\ u \Leftarrow [\![g(1)]\!]\, \varphi\, \rho \centerdot (\lambda n \in N.\, \lfloor 1 \rfloor)(u) =$

$= \mathit{let}\ u \Leftarrow (\mathit{let}\ v \Leftarrow [\![1]\!]\, \varphi\, \rho \centerdot (\lambda n \in N.\perp)\, v) \centerdot (\lambda n \in N.\, \lfloor 1 \rfloor)(u) =$

$= \mathit{let}\ u \Leftarrow \perp \centerdot (\lambda n \in N.\, \lfloor 1 \rfloor)(u) =$

$= \perp \in N_\perp.$ □

We have the following result which establishes a correspondence between the operational and the denotational semantics in the call-by-value regime.

THEOREM 3.4. [**Equivalence of Operational and Denotational Semantics of REC. Call-by-value**] For all closed terms t, for all integers n, for all function environment φ, for all environment ρ, we have that $t \rightarrow^{va} n$ iff $[\![t]\!]^{va}\, \varphi\rho = \lfloor n \rfloor$.

Call-by-name Operational Semantics of the language REC.

$$n \rightarrow n$$

$$\frac{t_1 \rightarrow n_1 \quad t_2 \rightarrow n_2}{t_1 \,\mathbf{op}\, t_2 \rightarrow n}$$

where $n = n_1 \, op \, n_2$ and op is the semantic operation corresponding to **op**

$$\frac{t_0 \rightarrow 0 \qquad t_1 \rightarrow n_1}{\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \rightarrow n_1} \qquad \frac{t_0 \rightarrow n \quad t_2 \rightarrow n_2 \quad n \neq 0}{\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \rightarrow n_2}$$

$$\frac{d_i[t_1/x_1, ..., t_{a_i}/x_{a_i}] \rightarrow n}{f_i(t_1,...,t_{a_i}) \rightarrow n}$$

where d_i is the right hand side of the declaration of the function f_i.

TABLE 4. Call-by-name Operational Semantics of the language REC. For simplicity, we write $t_1 \rightarrow t_2$, instead of $t_1 \rightarrow^{na} t_2$.

4. Call-by-name Operational Semantics of REC

In this section we give the operational semantics of the language REC under the call-by-name regime. It will be given as a set of deduction rules which are shown in Table 4. Those rules define the rewriting relation $\rightarrow^{na} \subseteq \mathbf{Terms} \times \mathbf{Terms}$. The superscript ^{na} is for denoting the call-by-name regime. For reasons of simplicity, we will also write $t_1 \rightarrow t_2$, instead of $t_1 \rightarrow^{na} t_2$, for any two terms t_1 and t_2.

Note that in the call-by-name semantics we have that $f_i(t_1, \ldots, t_{a_i}) \rightarrow n$ holds if we have that $d_i[t_1/x_1, \ldots, t_{a_i}/x_{a_i}] \rightarrow n$ holds. In the call-by-name semantics the rule for function application does *not* enforce the evaluation of the arguments before the evaluation of the function declaration. Indeed, we do not require to establish that $t_1 \rightarrow n_1, \ldots, t_{a_i} \rightarrow n_{a_i}$, for some $n_1 \in N, \ldots, n_{a_i} \in N$.

As in the case of the call-by-value denotational semantics, for the call-by-name operational semantics we do not have a context-rule (see Remark 2.1 on page 166).

We have the following fact which tells us that the call-by-name operational semantics defines a function. We leave the proof of this fact to the reader. The proof can be done by rule induction.

FACT 4.1. [**Determinism of Call-by-name Operational Semantics**] For all term t, integers n_1 and n_2, if $t \rightarrow^{na} n_1$ and $t \rightarrow^{na} n_2$ then $n_1 = n_2$.

5. Call-by-name Denotational Semantics of REC

In this section we give the denotational semantics of the language REC under the call-by-name regime as a set of equations. These equations define a unique semantic function $[\![\_]\!]^{na}$ by structural induction.

The semantic domains for this semantic function are the following ones. Let N denote the discrete cpo of the integers and $N_\perp$ denote the flat cpo of the integers with the bottom element $\perp$.

(i) $\mathbf{Env}^{na}$ is the cpo $[\mathbf{Var} \to N_\perp]$ of the *environments*.
Thus, an environment $\rho \in \mathbf{Env}^{na}$ is a function from the discrete cpo **Var** of the integer variables to $N_\perp$.

Note that in the call-by-name semantics environments may map variables to $\perp$, contrary to the case of the call-by-value semantics where variables are mapped to integers only (recall that $\mathbf{Env}^{va}$ is the cpo $[\mathbf{Var} \to N]$). This is due to the fact that in the call-by-name semantics the evaluation of a function does not require the prior evaluation of its arguments and the value of some of those arguments may be $\perp$.

(ii) $\mathbf{Fenv}^{na}$ is the cpo of *function environments*.
For a given set of k declarations whose i-th declaration defines a function with arity a_i, $\mathbf{Fenv}^{na}$ is the cpo $[N_\perp^{a_1} \to N_\perp] \times \ldots \times [N_\perp^{a_k} \to N_\perp]$. Thus, a function environment $\varphi \in \mathbf{Fenv}^{na}$ is the product of k continuous functions. For $i = 1, \ldots, k$, the i-th projection of φ, denoted φ_i, is a continuous function in the cpo $[N_\perp^{a_i} \to N_\perp]$.

REMARK 5.1. The cpo $\mathbf{Env}^{na}$ of the environments and the cpo $\mathbf{Fenv}^{na}$ of the function environments of the call-by-name denotational semantics differ from those of the call-by-value denotational semantics. Indeed, variables and arguments of functions, respectively, may be bound to $\perp$ (in $N_\perp$) in the call-by-name semantics, while this is not the case in the call-by-value semantics (see page 168). This change of the domains of the environments and function environments is required by the fact that in the case of the call-by-name semantics, arguments of functions may be $\perp$, that is, values of non-terminating computations.

The denotational semantic function under the call-by-name regime will be denoted by $[\![\_]\!]^{na}$.
For all terms t, $[\![t]\!]^{na}$ is the unique continuous function in $[\mathbf{Fenv}^{na} \to [\mathbf{Env}^{na} \to N_\perp]]$, defined by structural induction by the equations of Table 5 on the facing page for any given function environment $\varphi \in \mathbf{Fenv}^{na}$ and any given environment $\rho \in \mathbf{Env}^{na}$.

When understood from the context we will avoid writing the superscript ^{na}.

As in the case of the call-by-value denotational semantics, for the call-by-name denotational semantics we do have a context-rule. Note also that, as usually in mathematics, the expressions which provide the call-by-name semantics, should be evaluated in the call-by-name regime. Thus, in particular, if the function $\varphi \in [N_\perp \to N_\perp]$ is the constant function $\lambda n.\lfloor 1 \rfloor$ then $\varphi(\perp) = \lfloor 1 \rfloor$.

5.1. Computation of the function environment in call-by-name.

The function environment $\varphi \in \mathbf{Fenv}^{na}$ for the call-by-name semantics, associated with a given set of declarations, can be computed analogously to what has been indicated for the call-by-value semantics on page 169. We leave this task as an exercise to the reader.

Call-by-name Denotational Semantics of the language REC.

$[\![n]\!]\,\varphi\,\rho \;=\; \lfloor n \rfloor$
$[\![x]\!]\,\varphi\,\rho \;=\; \rho(x)$
$[\![t_1 \,\mathbf{op}\, t_2]\!]\,\varphi\,\rho \;=\; [\![t_1]\!]\,\varphi\,\rho \;\; op_\perp \;\; [\![t_2]\!]\,\varphi\,\rho$
$[\![\mathbf{if}\, t_0 \,\mathbf{then}\, t_1 \,\mathbf{else}\, t_2]\!]\,\varphi\,\rho \;=\; Cond([\![t_0]\!]\,\varphi\,\rho, [\![t_1]\!]\,\varphi\,\rho, [\![t_2]\!]\,\varphi\,\rho)$
$[\![f_i(t_1, \ldots, t_{a_i})]\!]\,\varphi\,\rho \;=\; \varphi_i([\![t_1]\!]\,\varphi\,\rho, \ldots, [\![t_{a_i}]\!]\,\varphi\,\rho)$

TABLE 5. Call-by-name Denotational Semantics of the language REC. For simplicity, we write $[\![\_]\!]$, instead of $[\![\_]\!]^{na}$.

EXAMPLE 5.2. [**Factorial Function in Call-by-name**] As in the case of the call-by-value semantics, let us consider the following declaration:

$fact(x) = \mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x \times fact(x-1)$

The associated functional τ satisfies the following equation:

$\delta = (\lambda\chi.\, \lambda n \in N_\perp.\, [\![\mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times fact(x-1)]\!]\, \chi\, \rho[n/x])\, \delta$

and the value of function variable *fact* is the minimal fixpoint of the functional

$\tau =_{def} \lambda\chi.\, \lambda n \in N_\perp.\, [\![\mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times fact(x-1)]\!]\, \chi\, \rho[n/x].$

Thus, in this case $\mathbf{Fenv} = [N_\perp \to N_\perp]$ and the function environment $\varphi \in [N_\perp \to N_\perp]$ is the function $fix(\tau) = \bigsqcup_{k\geq 0} \tau^k(\perp)$. We have that:

$\tau^0(\perp) = \lambda n \in N_\perp.\, \perp$

$$\tau^1(\perp) = \lambda n \in N_\perp.\, Cond(n, \lfloor 1 \rfloor, n \times_\perp (\tau^0(\perp))(n -_\perp \lfloor 1 \rfloor)) = \begin{cases} \lfloor 1 \rfloor & \text{if } n = \lfloor 0 \rfloor \\ \perp & \text{otherwise} \end{cases}$$

$$\tau^2(\perp) = \lambda n \in N_\perp.\, Cond(n, \lfloor 1 \rfloor, n \times_\perp (\tau^1(\perp))(n -_\perp \lfloor 1 \rfloor)) = \begin{cases} \lfloor 1 \rfloor & \text{if } n = \lfloor 0 \rfloor \\ \lfloor 1 \rfloor & \text{if } n = \lfloor 1 \rfloor \\ \perp & \text{otherwise} \end{cases}$$

Then, as one can prove by mathematical induction, we get that, for any $k \geq 0$,

$$\tau^k(\perp) = \lambda n \in N_\perp. \begin{cases} \lfloor x! \rfloor & \text{if } n = \lfloor x \rfloor \text{ and } 0 \leq x < k \\ \perp & \text{otherwise} \end{cases}$$

Thus, $\varphi = fix(\tau) = \bigsqcup_{k\geq 0} \tau^k(\perp) = \lambda n \in N_\perp. \begin{cases} \lfloor x! \rfloor & \text{if } n = \lfloor x \rfloor \text{ and } 0 \leq x \\ \perp & \text{otherwise} \end{cases}$

The call-by-name evaluation of the factorial function for the input 3 is done as follows.

$[\![fact(3)]\!]\,\varphi\,\rho = \varphi([\![3]\!]\,\varphi\,\rho) = \varphi(\lfloor 3 \rfloor) = \lfloor 3! \rfloor = \lfloor 6 \rfloor.$ □

EXAMPLE 5.3. [**Constant Functions and Looping Functions in Call-by-name**] Let us consider the declaration:

$$\begin{cases} f(x) = 1 \\ g(x) = g(x+1) \end{cases}$$

The function environment φ is given by the pair of functions $\langle \varphi_1, \varphi_2 \rangle$ which belongs to the cpo $[N_\perp \to N_\perp] \times [N_\perp \to N_\perp]$, the function variable f being associated with φ_1 and the function variable g being associated with φ_2. We have that $\langle \varphi_1, \varphi_2 \rangle$ is the minimal fixpoint of the functional which transforms the pair $\langle \delta_1, \delta_2 \rangle$ of functions into itself, as specified by the following equation:

$$\langle \delta_1, \delta_2 \rangle = \langle \lambda\chi.\, \lambda n \in N_\perp.\, [\![1]\!]\, \chi\, \rho[n/x],\ \lambda\chi.\, \lambda n \in N_\perp.\, [\![g(x{+}1)]\!]\, \chi\, \rho[n/x] \rangle\ \langle \delta_1, \delta_2 \rangle$$

that is,

$$\langle \delta_1, \delta_2 \rangle = \langle \lambda\chi.\, \lambda n \in N_\perp.\, \lfloor 1 \rfloor,\ \lambda\chi.\, \lambda n \in N_\perp.\, \chi_2(n +_\perp \lfloor 1 \rfloor) \rangle\ \langle \delta_1, \delta_2 \rangle$$

where by χ_2 we have denoted the second component of the argument χ which is a pair. Thus, we have that $\langle \varphi_1, \varphi_2 \rangle$ is the minimal fixpoint of the following functional τ:

$$\tau =_{def} \langle \lambda\chi.\, \lambda n \in N_\perp.\, \lfloor 1 \rfloor,\ \lambda\chi.\, \lambda n \in N_\perp.\, \chi_2(n +_\perp \lfloor 1 \rfloor) \rangle.$$

As in the case of the call-by-value regime, the minimal fixpoint of τ is

$$\varphi = \mathit{fix}(\tau) = \bigsqcup_{k \geq 0} \tau^k(\perp) = \langle \lambda n \in N_\perp.\, \lfloor 1 \rfloor,\ \lambda n \in N_\perp.\, \perp \rangle.$$

Indeed, we have that

$$\begin{aligned}
&\tau^0(\perp) = \langle \lambda n \in N_\perp.\, \perp,\ \lambda n \in N_\perp.\, \perp \rangle \\
&\tau^1(\perp) = \langle \lambda n \in N_\perp.\, \lfloor 1 \rfloor,\ \lambda n \in N_\perp.\, ((\tau^0(\perp))_2(n +_\perp \lfloor 1 \rfloor)) \rangle = \\
&\qquad = \langle \lambda n \in N_\perp.\, \lfloor 1 \rfloor,\ \lambda n \in N_\perp.\, \perp \rangle \\
&\tau^2(\perp) = \langle \lambda n \in N_\perp.\, \lfloor 1 \rfloor,\ \lambda n \in N_\perp.\, ((\tau^1(\perp))_2(n +_\perp \lfloor 1 \rfloor)) \rangle = \\
&\qquad = \langle \lambda n \in N_\perp.\, \lfloor 1 \rfloor,\ \lambda n \in N_\perp.\, \perp \rangle.
\end{aligned}$$

Therefore, $[\![f(g(1))]\!]\, \varphi\, \rho =$

$$\begin{aligned}
&= (\lambda n \in N_\perp.\, \lfloor 1 \rfloor)\, ((\lambda n \in N_\perp.\, \perp)\, \lfloor 1 \rfloor) = \\
&= \lfloor 1 \rfloor \in N_\perp.
\end{aligned}$$

□

We have the following result which establishes a correspondence between the operational and the denotational semantics in the call-by-name regime.

THEOREM 5.4. **[Equivalence of Operational and Denotational Semantics of REC. Call-by-name]** For all closed terms t, for all integers n, for all function environment φ, for all environment ρ, we have that $t \to^{na} n$ iff $[\![t]\!]^{na}\, \varphi\rho = \lfloor n \rfloor$.

6. Proving Properties of Functions in the Language REC

In this section we will prove some properties of functions defined by declarations in the language REC. Let us begin by showing a fact which will be useful in Example 6.2 on the next page.

FACT 6.1. **[Call-by-value Equivalence of Two Functionals]** Let us consider a cpo D and the lifted cpo $D_\perp$. Let us also consider the functions $h : [D_\perp \to D_\perp]$, $\overline{h} : [D \to D]$, and $k : [D \to D]$, and the predicate $p : [D \to T_\perp]$, where $T_\perp$ is the lifted cpo of the discrete cpo $T = \{\mathit{true}, \mathit{false}\}$. Let us assume that: (i) the function h is strict, that is, $h(\perp) = \perp \in D_\perp$, and (ii) $\forall y \in D.\, h(\lfloor y \rfloor) = \lfloor \overline{h}(y) \rfloor$. The minimal fixpoints of the two functionals:

$$\tau_1\ \delta_1\ (x,y) = \ p(x) \to \lfloor y \rfloor \mid h(\delta_1(k(x),y))$$
$$\tau_2\ \delta_2\ (x,y) = \ p(x) \to \lfloor y \rfloor \mid \delta_2(k(x),\overline{h}(y))$$

where $\delta_1, \delta_2 \in [D{\times}D \to D_\perp]$, are equal. Recall that $b \to d_1 \,|\, d_2 \in [T_\perp{\times}D_\perp{\times}D_\perp \to D_\perp]$ stands for *let* $t \Leftarrow b\,.\,cond(t,d_1,d_2)$, where $cond \in [T \times D_\perp \times D_\perp \to D_\perp]$.

PROOF. We want to show that $\mathit{fix}(\tau_1) = \mathit{fix}(\tau_2)$. We do so by considering the inclusive predicate

$$P(f,g) =_{def} \forall x{\in}D, y{\in}D.\ f(x,y){=}g(x,y) \wedge g(x,\overline{h}(y)){=}h(g(x,y))$$

and by applying Scott induction using the continuous function which is the pair $\langle \tau_1, \tau_2 \rangle$ of the continuous functionals τ_1 and τ_2. Thus, if we get $P(\mathit{fix}(\langle \tau_1, \tau_2 \rangle))$, we also get $P(\mathit{fix}(\tau_1), \mathit{fix}(\tau_2))$, because $\mathit{fix}(\langle \tau_1, \tau_2 \rangle) = \langle \mathit{fix}(\tau_1), \mathit{fix}(\tau_2) \rangle$ (see Equation (†) on page 85).
We have to show that:
(i) $P(\perp, \perp)$ holds, and
(ii) for all $\delta_1, \delta_2 \in [D{\times}D \to D_\perp]$, if $P(\delta_1, \delta_2)$ holds then $P(\tau_1(\delta_1), \tau_2(\delta_2))$ holds.
For Point (i) we have to show that $\forall x \in D, y \in D.\ \perp(x,y) = \perp(x,y) \wedge \perp(x,\overline{h}(y)) = h(\perp(x,y))$ which is immediate because h is strict.
For Point (ii) we assume by induction hypothesis:

(1) $\forall x{\in}D, y{\in}D.\, \delta_1(x,y) = \delta_2(x,y)$ and
(2) $\forall x{\in}D, y{\in}D.\, \delta_2(x,\overline{h}(y)) = h(\delta_2(x,y))$

We have to show that $\forall x{\in}D, y{\in}D$,

$$p(x) \to \lfloor y \rfloor \quad \mid h(\delta_1(k(x),y)) \quad = \ p(x) \to \lfloor y \rfloor \quad \mid \delta_2(k(x),\overline{h}(y)) \qquad (\dagger 1)$$
$$p(x) \to h(\lfloor y \rfloor) \mid \delta_2(k(x),\overline{h}(\overline{h}(y))) = \ h(p(x) \to \lfloor y \rfloor \mid \delta_2(k(x),\overline{h}(y))) \qquad (\dagger 2)$$

For Equality (†1) we have that
$h(\delta_1(k(x),y)) = \{$by induction hypothesis (1)$\} =$
$= h(\delta_2(k(x),y)) = \{$by induction hypothesis (2)$\} =$
$= \delta_2(k(x),\overline{h}(y))$.
For Equality (†2) we have that
$\delta_2(k(x),\overline{h}(\overline{h}(y))) = \{$by induction hypothesis (2)$\} = h(\delta_2(k(x),\overline{h}(y)))$. (†3)
Thus,
$p(x) \to h(\lfloor y \rfloor) \mid \delta_2(k(x),h(h(y))) = \{$by (†3)$\} =$
$= p(x) \to h(\lfloor y \rfloor) \mid h(\delta_2(k(x),\overline{h}(y))) =$
$= \{$by Lemma 8.1 on page 112 because $h(\perp){=}\perp\} =$
$= h(p(x) \to \lfloor y \rfloor \mid \delta_2(k(x),\overline{h}(y)))$. □

In the following example we establish the equivalence of two function declarations for the call-by-value regime.

EXAMPLE 6.2. **[Call-by-value Equivalence of Sum Declarations]** Let N denote the discrete cpo of the integers $\{\ldots,-2,-1,0,1,2,\ldots\}$ and $N_\perp$ denote the flat

cpo of the integers with the bottom element $\bot$. Let us consider again the two declarations:

$$sum1(x,y) = \textbf{if } x \textbf{ then } y \textbf{ else } sum1(x-1,y)+1$$
$$sum2(x,y) = \textbf{if } x \textbf{ then } y \textbf{ else } sum2(x-1,y+1)$$

We want to show that under the call-by-value semantics they define the same functions. In the call-by-value semantics those declarations define two functionals, called them τ_1 and τ_2, satisfying the following equations:

$$\tau_1\ \delta_1\ (x,y) = Cond(\lfloor x\rfloor,\ \lfloor y\rfloor,\ \delta_1(x-1,y)+_\bot\lfloor 1\rfloor) \quad \text{where } \delta_1 \in [N\times N\to N_\bot]$$
$$\tau_2\ \delta_2\ (x,y) = Cond(\lfloor x\rfloor,\ \lfloor y\rfloor,\ \delta_2(x-1,y+1)) \quad \text{where } \delta_2 \in [N\times N\to N_\bot]$$

(For reasons of simplicity we have made some simplifications of the *let* construct and, in particular, we have written $x-1$, instead of $down(\lfloor x\rfloor -_\bot \lfloor 1\rfloor)$). Now, if we indicate by φ_1 and φ_2 the minimal fixpoints of τ_1 and τ_2, respectively, and we use the $\_\to\_\,|\,\_$ construct, instead of *Cond*, we get:

$$\varphi_1(x,y) = \lfloor x=0\rfloor \to \lfloor y\rfloor \mid \varphi_1(x-1,y)+_\bot\lfloor 1\rfloor$$
$$\varphi_2(x,y) = \lfloor x=0\rfloor \to \lfloor y\rfloor \mid \varphi_2(x-1,y+1).$$

Then, by using Fact 6.1 on page 176, where we take: (i) D to be N, (ii) $D_\bot$ to be $N_\bot$, (iii) $h \in [N_\bot \to N_\bot]$ to be $\lambda x.x+_\bot\lfloor 1\rfloor$, (iv) $\overline{h} \in [N\to N]$ to be $\lambda x.x+1$, and (v) $k \in [N\to N]$ is $\lambda x.x-1$, and (vi) $p \in [N\to T_\bot]$ to be $\lambda x.\lfloor x=0\rfloor$, we conclude that $\forall x,y \in N.\, \varphi_1(x,y)=\varphi_2(x,y)$.

EXAMPLE 6.3. [**McCarthy 91 Function. Call-by-value. Denotational Semantics**] Let us consider the McCarthy 91 function defined for all (negative, or null, or positive) integers $n \in N$ as follows:

$$f(n) = \textbf{if } n>100 \textbf{ then } n-10 \textbf{ else } f(f(n+11))$$

Note that, if we use the usual semantics of $>$, this declaration is *not* a legal declaration in the language REC, but it can be rewritten as a legal declaration as follows:

$$f(n) = \textbf{if } p(n) \textbf{ then } n-10 \textbf{ else } f(f(n+11)) \qquad (\dagger 1)$$

where p denotes a function from N to $N_\bot$ such that for all n in the set N of integers, $[\![p(n)]\!]^{va}\varphi\rho = cond(n>100, \lfloor 0\rfloor\lfloor 1\rfloor)$. (Recall that $cond \in [\{true, false\}\times N_\bot\times N_\bot \to N_\bot]$.) The declaration of the function p can be given in the language REC as follows:

$$p(n) = \textbf{if } q(n-101) \textbf{ then } 0 \textbf{ else } 1$$
$$q(n) = n - sqroot(n\times n)$$
$$sqroot(n) = \textbf{if } n \textbf{ then } 0 \textbf{ else } sqroot1(n,1)$$
$$sqroot1(n,m) = \textbf{if } n-(m\times m) \textbf{ then } m \textbf{ else } sqroot1(n,m+1)$$

Note that the argument of the function *sqroot* is always the square of an integer. Indeed, we have that for all $n \in N$: (i) $sqroot(n\times n)$ is the absolute value of n, and (ii) $q(n)=0$ iff $n\geq 0$. Thus, $q(n-101) = 0$ iff $n>100$, and we get that $p(n) = 0$ iff $n>100$.

A different approach to make the declaration (†1) to be a legal declaration in the language REC is to assume that the operator $>$ is a primitive operator (such as $+$,

$-$, and $\times$) whose semantics is defined as follows: for all $n, m \in N$, $(n>m) = 0$ iff n is greater than m.

The continuous functional τ associated with the declaration of the function f in the call-by-value semantics, is as follows:

$$\tau =_{def} \lambda\delta.\lambda n.\ cond(n>100, \lfloor n-10 \rfloor,\ let\ v \Leftarrow \big(let\ u \Leftarrow [\![n+11]\!]\ \delta\ \rho\ .\ \delta(u)\big)\ .\ \delta(v)) =$$
$$=_{def} \lambda\delta.\lambda n.\ cond(n>100, \lfloor n-10 \rfloor,\ let\ v \Leftarrow \delta(n+11)\ .\ \delta(v))$$

We want to show that the minimal fixpoint of τ is the function, call it g, defined under the call-by-value semantics by the following declaration (where we used the identifier g_{91}, instead of g, to distinguish syntax from semantics):

$$g_{91}(n) = \textbf{if}\ p(n)\ \textbf{then}\ n-10\ \textbf{else}\ 91 \qquad (\dagger 2)$$

Thus, g is the continuous function in $[N \to N_\perp]$ such that for all $n \in N$:

$$g(n) =_{def}\ cond(n>100, \lfloor n-10 \rfloor, \lfloor 91 \rfloor). \qquad (\dagger 3)$$

We will show that for every function $\delta \in [N \to N_\perp]$ which is a fixpoint of τ (and, thus, $\tau\delta = \delta$), we have that for all $n \in N$, $\delta(n) = g(n)$. Thus, also the minimal fixpoint of τ is equal to the function g defined by Equation (†3).

Moreover, since for all $n \in N$, $g(n) \neq \perp$ (as it can easily be proved by considering the cases $n>100$ and $n \leq 100$), there is a unique fixpoint of τ.

The proof that for all $n \in N$, $\delta(n) = g(n)$ is done by well-founded induction on the integers N according to the well-founded order $\prec \subseteq N \times N$ defined as follows:

$$\forall m, n \in N.\ n \prec m\ \textit{iff}\ m < n \leq 101$$

First note that the order $\prec$ is well-founded, that is, there is no infinite descending sequence $\ldots \prec n_2 \prec n_1 \prec n_0$ in N. Indeed,
(i) if $n_0 \geq 101$ then no n_1 exists such that $n_1 \prec n_0$ because no n_1 exists such that $n_0 < n_1 \leq 101$, and
(ii) if $n_0 < 101$ then every sequence of the form $n_0 < n_1 < \ldots \leq 101$ which is necessarily finite, gives us a sequence $\ldots \prec n_1 \prec n_0$ which is necessarily finite.

Take any $n \in N$.

There are three cases: (i) $n > 100$, (ii) $90 \leq n \leq 100$, and (iii) $n < 90$.

Case (i): $n > 100$. Since $\delta(n) = \tau(\delta)(n)$, we have that

$$\delta(n) = cond(n>100, \lfloor n-10 \rfloor,\ let\ v \Leftarrow \delta(n+11)\ .\ \delta(v)).$$

Thus, in this case $\delta(n) = \lfloor n-10 \rfloor$ and $\delta(n) = g(n)$.

Case (ii): $90 \leq n \leq 100$. Since $\delta(n) = \tau(\delta)(n)$, we have that

$$\delta(n) = let\ v \Leftarrow \delta(n+11)\ .\ \delta(v) = \{\text{since } n+11 > 100\} =$$
$$= let\ v \Leftarrow \lfloor n+11-10 \rfloor\ .\ \delta(v) =$$
$$= \delta(n+1) =$$
$$= \{\text{by induction hypothesis because } n+1 \prec n \text{ (note that } n < n+1 \leq 101)\} =$$
$$= g(n+1) = \lfloor 91 \rfloor.$$

Case (iii): $n < 90$. Since $\delta(n) = \tau(\delta)(n)$, we have that

$$\delta(n) = let\ v \Leftarrow \delta(n+11)\ .\ \delta(v) =$$
$$= \{\text{by induction hypothesis because } n+11 \prec n \text{ (note that } n < n+11 \leq 101)\} =$$
$$= let\ v \Leftarrow cond(n+11>100, \lfloor n+11-10 \rfloor, \lfloor 91 \rfloor)\ .\ \delta(v) = \{\text{since } n+11 \leq 100\} =$$

$= \mathit{let}\, v \Leftarrow \lfloor 91 \rfloor \,.\, \delta(v) =$
$= \delta(91) = \{\text{by Case (ii)}\} = \lfloor 91 \rfloor.$

EXERCISE 6.4. Redo Example 6.3 on page 178 in the call-by-name semantics by considering the functional

$\chi =_{def} \lambda\delta.\, \lambda n \in N_\perp .\, n >_\perp \lfloor 100 \rfloor \;\rightarrow\; n -_\perp \lfloor 10 \rfloor \;\mid\; \delta(\delta(n +_\perp \lfloor 11 \rfloor))$

in $[[N_\perp \rightarrow N_\perp] \rightarrow [N_\perp \rightarrow N_\perp]]$. The order $>_\perp \subseteq N_\perp \times N_\perp$ is defined as follows:

$$\begin{aligned} n >_\perp m &= \mathit{true} && \mathit{if}\ \mathit{down}(n) > \mathit{down}(m) \\ &= \mathit{false} && \mathit{if}\ \mathit{down}(n) \leq \mathit{down}(m) \\ &= \perp && \mathit{if}\ n = \perp\ \mathit{or}\ m = \perp \end{aligned}$$

Therefore, you should show that for every fixpoint δ of χ, for every $n \in N_\perp$, $\delta(n) = g(n)$, where $g(n) =_{def} n >_\perp \lfloor 100 \rfloor \;\rightarrow\; n -_\perp \lfloor 10 \rfloor \;\mid\; \lfloor 91 \rfloor$.

EXERCISE 6.5. **[McCarthy 91 Function. Call-by-value. Operational Semantics]** Consider the function declaration

$f(n) = \textbf{if}\ n > 100\ \textbf{then}\ n-10\ \textbf{else}\ f(f(n+11))$.

Show that for all (negative, null, and positive) integers $n \in N$, $f(n) \rightarrow^{va} m$, for some $m \geq 91$. As already mentioned in Example 6.3 on page 178, this declaration of the function f is *not* a legal declaration in the language REC, and in order to make it legal we will consider $>$ as a primitive operator whose semantics is defined as follows: for all $n, m \in N$, $(n > m) = 0$ iff n is an integer larger than m.

Hint. Consider the well-founded order $\prec$ defined in Example 6.3, and the three cases: (i) $n \leq 89$, (ii) $89 < n \leq 100$, and (iii) $n > 100$, as indicated in Example 6.3. □

EXAMPLE 6.6. **[Ackermann Function: Call-by-value. (1) Termination Implies Computation of Positive Values]** Let N denote the set of all (negative, or null, or positive) integers. Let us consider the Ackermann function which has the following declaration in the language REC, where $x, y \in N$:

$$\begin{aligned} \mathit{Ack}(x,y) = \ & \textbf{if}\ x\ \textbf{then}\ y+1\ \textbf{else} \\ & \textbf{if}\ y\ \textbf{then}\ \mathit{Ack}(x-1,1)\ \textbf{else} \\ & \mathit{Ack}(x-1, \mathit{Ack}(x, y-1)) \end{aligned} \qquad (\ddagger)$$

The functional $\tau^{va} \in \big[[N \times N \rightarrow N_\perp] \rightarrow [N \times N \rightarrow N_\perp]\big]$ associated with that declaration of the Ackermann function in the call-by-value semantics is as follows:

$$\begin{aligned} \tau^{va} =_{def} \lambda\delta.\, \lambda m \in N.\, \lambda n \in N.\, & \mathit{Cond}\big(\lfloor m \rfloor, \lfloor n+1 \rfloor, \\ & \mathit{Cond}\big(\lfloor n \rfloor,\ \mathit{let}\ p \Leftarrow [\![x-1]\!]\, \delta\, \rho[m/x,\, n/y], \\ & \qquad\qquad q \Leftarrow [\![1]\!]\, \delta\, \rho[m/x,\, n/y] \,.\, \delta(p,q), \\ & \mathit{let}\ k \Leftarrow [\![x-1]\!]\, \delta\, \rho[m/x,\, n/y], \\ & \quad \ell \Leftarrow \big(\mathit{let}\ r \Leftarrow [\![x]\!]\, \delta\, \rho[m/x,\, n/y], \\ & \qquad\qquad s \Leftarrow [\![y-1]\!]\, \delta\, \rho[m/x,\, n/y] \,.\, \delta(r,s)\big) \,.\, \delta(k,\ell)\big)\big) \end{aligned}$$

After some simplifications we get:

$$\begin{aligned} \tau^{va} =_{def}\ \lambda\delta.\, \lambda m \in N.\, \lambda n \in N.\ & \mathit{Cond}\big(\lfloor m \rfloor, \lfloor n+1 \rfloor, \\ & \mathit{Cond}\big((\lfloor n \rfloor,\ \delta(m-1,1), \\ & \mathit{let}\ \ell \Leftarrow \delta(m, n-1) \,.\, \delta(m-1, \ell)\big)\big) \end{aligned} \qquad (\ddagger 1)$$

We want to show that the following property holds:

$\forall m{\in}N, n{\in}N.\, \mathit{if}(m{\geq}0 \wedge n{\geq}0)\ \mathit{then}\ \mathit{fix}(\tau^{va})(m,n){\neq}\bot \ \rightarrow\ \mathit{fix}(\tau^{va})(m,n){>_\bot}\lfloor 0\rfloor$

where for all $x, y \in N$, $\lfloor x\rfloor >_\bot \lfloor y\rfloor$ iff $x>y$. That is, by using a more concise notation, we want to show that:

$\forall m, n{\geq}0.\, P(\mathit{fix}(\tau^{va}), m, n)$ holds, where:

(i) for all functions $f \in [N{\times}N \rightarrow N_\bot]$, for all $m, n \geq 0$, the property $P(f, m, n)$ is defined as follows:

$$P(f,m,n) =_{def}\ f(m,n){\neq}\bot \ \rightarrow\ f(m,n){>_\bot}\lfloor 0\rfloor \qquad (\ddagger 2)$$

and, as usual,

(ii) $\mathit{fix}(\tau^{va}) \in [N{\times}N \rightarrow N_\bot]$ is the minimal fixpoint of the functional τ^{va}.

Thus, by using Scott induction we have to show:

(i) $\forall m, n{\geq}0.\, P(\bot, m, n)$, where $\bot$ is the everywhere undefined function $\lambda m.\lambda n.\bot$ in $[N{\times}N \rightarrow N_\bot]$, and

(ii) for all $f \in [N{\times}N \rightarrow N_\bot]$, if $\forall m, n{\geq}0.\, P(f, m, n)$ then $\forall m, n{\geq}0.\, P(\tau^{va}(f), m, n)$.

The proof of Point (i) is obvious because for $f = \lambda m.\lambda n.\bot$, for all $m, n \geq 0$, the premise of the implication ($\ddagger 2$) is false.

The proof of Point (ii) is as follows. Take any function $f \in [N{\times}N \rightarrow N_\bot]$. Let us assume that:

$$\forall m, n{\geq}0.\, P(f, m, n). \qquad (\mathrm{H})$$

We have to show that: $\forall m, n{\geq}0.\, P(\tau^{va}(f), m, n)$. Let us take any $m, n \geq 0$. There are three cases: (ii.1) $m{=}0 \wedge n{\geq}0$, (ii.2) $m{>}0 \wedge n{=}0$, and (ii.3) $m{>}0 \wedge n{>}0$.

Case (ii.1): $m = 0 \wedge n \geq 0$. In this case $\tau^{va}(f) = \lambda m.\lambda n.\lfloor n{+}1\rfloor$. Thus, in order to show $P(\tau^{va}(f), m, n)$, it is enough to show that $\lfloor n{+}1\rfloor >_\bot \lfloor 0\rfloor$. This is obvious because we have that $n{\geq}0$.

Case (ii.2): $m{>}0 \wedge n{=}0$. In this case $\tau^{va}(f){=}\lambda m.\lambda n.\, f(m{-}1, 1)$. Thus, in order to show $P(\tau^{va}(f), m, n)$, it is enough to show that:

$$f(m{-}1,1){\neq}\bot \rightarrow f(m{-}1,1){>_\bot}\lfloor 0\rfloor. \qquad (\ddagger 3)$$

By the hypothesis (H) we have that ($\ddagger 3$) holds.

Case (ii.3): $m{>}0 \wedge n{>}0$. In this case

$$\tau^{va}(f) = \lambda m.\lambda n.\, (\mathit{let}\ \ell \Leftarrow f(m,n{-}1)\,\centerdot\, f(m{-}1,\ell)).$$

We have to show $P(\tau^{va}(f), m, n)$, that is:

$$\big(\mathit{let}\ \ell \Leftarrow f(m,n{-}1)\,\centerdot\, f(m{-}1,\ell){\neq}\bot\big) \ \rightarrow\ \big(\mathit{let}\ \ell \Leftarrow f(m,n{-}1)\,\centerdot\, f(m{-}1,\ell){>_\bot}\lfloor 0\rfloor\big).$$

Thus, we assume:

$$\mathit{let}\ \ell \Leftarrow f(m,n{-}1)\,\centerdot\, f(m{-}1,\ell){\neq}\bot \qquad (\mathrm{H1})$$

and we have to show:

$$\mathit{let}\ \ell \Leftarrow f(m,n{-}1)\,\centerdot\, f(m{-}1,\ell){>_\bot}\lfloor 0\rfloor. \qquad (\mathrm{T})$$

Hypothesis (H1) is equivalent to:

$$f(m,n{-}1) \neq \bot \text{ and} \qquad (\mathrm{H1.1})$$

$$f(m{-}1,\ell) \neq \bot, \text{ where } \lfloor\ell\rfloor = f(m,n{-}1). \qquad (\mathrm{H1.2})$$

Now by the hypothesis (H) we have that $P(f, m, n{-}1)$ holds, that is,

$$f(m,n{-}1) \neq \bot \;\rightarrow\; f(m,n{-}1) >_\bot \lfloor 0 \rfloor \tag{‡4}$$

holds. From (‡4) and (H1.1) we get:

$$f(m,n{-}1) >_\bot \lfloor 0 \rfloor \tag{‡5}$$

that is, there exists $\ell > 0$ such that $\lfloor \ell \rfloor = f(m,n{-}1)$.

From (‡5) we have that in order to show (T) is enough to show that:

$$f(m{-}1,\ell) >_\bot \lfloor 0 \rfloor \text{ where } \lfloor \ell \rfloor = f(m,n{-}1). \tag{T1}$$

This statement (T1) is a consequence of the hypotheses (H) and (H1.2). □

EXAMPLE 6.7. [**Ackermann Function: Call-by-value. (2) Termination**] We want to show that for all functions $\delta \in [N \times N \rightarrow N_\bot]$ satisfying the following equation:

$$\begin{array}{l} \delta\,(m,n) = \mathit{Cond}\big(\lfloor m \rfloor, \lfloor n{+}1 \rfloor, \\ \qquad\qquad \mathit{Cond}\big(\lfloor n \rfloor, \delta(m{-}1,1), \\ \qquad\qquad \mathit{let}\,\ell \Leftarrow \delta(m,n{-}1)\,\centerdot\,\delta(m{-}1,\ell)\big)\big) \end{array} \tag{‡6}$$

that is, for all functions δ which are fixpoints of Equation (‡6), we have that $\forall m,n \geq 0.\, \delta(m,n) \neq \bot$. Recall that N denotes the set of all negative, or null, or positive integers.

The proof is by well-founded induction using the well-founded lexicographic order $<_{lex} \subseteq N^{\geq 0} \times N^{\geq 0}$, where $N^{\geq 0} =_{def} \{n \mid n \in N \wedge n \geq 0\}$.

There are three cases: (i) $m = 0 \wedge n \geq 0$, (ii) $m > 0 \wedge n = 0$, and (iii) $m > 0 \wedge n > 0$.

Case (i): $m = 0 \wedge n \geq 0$. Since δ satisfies Equation (‡6), in this case we have that $\delta(m,n) = \lfloor n{+}1 \rfloor$. Thus, $\delta(m,n) \neq \bot$.

Case (ii): $m > 0 \wedge n = 0$. Since δ satisfies Equation (‡6), in this case we have that $\delta(m,n) = \delta(m{-}1,1)$. Since $\langle m{-}1,1\rangle <_{lex} \langle m,n\rangle$ by induction hypothesis we have that $\delta(m{-}1,1) \neq \bot$. Thus, $\delta(m,n) \neq \bot$.

Case (iii): $m > 0 \wedge n > 0$. Since δ satisfies Equation (‡6), in this case we have that $\delta(m,n) = \mathit{let}\,\ell \Leftarrow f(m,n{-}1)\,\centerdot\, f(m{-}1,\ell)$. Since $\langle m,n{-}1\rangle <_{lex} \langle m,n\rangle$ by induction hypothesis we have that $\delta(m,n{-}1) \neq \bot$ and thus, $\delta(m,n{-}1) = \lfloor \ell \rfloor$ for some $\ell \in N^{\geq 0}$. Hence, $\delta(m,n) = \delta(m{-}1,\ell)$ for some $\ell \in N^{\geq 0}$. Since $\langle m{-}1,k\rangle <_{lex} \langle m,n\rangle$ for all $k \in N^{\geq 0}$, by induction hypothesis we have that $\delta(m{-}1,\ell) \neq \bot$. Thus, we get that $\delta(m,n) \neq \bot$. □

The following Fact 6.7 is a consequence of the previous two examples (Example 6.6 on page 180 and Example 6.7) by taking into account that: (i) the minimal fixpoint of the functional τ^{va} for the Ackermann function (see Definition (‡1) on page 180) is a particular function δ satisfying Equation (‡6) on this page, and (ii) $\forall x\,(A(x) \rightarrow B(x))$ implies $(\forall x\, A(x)) \rightarrow (\forall x\, B(x))$.

FACT 6.8. [**Ackermann Function: Call-by-value. (3) Termination and Computation of Positive Values**] In the call-by-value regime, the Ackermann function terminates for all natural numbers m and n. Formally, given the following functional $\tau^{va} \in \big[[N{\times}N \rightarrow N_\bot] \rightarrow [N{\times}N \rightarrow N_\bot]\big]$ associated with the Declaration (‡) on page 180:

$$\tau^{va} =_{def} \lambda\delta.\,\lambda m{\in}N.\,\lambda n{\in}N.\; \mathit{Cond}\big(\lfloor m\rfloor, \lfloor n{+}1\rfloor,\; \mathit{Cond}\big((\lfloor n\rfloor,\; \delta(m{-}1, 1),\; \mathit{let}\,\ell \Leftarrow \delta(m, n{-}1)\,\centerdot\,\delta(m{-}1, \ell)\big)\big)$$

we have that $\forall m, n \geq 0.\, \mathit{fix}(\tau^{va})(m, n) >_{\perp} \lfloor 0\rfloor$. By the Unique Fixpoint Principle (see page 107), (i) for all $m, n \geq 0$, there exists a *unique fixpoint* of τ^{va} (and thus, it is equal to the minimal fixpoint $\mathit{fix}(\tau^{va})$), and (ii) for all $m, n \geq 0$, that unique fixpoint of τ^{va} applied to m and n, is different from $\perp \in N_{\perp}$. All fixpoints of τ^{va} differ only for the values they return for $m{<}0$ or $n{<}0$. □

Thus, from this Fact 6.8 on the preceding page and Theorem 3.4 on page 172, we have that for all $m, n \geq 0$, there exists a natural number k such that $Ack(m, n) \rightarrow^{va} k$ (see Declaration (‡) of the Ackermann function on page 180).

Fact 6.8 can be extended to the call-by-name regime because, as the reader may verify, what we have shown for the call-by-value regime in Example 6.6 on page 180 and Example 6.7 on the facing page, also holds for the call-by-name regime. Thus, we have the following fact.

Fact 6.9. [**Ackermann Function: Call-by-name. (4) Termination and Computation of Positive Values**] In the call-by-name regime, the Ackermann function terminates for all natural numbers m and n. Formally, given the following functional $\tau^{na} \in \big[[N_{\perp} \times N_{\perp} \rightarrow N_{\perp}] \rightarrow [N_{\perp} \times N_{\perp} \rightarrow N_{\perp}]\big]$ associated with the Declaration (‡) on page 180:

$$\tau^{na} =_{def} \lambda\delta.\,\lambda m{\in}N_{\perp}.\,\lambda n{\in}N_{\perp}.\; \mathit{Cond}\big(m, n{+_{\perp}}\lfloor 1\rfloor,\; \mathit{Cond}\big((n,\; \delta(m{-_{\perp}}\lfloor 1\rfloor, \lfloor 1\rfloor),\; \delta(m{-_{\perp}}\lfloor 1\rfloor, \delta(m, n{-_{\perp}}\lfloor 1\rfloor))\big)\big) \tag{‡7}$$

we have that $\forall m, n \geq 0.\, \mathit{fix}(\tau^{na})(\lfloor m\rfloor, \lfloor n\rfloor) >_{\perp} \lfloor 0\rfloor$. By the Unique Fixpoint Principle (see page 107), (i) for all $m, n \geq 0$, there exists a *unique fixpoint* of τ^{na} (and thus, it is equal to the minimal fixpoint $\mathit{fix}(\tau^{na})$), and (ii) for all $m, n \geq 0$, that unique fixpoint of τ^{na} applied to $\lfloor m\rfloor$ and $\lfloor n\rfloor$ is different from $\perp \in N_{\perp}$. All fixpoints of τ^{na} differ only for the values they return for $m{<}0$ or $n{<}0$. □

Exercise 6.10. Consider the functional τ^{va} (see Definition (‡1) on page 180) and the functional τ^{na} (see Definition (‡7) on page 183).

Show that $\forall m, n \geq 0$, $\mathit{fix}(\tau^{va})(m, n) = \mathit{fix}(\tau^{na})(\lfloor m\rfloor, \lfloor n\rfloor)$.

Hint. By induction on m and n. □

In the following Fact 6.13 on page 186 we will extend the termination results of Fact 6.8 and Fact 6.9. We will show strong termination (see page 42) of a rewriting system associated with the Ackermann function. For that rewriting system, in fact, given any initial term of the form $Ack(m, n)$, there exist the natural numbers r and v such that $Ack(m, n) \longrightarrow^{r} v$ *under any evaluation regime*, that is, for every possible choice of the subterm to be rewritten at every rewriting step. Thus, in particular, termination will be guaranteed when at every rewriting step we replace the innermost subterm or the outermost subterm and these regimes correspond, respectively, to the call-by-value and the call-by-name regimes (see also Section 7 on page 192).

In Fact 6.13 we will assume a binary relation, denoted $\longrightarrow$, which is defined by a set of rewriting rules and, as usual, we will also assume the following context-rule for that rewriting relation:

for every term t and t', for every context $C[\_]$,

$$\frac{t \longrightarrow t'}{C[t] \longrightarrow C[t']}$$

Recall that a context $C[\_]$ is a term 'with a missing subterm' (see Remark 2.1 on page 166).

The following definition introduces the rewriting system associated with the Ackermann function.

DEFINITION 6.11. [**Ackermann Rewriting Rules**] Let us consider the following rewriting rules $R1$–$R3$ are associated with the Ackermann function. The variables m and n range over the set of terms t of the form

$t ::= 0 \mid succ(t) \mid t + t \mid Ack(t, t)$

We will write n as an abbreviation for $succ^n(0)$, for any $n \in N$. In particular 1 is an abbreviation for $succ(0)$. The term $t{+}1$ stands for $succ(t)$, for any term t.

We will also implicitly use commutativity and associativity of $+$.

$$\begin{array}{lll} R1. & Ack(0,n) & \longrightarrow n{+}1 \\ R2. & Ack(m{+}1,0) & \longrightarrow Ack(m,1) \\ R3. & Ack(m{+}1,n{+}1) & \longrightarrow Ack(m,Ack(m{+}1,n)) \end{array}$$ □

Whenever the term t_{i+1} is derived from the term t_i by applying the rewriting rule Rk, for $k = 1, 2$, and 3, we will write $t_i \longrightarrow_k t_{i+1}$. For instance, starting from the term $Ack(1,2)$, we have the following three sequences of terms, where at every rewriting step we have underlined the term which has been replaced (see also Section 7 on page 192).

(A) Call-by-name regime (outermost call):

$$\begin{array}{lll} \underline{Ack(1,2)} & \longrightarrow_3 \underline{Ack(0,Ack(1,1))} & \longrightarrow_1 \underline{Ack(1,1)}{+}1 \\ & \longrightarrow_3 \underline{Ack(0,Ack(1,0))}{+}1 & \longrightarrow_1 \underline{Ack(1,0)}{+}1{+}1 \\ & \longrightarrow_2 \underline{Ack(0,1)}{+}1{+}1 & \longrightarrow_1 1{+}1{+}1{+}1 = 4 \end{array}$$

(B) Call-by-value regime (innermost call):

$$\begin{array}{lll} \underline{Ack(1,2)} & \longrightarrow_3 Ack(0,\underline{Ack(1,1)}) & \longrightarrow_3 Ack(0,Ack(0,\underline{Ack(1,0)})) \\ & \longrightarrow_2 Ack(0,Ack(0,\underline{Ack(0,1)})) & \longrightarrow_1 Ack(0,\underline{Ack(0,1{+}1)}) \\ & \longrightarrow_1 \underline{Ack(0,1{+}1{+}1)} & \longrightarrow_1 1{+}1{+}1{+}1 = 4 \end{array}$$

(C) A regime which is neither call-by-name nor call-by-value:

$$\begin{array}{lll} \underline{Ack(1,2)} & \longrightarrow_3 Ack(0,\underline{Ack(1,1)}) & \longrightarrow_3 Ack(0,\underline{Ack(0,Ack(1,0))}) \\ & \longrightarrow_1 Ack(0,\underline{Ack(1,0)}{+}1) & \longrightarrow_2 \underline{Ack(0,Ack(0,1){+}1)} \\ & \longrightarrow_1 \underline{Ack(0,1)}{+}1{+}1 & \longrightarrow_1 1{+}1{+}1{+}1 = 4 \end{array}$$

Now, in the following Fact 6.12 we will prove some properties of the Ackermann function defined as the minimal fixpoint of the functional τ^{na} (see Definition (‡7) on page 183). We leave it to the reader to show that analogous properties hold for the Ackermann function defined as the minimal fixpoint of the functional τ^{va} (see Definition (‡1) on page 180).

Then in Fact 6.13 on the next page we will show strong termination (see Definition 9.7 on page 42) of the rewriting system of Definition 6.11 on the preceding page.

FACT 6.12. [**Monotonicities of the Ackermann Function**] Let the function $\lambda m \in N.\, \lambda n \in N.\, A(\lfloor m \rfloor, \lfloor n \rfloor) \in [N_\perp \times N_\perp \to N_\perp]$ be the minimal fixpoint of the functional τ^{na} (see Definition (‡7) on page 183). We have the following inequalities:

(F1) $\forall m, n \geq 0,\ A(\lfloor m \rfloor, \lfloor n \rfloor) >_\perp \lfloor n \rfloor$
(F2) $\forall m, n \geq 0,\ A(\lfloor m \rfloor, \lfloor n+1 \rfloor) >_\perp A(\lfloor m \rfloor, \lfloor n \rfloor)$ (Right Monotonicity)
(F3) $\forall m, n \geq 0,\ A(\lfloor m+1 \rfloor, \lfloor n \rfloor) >_\perp A(\lfloor m \rfloor, \lfloor n \rfloor)$ (Left Monotonicity)

where the relation $>_\perp$ is the strict extension of the usual $>$ relation between natural numbers (see also Exercise 6.4 on page 180).

PROOF. (F1) By induction on $m \in N$.
(*Basis*) We have to prove that $\forall n \in N.\, A(\lfloor 0 \rfloor, \lfloor n \rfloor) >_\perp \lfloor n \rfloor$. This is obvious, because by definition of τ^{na}, $\forall n \in N.\, A(\lfloor 0 \rfloor, \lfloor n \rfloor) = \lfloor n+1 \rfloor$.
(*Step*) We take any $m \in N$ and we assume (α): $\forall n \in N.\, A(\lfloor m \rfloor, \lfloor n \rfloor) >_\perp \lfloor n \rfloor$ and we have to show: $\forall n \in N.\, A(\lfloor m+1 \rfloor, \lfloor n \rfloor) >_\perp \lfloor n \rfloor$. We prove this by induction on $n \in N$.
(*Basis*) $A(\lfloor m+1 \rfloor, \lfloor 0 \rfloor) = \{$by definition of $\tau^{na}\} = A(\lfloor m \rfloor, \lfloor 1 \rfloor) = \{$by induction hypothesis (α)$\} >_\perp \lfloor 1 \rfloor$. Thus, $A(\lfloor m+1 \rfloor, \lfloor 0 \rfloor) >_\perp \lfloor 0 \rfloor$.
(*Step*) We take any $n \in N$. We assume (β): $A(\lfloor m+1 \rfloor, \lfloor n \rfloor) >_\perp \lfloor n \rfloor$. We have to show: $A(\lfloor m+1 \rfloor, \lfloor n+1 \rfloor) >_\perp \lfloor n+1 \rfloor$. Indeed,
$A(\lfloor m+1 \rfloor, \lfloor n+1 \rfloor) = \{$by definition of $\tau^{na}\} =$
$= A(\lfloor m \rfloor, A(\lfloor m+1 \rfloor, \lfloor n \rfloor)) = \{$by induction hypothesis (α)$\} =$
$>_\perp A(\lfloor m+1 \rfloor, \lfloor n \rfloor) = \{$by induction hypothesis (β)$\} >_\perp \lfloor n \rfloor$.
Thus, $A(\lfloor m+1 \rfloor, \lfloor n+1 \rfloor) >_\perp \lfloor n+1 \rfloor$. (Note that if $\lfloor n_1 \rfloor >_\perp \lfloor n_2 \rfloor >_\perp \lfloor n_3 \rfloor$ then $\lfloor n_1 \rfloor >_\perp \lfloor n_3+1 \rfloor$.)

(F2) By induction on $m \in N$.
(*Basis*) We have to prove: $A(\lfloor 0 \rfloor, \lfloor n+1 \rfloor) >_\perp A(\lfloor 0 \rfloor, \lfloor n \rfloor)$. This is obvious, because by definition of τ^{na}, $A(\lfloor 0 \rfloor, \lfloor n+1 \rfloor) = \lfloor n+2 \rfloor$ and $A(\lfloor 0 \rfloor, \lfloor n \rfloor) = \lfloor n+1 \rfloor$.
(*Step*) We take any $m \in N$ and we assume: $A(\lfloor m \rfloor, \lfloor n+1 \rfloor) >_\perp A(\lfloor m \rfloor, \lfloor n \rfloor)$. We have to show: $A(\lfloor m+1 \rfloor, \lfloor n+1 \rfloor) >_\perp A(\lfloor m+1 \rfloor, \lfloor n \rfloor)$. Indeed,
$A(\lfloor m+1 \rfloor, \lfloor n+1 \rfloor) = \{$by definition of $\tau^{na}\} =$
$= A(\lfloor m \rfloor, A(\lfloor m+1 \rfloor, \lfloor n \rfloor)) = \{$by (F1)$\} >_\perp A(\lfloor m+1 \rfloor, \lfloor n \rfloor)$.

(F3) By induction on $n \in N$.
(*Basis*) We have to prove: $\forall m.\, A(\lfloor m+1 \rfloor, \lfloor 0 \rfloor) >_\perp A(\lfloor m \rfloor, \lfloor 0 \rfloor)$. We take any $m \in N$. We have: $A(\lfloor m+1 \rfloor, \lfloor 0 \rfloor) = \{$by definition of $\tau^{na}\} = A(\lfloor m \rfloor, \lfloor 1 \rfloor) = \{$by (F2)$\} >_\perp A(\lfloor m \rfloor, \lfloor 0 \rfloor)$.
(*Step*) We take any $n \in N$. We assume (γ): $\forall m \in N.\, A(\lfloor m+1 \rfloor, \lfloor n \rfloor) >_\perp A(\lfloor m \rfloor, \lfloor n \rfloor)$. We have to show: $\forall m \in N.\, A(\lfloor m+1 \rfloor, \lfloor n+1 \rfloor) >_\perp A(\lfloor m \rfloor, \lfloor n+1 \rfloor)$.

We take any $m \in N$. We have:
$A(\lfloor m{+}1 \rfloor, \lfloor n{+}1 \rfloor) = \{$by definition of $\tau^{na}\} =$
$= A(\lfloor m \rfloor, A(\lfloor m{+}1 \rfloor, \lfloor n \rfloor)) = \{$by induction hypothesis (γ) and (F2)$\} =$
$>_\perp A(\lfloor m \rfloor, A(\lfloor m \rfloor, \lfloor n \rfloor))$.
It remains to show that $A(\lfloor m \rfloor, A(\lfloor m \rfloor, \lfloor n \rfloor)) \geq_\perp A(\lfloor m \rfloor, \lfloor n{+}1 \rfloor)$, where $\geq_\perp$ means, as usual, $>_\perp$ or $=$.

Now, since: (i) by (F1) $A(\lfloor m \rfloor, \lfloor n \rfloor) >_\perp \lfloor n \rfloor$, and (ii) $A(\lfloor m \rfloor, \lfloor n \rfloor) >_\perp \lfloor n \rfloor$ implies $A(\lfloor m \rfloor, \lfloor n \rfloor) \geq_\perp \lfloor n{+}1 \rfloor$, we have (δ): $A(\lfloor m \rfloor, \lfloor n \rfloor) \geq_\perp \lfloor n{+}1 \rfloor$.
We have two cases:

Case $(\delta 1)$: $A(\lfloor m \rfloor, \lfloor n \rfloor) = \lfloor n{+}1 \rfloor$, and Case $(\delta 2)$: $A(\lfloor m \rfloor, \lfloor n \rfloor) >_\perp \lfloor n{+}1 \rfloor$.

In Case $(\delta 1)$ we have: $A(\lfloor m \rfloor, A(\lfloor m \rfloor, \lfloor n \rfloor)) = \{A$ is a function$\} =$
$= A(\lfloor m \rfloor, \lfloor n{+}1 \rfloor)$.
In Case $(\delta 2)$, by (F2) we get: $A(\lfloor m \rfloor, A(\lfloor m \rfloor, \lfloor n \rfloor)) >_\perp A(\lfloor m \rfloor, \lfloor n{+}1 \rfloor)$. Thus, by combining cases $(\delta 1)$ and $(\delta 2)$, we get:
$A(\lfloor m \rfloor, A(\lfloor m \rfloor, \lfloor n \rfloor)) \geq_\perp A(\lfloor m \rfloor, \lfloor n{+}1 \rfloor)$, as desired.
This concludes the proof of (F3). □

We have the following fact which states that the strong termination of the rewriting system of Definition 6.11 on page 184.

FACT 6.13. [**Ackermann Rewriting Rules. (5) Strong Termination**] For all natural numbers m and n, every sequence $t_0, t_1, \ldots$ of terms such that: (i) t_0 is $Ack(m, n)$, and (ii) for all $i \geq 0$, t_{i+1} is derived from t_i by applying in *any* subterm of t_i *any* of the three rewriting rules $R1$–$R3$ of the rewriting system of Definition 6.11 on page 184, is finite, that is, for all $m \geq 0$, for all $n \geq 0$, there exist $r > 0$, $v > 0$ such that $Ack(m, n) \longrightarrow^r v$.

PROOF. The strong termination of the Ackermann rewriting system $R1$–$R3$ is proved by using a bounded lexicographic recursive path order (see Theorem 9.21 on page 44).

Let us consider the set $\{Ack, +, succ, 0\}$ of symbols with the following order $\succcurlyeq$: $Ack \succcurlyeq + \succcurlyeq succ \succcurlyeq 0$. We write 1, instead of $succ(0)$. Let $\succ$ be the bounded lexicographic recursive path order associated with $\succcurlyeq$ (see Definition 9.20 on page 44), and $\succ_{lex}$ be the lexicographic order associated with $\succ$ (see Definition 9.18 on page 44).

For rule $R1$ we have that: $Ack(0, n) \succ n{+}1$ because:
(1.1) $Ack \succcurlyeq +$, (1.2) $Ack(0, n) \succ n$ (by subterm (see Definition 9.15 on page 43)), and (1.3) $Ack(0, n) \succ succ(0)$ because: (1.3.1) $Ack \succcurlyeq succ$, (1.3.2) $Ack(0, n) \succ 0$ (by subterm).

For rule $R2$ we have that: $\langle m{+}1, 0 \rangle \succ_{lex} \langle m, succ(0) \rangle$, because:
(2.1) $m{+}1 \succ m$, (2.2) $Ack(m{+}1, 0) \succ m$ (by subterm), and (2.3) $Ack(m{+}1, 0) \succ succ(0)$ because: (2.3.1) $Ack \succcurlyeq succ$ and (2.3.2) $Ack(m{+}1, 0) \succ 0$ (by subterm).

Finally, for rule $R3$ we have that: $Ack(m{+}1, n{+}1) \succ Ack(m, Ack(m{+}1, n))$, because:
(3.1) $m{+}1 \succ m$, (3.2) $Ack(m{+}1, n{+}1) \succ m$ (by subterm), and (3.3) $Ack(m{+}1, n{+}1) \succ Ack(m{+}1, n)$ because: $\langle m{+}1, n{+}1 \rangle \succ_{lex} \langle m{+}1, n \rangle$ and this holds because $n{+}1 \succ n$ (by subterm).

An alternative proof of Fact 6.13 on the facing page is based on a bounded lexicographic recursive path order with Condition (bl-rpo 2.2*), instead of Condition (bl-rpo 2.2) (see Definition 9.20 on page 44). We take the semantic function $[\![\_]\!]$ of Condition (bl-rpo 2.2*) defined as follows: for all $m, n \geq 0$,

$[\![Ack(m,n)]\!] = v$ iff there exists a natural number v such that $Ack(m,n) \rightarrow^{na} v$

and, indeed, this is the case because of: (i) Fact 4.1 on page 173, (ii) Theorem 5.4 on page 176, and (iii) Fact 6.9 on page 183.

Let us consider the difficult part of this alternative proof. It refers to the case of rule $R3$. The easy cases of the rules $R1$ and $R2$ are left to the reader.

For rule $R3$ we have that: $Ack(m{+}1, n{+}1) \succ Ack(m, Ack(m{+}1, n))$, because: (3.1) $m{+}1 \succ m$, (3.2) $Ack(m{+}1, n{+}1) \succ m$ (by subterm), and (3.3) $Ack(m{+}1, n{+}1) \succ Ack(m{+}1, n)$ because:

$[\![Ack(m{+}1, n{+}1)]\!] >_\perp [\![Ack(m{+}1, n)]\!]$

(by Fact 6.12 on page 185 and the fact that $>_\perp$ is well-founded on $N_\perp^{\geq 0} \times N_\perp^{\geq 0}$, where $N_\perp^{\geq 0}$ is the flat cpo of the natural numbers). □

Having shown the strong termination of the Ackermann rewriting rules of Definition 6.11 on page 184, we have that the rewriting relation $\longrightarrow$ is confluent. Indeed, since there are no critical pairs in those rules, confluence follows from Theorem 9.28 on page 46 and Theorem 9.30 on page 48.

From strong termination and confluence, by Theorem 9.26 on page 46, it follows that the Ackermann rewriting rules associate a unique normal form to every term $Ack(t_1, t_2)$, where t_1 and t_2 are terms of the form (see Definition 6.11):

$t ::= 0 \mid succ(t) \mid t + t \mid Ack(t, t)$.

Now we prove a fact which is analogous to Fact 6.1 on page 176. It will be useful in the following Example 6.15 where we will prove the equivalence of two function declarations under the call-by-name regime.

FACT 6.14. **[Call-by-name Equivalence of Two Functionals]** Let us consider a cpo $D_\perp$ with bottom and the functions $h \in [D_\perp \rightarrow D_\perp]$ and $k \in [D_\perp \rightarrow D_\perp]$, and the predicate $p \in [D_\perp \rightarrow T_\perp]$, where $T_\perp$ is the lifted cpo of the discrete cpo $T = \{true, false\}$. Let us assume that the function h is strict, that is, $h(\perp) = \perp \in D_\perp$. We have that the minimal fixpoint of the two functionals

$\tau_1\ \delta_1\ (x, y) = \ p(x) \rightarrow y \mid h(\delta_1(k(x), y))$
$\tau_2\ \delta_2\ (x, y) = \ p(x) \rightarrow y \mid \delta_2(k(x), h(y))$

where $\delta_1, \delta_2 \in [D_\perp \times D_\perp \rightarrow D_\perp]$, are equal. Recall that $b \rightarrow d_1 \,|\, d_2$ is a continuous function in $[T_\perp \times D_\perp \times D_\perp \rightarrow D_\perp]$ and it stands for *let* $t \Leftarrow b \,.\, cond(t, d_1, d_2)$, where *cond* is a continuous function in $[T \times D_\perp \times D_\perp \rightarrow D_\perp]$.

PROOF. The proof is similar to that of Fact 6.1 on page 176 and can be done by Scott induction by considering the inclusive predicate

$P(f, g) =_{def} \forall x, y.\ f(x, y) = g(x, y) \wedge g(x, h(y)) = h(g(x, y))$. □

EXAMPLE 6.15. **[Call-by-name Equivalence of Sum Declarations]** Let N denote the discrete cpo of the integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$ and $N_\perp$ denote the

flat cpo of the integers with the bottom element $\bot$. Let us consider the following two declarations:

$$sum1(x,y) = \textbf{if } x \textbf{ then } y \textbf{ else } sum1(x{-}1,y)+1$$
$$sum2(x,y) = \textbf{if } x \textbf{ then } y \textbf{ else } sum2(x{-}1,y{+}1)$$

In the call-by-name semantics those declarations define two functionals, call them τ_1 and τ_2, respectively. They satisfy the following equations (recall that $Cond \in [N_\bot \times N_\bot \times N_\bot \to N_\bot]$):

$$\tau_1\ \delta_1\ (x,y) = Cond(x,y,\delta_1(x -_\bot \lfloor 1 \rfloor, y) +_\bot \lfloor 1 \rfloor) \quad \text{where } \delta_1 \in [N_\bot \times N_\bot \to N_\bot]$$
$$\tau_2\ \delta_2\ (x,y) = Cond(x,y,\delta_2(x -_\bot \lfloor 1 \rfloor, y +_\bot \lfloor 1 \rfloor)) \quad \text{where } \delta_2 \in [N_\bot \times N_\bot \to N_\bot]$$

Now, if we indicate by φ_1 and φ_2 the minimal fixpoints of τ_1 and τ_2, respectively, and we use the $\_ \to \_ \mid \_$ construct, instead of $Cond$, we get (recall that $\_ \to \_ \mid \_ \in [T_\bot \times N_\bot \times N_\bot \to N_\bot]$):

$$\varphi_1(x,y) = x =_\bot \lfloor 0 \rfloor \to y \mid \varphi_1(x -_\bot \lfloor 1 \rfloor, y) +_\bot \lfloor 1 \rfloor$$
$$\varphi_2(x,y) = x =_\bot \lfloor 0 \rfloor \to y \mid \varphi_2(x -_\bot \lfloor 1 \rfloor, y +_\bot \lfloor 1 \rfloor)$$

Then, we use Fact 6.14 on the preceding page, where we take: (i) the cpo $D_\bot$ to be $N_\bot$, (ii) the function $h \in [N_\bot \to N_\bot]$ to be the strict function $\lambda x.\, x +_\bot \lfloor 1 \rfloor$, (iii) the function $k \in [N_\bot \to N_\bot]$ to be $\lambda x.x -_\bot \lfloor 1 \rfloor$, and (iv) the predicate $p \in [N_\bot \to T_\bot]$ to be $\lambda x.x =_\bot \lfloor 0 \rfloor$, and we conclude that $fix(\tau_1) = fix(\tau_2)$, that is, $\forall x, y \in N_\bot.\, \varphi_1(x,y) = \varphi_2(x,y)$. □

In Example 6.16 and Example 6.18 on the next page we will see in action the Structural Induction rule. Before presenting those examples we need introduce the discrete cpo, called $List(\Sigma)$, of the lists of elements taken from a given set Σ.

The discrete cpo $List(\Sigma)$ is the smallest set, also denoted $List(\Sigma)$, which has:
(i) the empty list $[\,]$, and
(ii) the list $a : \ell$, for any $a \in \Sigma$, for any list $\ell \in List(\Sigma)$, where ':' denotes the familiar infix constructor *cons* for lists.

For instance, given the list $[c,a,a]$, we have that $b : [c,a,a]$ is the list $[b,c,a,a]$. As usual, for any $a \in \Sigma$, $a : [\,]$ is also written as $[a]$. The partial order $\sqsubseteq$ on the cpo $List(\Sigma)$ is defined as follows:

for all $\ell \in List(\Sigma)$, $\ell \sqsubseteq \ell$.

Thus, the elements of the set $List(\Sigma)$ are those generated by the following productions from the nonterminal symbol L, for any $a \in \Sigma$:

$$L \ \to\ [\,] \ \mid\ a : L \qquad \text{(Lists)}$$

We will also use the *head* function hd: $List(\Sigma) - \{[\,]\} \to \Sigma$ and the *tail* function tl: $List(\Sigma) - \{[\,]\} \to List(\Sigma)$, defined as usual. For instance, $hd([b,a,a]) = b$ and $tl([b,a,a]) = [a,a]$.

EXAMPLE 6.16. [**Associativity of the Append Function**] Let us consider the following functional $\tau_{app} \in \big[[List(\Sigma) \times List(\Sigma) \to List(\Sigma)] \to [List(\Sigma) \times List(\Sigma) \to List(\Sigma)]\big]$:

$\tau_{app} =_{def} \lambda\delta.\, \lambda(x,y).\, cond(x{=}[\,], y, hd(x){:}\delta(tl(x), y))$ (functional τ_{app} for *append*)

where *cond*: $\{true, false\} \times List(\Sigma) \times List(\Sigma) \to List(\Sigma)$. Let the *append* function *app*: $List(\Sigma) \times List(\Sigma) \to List(\Sigma)$ be defined as the minimal fixpoint $fix(\tau_{app})$. Thus, we have the following equation:

$app(x,y) = cond(x{=}[\,], y, hd(x){:}app(tl(x), y))$. (*append* function) (†)

For instance, $app([a,c,b],[c,b]) = [a,c,b,c,b]$.

Now we prove that *app* enjoys the associativity property, that is, we have that, for all $x, y, z \in List(\Sigma)$,

$app(app(x,y),z) = app(x, app(y,z))$ (associativity of *append*)

The proof is by structural induction on the list x as follows.

(*Basis*) $x{=}[\,]$. We have to show that $app(app([\,], y), z) = app([\,], app(y,z))$.

We have that:

(i) l.h.s: $app(app([\,], y), z) =$

$= app(cond([\,]{=}[\,], y, hd(x){:}app(tl(x), y)), z) = app(y, z)$.

(ii) r.h.s.: $app([\,], app(y,z)) =$

$= cond([\,]{=}[\,], app(y,z), hd(x){:}app(tl(x), y)), z){=} app(y,z)$.

(*Step*) For all $\ell, y, z \in List(\Sigma)$, for all $a \in \Sigma$,

we assume that $app(app(\ell, y), z) = app(\ell, app(y,z))$ and we have to show that $app(app(a{:}\ell, y), z) = app(a{:}\ell, app(y,z))$.

We have that:

(i) l.h.s.: $app(app(a{:}\ell, y), z) =$

$= app(cond(a{:}\ell{=}[\,], y, a{:}app(\ell, y)), z) =$

$= app(a{:}app(\ell, y), z) = cond(a{:}app(\ell, y){=}[\,], z, a{:}app(app(\ell, y), z)) =$

$= a{:}app(app(\ell, y), z) =$ {by induction hypothesis} $=$

$= a{:}app(\ell, app(y,z))$.

(ii) r.h.s.: $app(a{:}\ell, app(y,z)) =$

$= cond(a{:}\ell{=}[\,], app(y,z), a{:}app(\ell, app(y,z))) =$

$= a{:}app(\ell, app(y,z))$. □

Exercise 6.17. Show by structural induction on x that for all $x \in List(\Sigma)$, $app(x, [\,]) = x$. □

Example 6.18. **[Properties of the Reverse Function (1)]** Let us introduce the *reverse* function *rev*: $List(\Sigma) \to List(\Sigma)$. First, we consider the the functional $\tau_r \in \big[[List(\Sigma) \times List(\Sigma) \to List(\Sigma)] \to [List(\Sigma) \times List(\Sigma) \to List(\Sigma)]\big]$ defined as follows:

$\tau_r =_{def} \lambda\delta.\, \lambda(x,y).\, cond(x{=}[\,], y, \delta(tl(x), hd(x){:}y))$ (functional τ_r for r)

Then we define:

$rev(x) = r(x, [\,])$ (*reverse* function)

where the function r is the minimal fixpoint $fix(\tau_r)$, that is,

$r(x, y) = cond(x{=}[\,], y, r(tl(x), hd(x){:}y))$ (r function)

We have the following properties: for all $a \in \Sigma$, for all $\ell, m \in List(\Sigma)$,

$rev([\,]) = [\,]$ (both sides are equal to $r([\,], [\,])$) (†1)

$rev(a{:}\ell) = r(\ell, [a])$ (both sides are equal to $r(a{:}\ell, [\,])$) (†2)

$r(\ell, m) = app(rev(\ell), m)$ (†3)

where *app* denotes the *append* function on lists introduced in Example 6.16 on page 188. The proofs of (†1) and (†2) are left to the reader. Here is the proof of (†3) by structural induction on ℓ.

(*Basis*) For $\ell{=}[\,]$ we have:

l.h.s.: $r([\,], m) = \{$by definition of $r\} = m$.

r.h.s.: $app(rev([\,]), m) = \{$by (†1)$\} =$

$= app([\,], m) = \{$by (†) of Example 6.16 on page 188$\} = m$.

(*Step*) For any $a \in \Sigma$, any $\ell, m \in List(\Sigma)$, assume $r(\ell, m) = app(rev(\ell), m)$ and show that $r(a{:}\ell, m) = app(rev(a{:}\ell), m)$.

l.h.s.: $r(a{:}\ell, m) = \{$by definition of $r\} = r(\ell, a{:}m)$.

r.h.s.: $app(rev(a{:}\ell), m) = \{$by (†2)$\} =$

$= app(r(\ell, [a]), m) = \{$by induction hypothesis$\} =$

$= app(app(rev(\ell), [a]), m) =$

$= \{$by associativity of *app* (see Example 6.16 on page 188)$\} =$

$= app(rev(\ell), app([a], m)) =$

$= \{$by (†) of Example 6.16 on page 188$\} =$

$= app(rev(\ell), a{:}m) = \{$by induction hypothesis$\} =$

$= r(\ell, a{:}m)$.

This concludes the proof of (†3). As a particular instance of (†3) for $m{=}[a]$, we get:

$r(\ell, [a]) = app(rev(\ell), [a])$

that is, by (†2),

$rev(a{:}\ell) = app(rev(\ell), [a])$. (†4) □

EXAMPLE 6.19. **[Properties of the Reverse and Append Functions (2)]** Let us consider the flat cpo $List(\Sigma)_\perp$ which is the set $List(\Sigma) \cup \{\perp\}$ with the partial order $\sqsubseteq$ defined as follows:

for all $x, y \in List(\Sigma)_\perp$, $x \sqsubseteq y$ iff $x{=}\perp$ or $x{=}y$.

By using the McCarthy Induction rule, we will prove that for all $u, v \in List(\Sigma)_\perp$ such that $u{\neq}\perp$ and $v{\neq}\perp$:

$rev_\perp(app_\perp(u, v)) = app_\perp(rev_\perp(v), rev_\perp(u))$

where: (i) the function $app_\perp$: $List(\Sigma)_\perp \times List(\Sigma)_\perp \to List(\Sigma)_\perp$ is the strict extension of the function *app* defined in Example 6.16 on page 188, and (ii) the function $rev_\perp$: $List(\Sigma)_\perp \to List(\Sigma)_\perp$ is the strict extension of the function *rev* defined in Example 6.18 on the previous page.

Thus, we have that: for all $u, v \in List(\Sigma)_\perp$,

$$app_\bot(u,v) = cond\big(u=\bot \vee v=\bot,\ \bot,\ let\ x \Leftarrow u, y \Leftarrow v \centerdot \lfloor app(x,y)\rfloor\big) \qquad (app_\bot)$$

$$rev_\bot(u) = cond\big(u=\bot,\ \bot,\ let\ x \Leftarrow u \centerdot \lfloor rev(x)\rfloor\big) \qquad (rev_\bot)$$

Recall that *cond* is a continuous function in $[\{true, false\} \times List(\Sigma)_\bot \times List(\Sigma)_\bot \to List(\Sigma)_\bot]$.

Let us consider the continuous functional $\tau \in \big[[List(\Sigma)_\bot \times List(\Sigma)_\bot \to List(\Sigma)_\bot] \to [List(\Sigma)_\bot \times List(\Sigma)_\bot \to List(\Sigma)_\bot]\big]$ defined as follows:

$$\begin{array}{l}\tau =_{def} \lambda\delta.\,\lambda(u,v).\ cond\big(u=\bot \vee v=\bot,\ \bot,\\ \qquad\qquad let\ x \Leftarrow u, y \Leftarrow v \centerdot \\ \qquad\qquad \big\lfloor cond(x=[\,],\ rev(y),\\ \qquad\qquad\ let\ d \Leftarrow \delta(\lfloor tl(x)\rfloor, \lfloor y\rfloor) \centerdot app(d,\ [hd(x)])\big\rfloor\big)\end{array} \qquad (\text{functional } \tau)$$

where the outside *cond* is in $[\{true, false\} \times List(\Sigma)_\bot \times List(\Sigma)_\bot \to List(\Sigma)_\bot]$ and the inner *cond* is in $[\{true, false\} \times List(\Sigma) \times List(\Sigma) \to List(\Sigma)]$.

In order to use the McCarthy Induction rule, we have to show that:
for all $u, v \in List(\Sigma)_\bot$,

(i) $rev_\bot(app_\bot(u,v)) = \tau\big(\lambda(r,s).\, rev_\bot(app_\bot(r,s))\big)\,(u,v)$,
that is, $\lambda(r,s).\, rev_\bot(app_\bot(r,s))$ is a fixpoint of the functional τ,

(ii) $app_\bot(rev_\bot(v), rev_\bot(u)) = \tau\big(\lambda(r,s).\, app_\bot(rev_\bot(s), rev_\bot(r))\big)\,(u,v)$,
that is, $\lambda(r,s).\, app_\bot(rev_\bot(s), rev_\bot(r))$ is a fixpoint of the functional τ, and

(iii) for all $u, v \in List(\Sigma)_\bot$, if $u \neq \bot$ and $v \neq \bot$, then $(fix(\tau))(u,v) \neq \bot$.

Proof of Point (i). By cases. If $u=\bot \vee v=\bot$, Point (i) is obvious.

If $u = \lfloor[\,]\rfloor$ and $v = \lfloor y\rfloor$, for some $y \in List(\Sigma)$, we have the following.
l.h.s. of (i): $rev_\bot(app_\bot(\lfloor[\,]\rfloor, \lfloor y\rfloor)) = \{$by $(rev_\bot)$ and $(app_\bot)\} =$
$= \lfloor rev(app([\,], y))\rfloor = \{$by (†) of Example 6.16 on page 188$\} = \lfloor rev(y)\rfloor$.
r.h.s. of (i): $\tau\big(\lambda(r,s).\, rev_\bot(app_\bot(r,s))\big)\,(\lfloor[\,]\rfloor, \lfloor y\rfloor) = \{$by the definition of $\tau\} =$
$= \lfloor rev(y)\rfloor$.

If $u = \lfloor a:\ell\rfloor$ and $v = \lfloor y\rfloor$, for some $a \in \Sigma$, for some $\ell, y \in List(\Sigma)$, we have the following.
l.h.s. of (i): $rev_\bot(app_\bot(\lfloor a:\ell\rfloor, \lfloor y\rfloor)) = \{$by $(rev_\bot)$ and $(app_\bot)\} =$
$= \lfloor rev(app(a:\ell, y))\rfloor = \{$by (†) of Example 6.16 on page 188$\} =$
$= \lfloor rev(a:app(\ell, y))\rfloor = \{$by (†4) of Example 6.18 on page 189$\} =$
$= \lfloor app(rev(app(\ell, y)), [a])\rfloor$.
r.h.s. of (i): $\tau\big(\lambda(r,s).\, rev_\bot(app_\bot(r,s))\big)\,(\lfloor a:\ell\rfloor, \lfloor y\rfloor) = \{$by the definition of $\tau\} =$
$= \lfloor let\ d \Leftarrow rev_\bot(app_\bot(\lfloor\ell\rfloor, \lfloor y\rfloor)) \centerdot app(d, [a])\rfloor =$
$= \lfloor let\ d \Leftarrow \lfloor rev(app(\ell, y))\rfloor \centerdot app(d, [a])\rfloor =$
$= \lfloor app(rev(app(\ell, y)), [a])\rfloor$.

Proof of Point (ii). By cases. If $u=\bot \vee v=\bot$, Point (ii) is obvious.

If $u = \lfloor[\,]\rfloor$ and $v = \lfloor y\rfloor$ for some $y \in List(\Sigma)$, we have the following.
l.h.s. of (ii): $app_\bot(rev_\bot(\lfloor y\rfloor), rev(\lfloor[\,]\rfloor)) = \{$by $(rev_\bot)$ and $(app_\bot)\} =$

$= \lfloor app(rev(y), rev([\,])) \rfloor$ = {by (†1) of Example 6.18 on page 189} =
$= \lfloor app(rev(y), [\,]) \rfloor$ = {by Exercise 6.17 on page 189} =
$= \lfloor rev(y) \rfloor$.

r.h.s. of (ii): $\tau\big(\lambda(r,s).\, app_\perp(rev_\perp(s), rev_\perp(r))\big)\,(\lfloor [\,] \rfloor, \lfloor y \rfloor)$ =
$=$ {by the definition of τ} $= \lfloor rev(y) \rfloor$.

If $u = \lfloor a : \ell \rfloor$ and $v = \lfloor y \rfloor$, for some $a \in \Sigma$, for some $\ell, y \in List(\Sigma)$, we have the following.

l.h.s. of (ii): $app_\perp(rev_\perp(\lfloor y \rfloor), rev_\perp(\lfloor a{:}\ell \rfloor))$ = {by $(rev_\perp)$ and $(app_\perp)$} =
$= \lfloor app(rev(y), rev(a{:}\ell)) \rfloor$ = {by (†4) of Example 6.18 on page 189} =
$= \lfloor app(rev(y), app(rev(\ell), [a])) \rfloor$.

r.h.s. of (ii): $\tau\big(\lambda(r,s).\, app_\perp(rev_\perp(s), rev_\perp(r))\big)\,(\lfloor a{:}\ell \rfloor, \lfloor y \rfloor)$ =
$=$ {by the definition of τ} =
$= \lfloor let\, d \Leftarrow app_\perp(rev_\perp(\lfloor y \rfloor), rev_\perp(\lfloor \ell \rfloor)) \centerdot app(d, [a]) \rfloor$ =
$= \lfloor let\, d \Leftarrow app_\perp(\lfloor rev(y) \rfloor, \lfloor rev(\ell) \rfloor) \centerdot app(d, [a]) \rfloor$ =
$= \lfloor app(app(rev(y), rev(\ell)), [a]) \rfloor$ =
$= \lfloor app(app(rev(y), rev(\ell)), [a]) \rfloor$ =
$=$ {by associativity of *app* (see Example 6.16 on page 188)} =
$= \lfloor app(rev(y), app(rev(\ell), [a])) \rfloor$.

Proof of Point (iii). For all $u, v \in List(\Sigma)_\perp$, if $u \neq \perp$ and $v \neq \perp$, that is, if $u = \lfloor x \rfloor$ and $v = \lfloor y \rfloor$ for some $x, y \in List(\Sigma)$, by the definition of τ we have that:
$(fix(\tau))(u,v) = \lfloor cond\big(x{=}[\,],\ rev(y),\ let\, d \Leftarrow fix(\tau)(\lfloor tl(x) \rfloor, \lfloor y \rfloor) \centerdot app(d, [hd(x)])\big) \rfloor$.
Thus, $(fix(\tau))(u,v) \neq \perp$. □

7. Computation of Fixpoints in the Language REC via Rewritings

In this section we will present some operational semantics for the language REC and we will investigate their relationship with the call-by-value and call-by-name denotational semantics (defined on pages 168 and 175, respectively).

In order to present the operational semantics we will follow the approach based on term rewriting [**9**, Chapter 5], rather than the one based on deduction rules which we have followed for the call-by-value semantics in Section 2 on page 166 and for the call-by-name semantics in Section 4 on page 173.

Let us first recall the definition of the set **Term** of *terms* of the language REC which we have presented on page 165. A term $t \in \mathbf{Term}$ is defined as follows:

$$t \;::=\; n \mid x \mid t_1\, \mathbf{op}\, t_2 \mid \mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \mid f_i(t_1, \ldots t_{a_i})$$

where t_0, t_1, t_2, t_{a_i} are terms in **Term**, n is an integer in $N = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$, x is a variable in **Var**, **op** is an operator in the set $\{+, -, \times, \}$, and f_i is a function variable in **Fvar**. Let us also consider a set of declarations of the form:

$$\begin{cases} f_1(x_1, \ldots, x_{a_1}) & = & d_1 \\ & \ldots & \\ f_k(x_1, \ldots, x_{a_k}) & = & d_k \end{cases}$$

Step (A). (*Unfolding*) We replace, simultaneously and in parallel, the function calls singled out in the term t_i by the rule R, by the corresponding instances of the right hand sides of their declarations.

For instance, we have that: $f_h(t_1,\ldots,t_{a_h}) \rightarrow_R d_h[t_1/x_1,\ldots,t_{a_h}/x_{a_h}]$, for $h=1,\ldots,k$.

Step (B). Perform the following Steps (B.1) and (B.2) as long as possible.

(B.1) (*Evaluation of arithmetic operators*) We evaluate *every* arithmetical operator $\mathbf{op} \in \{+,-,\times\}$ whose operands have both been evaluated to an integer.

For instance, we have that: $3\times(5-1) \rightarrow_R 3\times 4 \rightarrow_R 12$.

(B.2) (*Simplification of conditionals*) We simplify every **if-then-else** construct whose condition has been evaluated to an integer, that is, for all terms t_1 and t_2, **if** 0 **then** t_1 **else** $t_2 \rightarrow_R t_1$, and **if** n **then** t_1 **else** $t_2 \rightarrow_R t_2$, if $n \neq 0$.

For instance, we have that: **if** 2 **then** 1 **else** $0 \rightarrow_R 1$.

FIGURE 1. Operational Semantics of the language REC via term rewriting, according to the computation rule R.

where, for $i = 1,\ldots,k$, d_i is a term such that: (i) every variable occurring in d_i belongs to the set $\{x_1,\ldots,x_{a_i}\}$, and (ii) every function variable in d_i belongs to $\{f_1,\ldots,f_k\}$.

Let a *computation rule* R be a function that given a term t, singles out some occurrences of the function variables in t.

The *evaluation of a term* t according a given computation rule R, is defined to be the last term, if any, of a (possibly infinite) maximal sequence $\sigma = \langle t_0, t_1, \ldots, t_i, \ldots\rangle$ of terms, called the *evaluation sequence of the term* t, such that: (1) t_0 is t, and (2) for $i \geq 0$, the term t_{i+1} is derived by the term t_i, and we write $t_i \rightarrow_R t_{i+1}$, by performing the Steps (A) and (B) indicated in Figure 1. In that figure we present the operational semantics of the language REC via term rewriting according to a given computation rule R.

Note that the evaluation sequence is a *maximal* sequence in the sense that if its last term is the term t_n then there is no term t such that $t_n \rightarrow_R t$.

If $t_i \rightarrow_R t_{i+1}$ we say that the term t_i *is rewritten into* the term t_{i+1} by using the rule R.

When the computation rule R is understood from the context we will write $\rightarrow$, instead of $\rightarrow_R$.

We have that: (i) if the evaluation sequence σ is finite then only the last term of σ is an integer, and (ii) if the evaluation sequence σ is infinite then no term of σ is an integer.

These two properties of σ are in accordance with the fact that both the call-by-name and the call-by-value denotational semantics of a closed term in REC is a value in the lifted cpo $N_\perp$ of the integers.

If the evaluation sequence σ of the term t that we may construct using $\rightarrow_R$ with the call-by-value (or call-by-name) computation rule R (see below on page 194), is an infinite sequence, then the call-by-value (or call-by-name, respectively) denotational semantics of a term t is $\perp$.

If the evaluation of a term t according to a computation rule R is n, we will write $t \rightarrow_R^* n$.

REMARK 7.1. (i) The evaluation of any arithmetical operator requires first the evaluation of *all* its arguments. Thus, for instance, we *cannot* replace $0 \times t$ by 0, unless t has been first evaluated to an integer.

(ii) The simplification of any **if-then-else** construct requires first the evaluation of its condition to an integer. Thus, in particular, the expression **if** t_0 **then** t **else** t (in which the left arm is equal to the right arm), should not be replaced by t, unless t_0 has first been evaluated to an integer.

Note that, as usual for the language REC, in the expression **if** t_0 **then** t_1 **else** t_2 we have that if $t_0 = 0$ then t_0 stands for *true*, while if $t_0 \neq 0$ then t_0 stands for *false*. □

In the rewriting steps for constructing the sequence σ of terms starting from a given term t using a computation rule R, we have that:

for all terms t_1, t_2,
if $t_1 \rightarrow_R t_2$ then for every context $C[\_]$, if in $C[t_1]$ the computation rule R requires to first evaluate t_1, then $C[t_1] \rightarrow_R C[t_2]$.

In this sense we say that a context-rule holds for our operational semantics.

Let us consider the following computation rules:

(i) *leftmost-innermost rule*, which singles out the leftmost occurrence of a function variable which has *all* its arguments without function variables (this rule is a particular instance of the *call-by-value* rule as described by the deduction rules on page 167),

(ii) *parallel-innermost rule*, which singles out *all* occurrences of function variables, each of which has *all* its arguments without occurrences of function variables,

(iii) *leftmost rule*, which singles out the leftmost occurrence of a function variable (this rule is a particular instance of the *call-by-name* rule as described by the deduction rules on page 173),

(iv) *parallel-outermost rule*, which singles out *all* occurrences of function variables which do not occur in arguments of other function variables,

(v) *free-argument rule*, which singles out *all* occurrences of function variables, each of which has *at least one* argument without occurrences of function variables, and

(vi) *full-substitution rule*, which singles out *all* occurrences of function variables.

In the following table we indicate the occurrences of the function variables which are singled out by the various computation rules in the following term:

$f_1(f_2(0,1), f_3(2,3)) + f_4(f_5(4,5), 6)$.

Computation rule R	Occurrences which are singled out in the term $f_1(f_2(0,1), f_3(2,3)) + f_4(f_5(4,5),6)$
leftmost-innermost rule: *parallel-innermost rule*:	f_2 $f_2,\ f_3,\ f_5$
leftmost rule: *parallel-outermost rule*: *free-argument rule*: *full-substitution rule*:	f_1 $f_1,\ f_4$ $f_2,\ f_3,\ f_4,\ f_5$ $f_1,\ f_2,\ f_3,\ f_4,\ f_5$

Now we give an example of evaluation of a term.

EXAMPLE 7.2. [**McCarthy 91 Function. Leftmost Computation Rule**] Let us consider the McCarthy 91 function which we have introduced in Example 6.3 on page 178. We recall its declaration which is as follows: for all (negative, or null, or positive) integers $n \in N$,

$f(n) = \textbf{if } n{>}100 \textbf{ then } n{-}10 \textbf{ else } f(f(n{+}11))$ (McCarthy 91 function)

Let us consider the *leftmost computation* rule. Then evaluation sequence of the term $f(-3)$ is the following sequence of terms, where: (i) we have written $f^k(n)$, instead of $f(\ldots(f(n))\ldots)$ with k occurrences of f, and (ii) for reasons of brevity we have omitted the rewriting steps relative to the evaluation of arithmetic operators and the simplification of conditionals.

$$
\begin{aligned}
& f(-3) \to f^2(8) \to f^3(19) \to \ldots \to f^{10}(96) \to f^{11}(107) \to f^{10}(97) \\
& \quad \to f^{11}(108) \to f^{10}(98) \\
& \quad \to \underline{f^{11}(109) \to f^{10}(99)} && (\dagger 11) \\
& \quad \to f^{11}(110) \to f^{10}(100) \to f^{11}(111) \\
& \quad \to f^{10}(101) \to f^9(91) && (\dagger 101) \\
& \quad \to f^{10}(102) \to f^9(92) && (\dagger 102) \\
& \quad \ldots \{\text{by comparing the rewritings } (\dagger 101) \text{ and } (\dagger 102)\} \\
& \quad \to \underline{f^{10}(109) \to f^9(99)} && (\dagger 10) \\
& \quad \ldots \{\text{by comparing the rewritings } (\dagger 11) \text{ and } (\dagger 10)\} \\
& \quad \to \underline{f^2(109) \to f(99)} \\
& \quad \to f^2(110) \to f(100) \to f^2(111) \to f(101) \to 91.
\end{aligned}
$$

One can show that for all $n \in N = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$,

if $n{>}100$
then $f(n) \to_R^* n{-}10$
else $f(n) \to_R^* 91$

where R is the leftmost computation rule.

This fact follows from Theorem 7.4 on the next page, which tells us that the leftmost computation rule agrees with the value of the minimal fixpoint

$\lambda n.\, cond(n{>}100, \lfloor n{-}10 \rfloor, \lfloor 91 \rfloor)$

of the functional

$\tau =_{def} \lambda\delta.\lambda n.\ cond\big(n>100,\ \lfloor n-10\rfloor,\ \delta(\delta(n+11))\big)$

which expresses the call-by-name semantics of the above declaration of McCarthy 91 function. □

Now we state without proof the following theorems [**9**, Chapter 5] which refer to a given set $Decl = \{f_i(x_1, \ldots, x_{a_i}) = d_i \mid 1 \leq i \leq k\}$ of declarations.

Let $\varphi \in [N_\perp^{a_1} \to N_\perp] \times \ldots \times [N_\perp^{a_k} \to N_\perp]$ be the minimal fixpoint of the functional associated with the set *Decl* of declarations in the call-by-name semantics. Let ρ be an environment, that is, a function from the set **Var** of variables to the cpo $N_\perp$.

Given the set *Decl* of declarations and a term t, by $[\![t]\!]^{na}\varphi\rho$ we denote the element in $N_\perp$ which is the call-by-name semantics of the term t.

THEOREM 7.3. [**Cadiou Theorem**] For any term t and any computation rule R, if $t \to_R^* n$ then $[\![t]\!]^{na}\varphi\rho = \lfloor n\rfloor \in N_\perp$.

Note that the converse of Cadiou Theorem *does not hold*. Indeed, given a term t and a computation rule R, it may be the case that there exists an integer n such that $[\![t]\!]^{na}\varphi\rho = \lfloor n\rfloor$ holds and $t \to_R^* n$ does not hold, that is, the evaluation sequence of t is infinite.

THEOREM 7.4. [**Computation Rules for Call-by-name in REC**] If the computation rule R is either the *parallel-outermost* rule or the *full-substitution* rule or the *leftmost* rule, then the evaluation of a closed term t agrees with the value $[\![t]\!]^{na}\varphi\rho$, in the sense that for all integer n,

$t \to_R^* n$ iff $[\![t]\!]^{na}\varphi\rho = \lfloor n\rfloor \in N_\perp$.

Theorem 7.4 holds also for the *free-argument* computation rule, provided that in the set *Decl* there is no function variable symbol f_i such that the function environment φ associates a constant function with f_i.

Since Theorem 7.4 holds for the parallel-outermost rule, it generalizes Theorem 5.4 on page 176.

The following example shows that Theorem 7.4 *cannot* be extended to the case where the computation rule is the *leftmost-innermost* rule or the *parallel-innermost* rule.

EXAMPLE 7.5. [**Morris Function**] Let us consider the following declaration of the function f defined as follows [**9**, page 389]: for all (negative, or null, or positive) integers $m, n \in N$:

$f(m,n) =$ **if** $m=0$ **then** 1 **else** $f(m-1, f(m,n))$

The minimal fixpoint of the functional associated with this declaration for the call-by-name semantics is the function

$\delta =_{def} \lambda m.\lambda n.\ cond(m \geq 0,\ \lfloor 1\rfloor,\ \perp) \in N_\perp \times N_\perp \to N_\perp$

Now, $[\![f(1,0)]\!]^{na}\varphi\rho = \lfloor 1\rfloor$ (because the function environment φ associates δ with the function variable f), while according to the *leftmost-innermost* rule (and the *parallel-innermost* rule), we get the following infinite sequence:

$f(1,0) \to^* f(0, f(1,0)) \to^* f(0, f(0, f(1,0))) \to^* \ldots$ □

Now we want to present a variant of Theorem 7.4.

First, note that one may define the call-by-name denotational semantics of the arithmetic operators of the language REC by using *monotonic* functions, rather than *strict* functions (as we did on page 175 where we used the function $op_\perp$). Indeed, any functional $\tau \in [[N_\perp^{a_i} \to N_\perp] \to [N_\perp^{a_i} \to N_\perp]]$ which is constructed by using monotonic functions, the function *Cond* (or, equivalently, the function *cond*), and the function variables, is continuous and, thus, its minimal fixpoint exists and can be taken to define the denotational semantics of a declaration [**9**, Chapter 5].

Now, let us assume that for the call-by-name denotational semantics of the arithmetic operators $+$, $-$, and $\times$ we use, respectively, the strict functions $+_\perp$, $-_\perp$, and the *monotonic* (not strict) function $\times_m : N_\perp \times N_\perp \to N_\perp$, defined as follows:

$$\begin{array}{lll} \text{for all } n \in N_\perp, & n \times_m \lfloor 0 \rfloor \;=\; \lfloor 0 \rfloor \times_m n \;=\; \lfloor 0 \rfloor \\ \text{for all } n \in N_\perp \text{ different from } \lfloor 0 \rfloor, & \perp \times_m n \;=\; n \times_m \perp \;=\; \perp \end{array}$$

Let us also assume the following variant (B.1*) of Step (B.1) (see page 193) for rewriting products.

(B.1*) (*Evaluation of arithmetic operators*) We evaluate *every* arithmetical operator **op** $\in \{+, -\}$ whose operands have both been evaluated to an integer, except that, whenever possible, we use the following rewriting rule for products:

for all terms t, $\quad 0 \times t \to 0 \qquad$ and $\qquad t \times 0 \to 0$

(thus, for products when one operand has been evaluated to 0, we should *not* evaluate the other operand).

In these hypotheses we have that Theorem 7.4 on the preceding page still holds for the parallel-outermost rule and the full-substitution rule, but it *does not hold* for the leftmost computation rule as we now show.

Let us consider the following declaration:

$$f(x,y) = \textbf{if } x \textbf{ then } 0 \textbf{ else } f(x{+}1, f(x,y)) \times f(x{-}1, f(x,y))$$

whose associated functional $\tau \in \big[[N_\perp \times N_\perp \to N_\perp] \to [N_\perp \times N_\perp \to N_\perp]\big]$ in the call-by-name semantics is defined as follows:

$$\tau =_{def} \lambda\delta.\lambda m.\lambda n.\mathit{Cond}\big(m,\ \lfloor 0 \rfloor,\ \delta(m +_\perp \lfloor 1 \rfloor, \delta(m,n)) \times_m \delta(m -_\perp \lfloor 1 \rfloor, \delta(m,n))\big)$$

We have that: $\mathit{fix}(\tau) = \lambda m.\lambda n.\mathit{Cond}(m, \lfloor 0 \rfloor, \lfloor 0 \rfloor)$. Thus, $[\![f(1,0)]\!]^{na}\,\varphi\rho = \lfloor 0 \rfloor$ (because the function environment φ associates $\mathit{fix}(\tau)$ with the function variable f), while starting from the term $f(1,0)$ by using the *leftmost* computation rule, we get an infinite sequence of terms. Indeed, we have that:

$$\underline{f(1,0)} \to^* \underline{f(2,f(1,0))} \times f(0,f(1,0)) \to^* \big(\underline{f(3,f(1,0))} \times f(1,f(1,0))\big) \times f(0,f(1,0)) \to^* \ldots$$

where the first argument of the outermost f grows in an unbounded way (see the underlined terms). An infinite sequence of terms is also obtained starting from $f(1,0)$ if we use either the *leftmost-innermost* or the *parallel-innermost* computation rule.

On the contrary, if we use the *parallel outermost* rule we get:

$\underline{f(1,0)} \rightarrow^* \underline{f(2,f(1,0))}\times\underline{f(0,f(1,0))} \rightarrow^* \big(f(3,f(2,f(1,0)))\times f(1,f(2,f(1,0)))\big)\times 0 \rightarrow 0$

The reader may check that if we use the *full-substitution* rule we also get:

$f(1,0) \rightarrow^* 0$.

The following theorem which is analogous to Theorem 7.4, holds for the *call-by-value* semantics of REC.

THEOREM 7.6. [**Computation Rules for Call-by-value in REC**] If the computation rule R is either the *leftmost-innermost* rule or the *parallel-innermost* rule, then the evaluation of a closed term t agrees with the value $[\![t]\!]^{va}\varphi\rho$, in the sense that for all integer n,

$t \rightarrow_R^* n$ iff $[\![t]\!]^{va}\varphi\rho = \lfloor n \rfloor \in N_\perp$.

Since Theorem 7.6 holds for the parallel-innermost rule, it generalizes Theorem 3.4 on page 172.

Theorem 7.6 *does not hold* if we consider any of the following computation rules: (i) the *parallel-outermost* rule, or (ii) the *full-substitution* rule, or (iii) the *leftmost* rule. Indeed, let us consider the declaration of the following two functions:

$$\begin{cases} f(x) &= f(x+1) \\ g(x) &= 1 \end{cases}$$

By using either the *parallel-outermost* rule, or the *full-substitution* rule, or the *leftmost* rule, we have that: $g(f(0)) \rightarrow 1$, while $[\![g(f(0))]\!]^{va}\varphi\rho = \perp \in N_\perp$.

If we assume that in the set *Decl* of declarations there is no function variable f_i such that the function environment φ associates a constant function with f_i, then Theorem 7.6 *does not hold* for the *free-argument* computation rule [**9**, Chapter 5]. Indeed, for the Morris function (see Example 7.5 on page 196) we have that $[\![f(1,0)]\!]^{va}\varphi\rho = \perp$, while by using the *free-argument* rule we get that: $f(1,0) \rightarrow^* 1$.

CHAPTER 7

Syntax and Semantics of Higher Order Functional Languages

In this chapter we introduce two higher order, typed functional languages, the Eager language and the Lazy language. Actually, as we will see below, since we will define two different denotational semantics of the Lazy language, we will have two Lazy languages, the Lazy1 language and the Lazy2 language. Recall that the notion of a programming language depends on the definition of:

(i) its syntax,
(ii) its operational semantics, and
(iii) its denotational semantics, and

thus, by changing one of those definitions we change the language.

Let us first introduce the following basic sets.

(i) The set *Types* of the *types*, which is defined as follows:

$$\tau \ ::= \ int \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2$$

where $\tau, \tau_1, \tau_2 \in \mathit{Types}$. We have that: (i) *int* is the type of the *integers*, (ii) the product type $\tau_1 \times \tau_2$ is the type of the pairs whose first component is of type τ_1 and whose second component is of type τ_2, and (iii) the function type $\tau_1 \rightarrow \tau_2$ is the type of the functions from type τ_1 to type τ_2.

(ii) The set $N = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ of *integers*. The variables ranging over N are: $n, m, \ldots$

(iii) The set $\{+, -, \times\}$ of *arithmetic operators*. **op** ranges over $\{+, -, \times\}$.

(iv) The set $\mathbf{Var} = \{x, y, f, v, w, \ldots\}$ of *variables* with arity r, with $r \geq 0$.

Note that, contrary to the REC language, we keep in the single set **Var** the variables of all types (and, thus, also function variables). Note that the function variables may have arity different from 0 and they may be applied to arguments.

Substitutions are denoted by using the square brackets notation, that is, $t[t_1/x]$ denotes the term t where each free occurrence of the variable x has been replaced by the term t_1 (see also the definition of the β-rule on page 20).

1. Syntax of the Eager Language and the Lazy Language

In this section we introduce the syntax of the Eager language and of the Lazy language.

1.1. Syntax of the Eager Language.

The set **Term** of the terms in the Eager language is defined as follows.

$$
\begin{array}{rcl}
t & ::= & n \mid x \mid t_1 \,\mathbf{op}\, t_2 \\
& & \mid \mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \\
& & \mid (t_1, t_2) \mid \mathbf{fst}(t) \mid \mathbf{snd}(t) \\
& & \mid (t_1\, t_2) \mid \lambda x.t \mid \mathbf{rec}\, y.(\lambda x.t)
\end{array}
$$

where $t, t_0, t_1, t_2 \in \mathbf{Term}$, $n \in N$, and $x, y \in \mathbf{Var}$.

The operators **fst** and **snd** denote, respectively, the *first* and *second projection* on pairs. In a pair (t_1, t_2) the first projection term t_1 and the second projection term t_2 are separated by a comma, while in a function application $(t_1\, t_2)$ no comma separates the function t_1 from the argument t_2. In a pair the projection terms are also called *components* of the pair.

Note that in the recursive definitions of the Eager language we insist that the variable y be bound to a lambda abstraction.

1.2. Syntax of the Lazy Language.

The set **Term** of the terms in the Lazy language is defined as follows.

$$
\begin{array}{rcl}
t & ::= & n \mid x \mid t_1 \,\mathbf{op}\, t_2 \\
& & \mid \mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \\
& & \mid (t_1, t_2) \mid \mathbf{fst}(t) \mid \mathbf{snd}(t) \\
& & \mid (t_1\, t_2) \mid \lambda x.t \mid \mathbf{rec}\, x.t
\end{array}
$$

where $t, t_0, t_1, t_2 \in \mathbf{Term}$, $n \in N$, and $x \in \mathbf{Var}$.

In the recursive definition of the Lazy language we do *not* insist that the variable x be bound to a lambda abstraction. The variable x can be bound to any term, thus, for instance, we also allow the term $\mathbf{rec}\, x.x{+}1$.

The syntax of the Lazy language is common to both languages, the Lazy1 and the Lazy2 language, for which we will introduce a common operational semantics, but two different denotational semantics.

In the Eager and in the Lazy languages, a function application $(t_1 t_2)$ will also be denoted by $t_1(t_2)$ or $t_1\, t_2$.

1.3. Typing Rules for the Eager Language and the Lazy Language.

Both the Eager and the Lazy languages are typed languages. For all terms t and types τ, by $t : \tau$ we indicate that the term t has type τ.

In Table 1 on the facing page we list the rules which give a type to each term of the Eager language and the Lazy language.

One can show that every term can be given exactly one type.

2. Operational Semantics of the Eager and Lazy Languages

The operational semantics rules are structural rules, that is, they are based on the syntactic structure of the terms.

As already mentioned, a *context* $C[\_]$ is a term with a missing subterm. In Table 2 on the next page we list some contexts. $C[t]$ denotes the context $C[\_]$ where we have inserted the term t in the place of the missing subterm.

Typing Rules for the Eager Language and the Lazy Language.

$n : int$ for each integer $n \in N$

$x : \tau$ (the type τ is known from the identifier of the variable x)

$$\frac{t_1 : int \quad t_2 : int}{t_1 \,\mathbf{op}\, t_2 : int}$$

$$\frac{t_0 : int \qquad t_1 : \tau \qquad t_2 : \tau}{\mathbf{if}\ t_0\ \mathbf{then}\ t_1\, \mathbf{else}\ t_2 : \tau}$$

$$\frac{t_1 : \tau_1 \qquad t_2 : \tau_2}{(t_1, t_2) : \tau_1 \times \tau_2} \qquad \frac{t : \tau_1 \times \tau_2}{\mathbf{fst}(t) : \tau_1} \qquad \frac{t : \tau_1 \times \tau_2}{\mathbf{snd}(t) : \tau_2}$$

$$\frac{x : \tau_1 \quad t : \tau_2}{\lambda x.t : \tau_1 \rightarrow \tau_2} \qquad \frac{t_1 : \tau_1 \rightarrow \tau_2 \quad t_2 : \tau_1}{(t_1\, t_2) : \tau_2}$$

$$\frac{y : \tau \quad \lambda x.t : \tau}{\mathbf{rec}\, y.(\lambda x.t) : \tau}$$ (for the Eager Language) τ is of the form: $\tau_1 \rightarrow \tau_2$

$$\frac{x : \tau \quad t : \tau}{\mathbf{rec}\, x.t : \tau}$$ (for the Lazy Language)

TABLE 1. Typing Rules for the Eager Language and the Lazy Language.

	context $C[\_]$	complete term $C[t]$
(i)	$(t_1\ [\_])$	$C[t] = (t_1\ t)$
(ii)	$[\_]\,\mathbf{op}\, t_2$	$C[t] = t\,\mathbf{op}\, t_2$
(iii)	$\mathbf{if}\ [\_]\ \mathbf{then}\ t_1\, \mathbf{else}\ t_2$	$C[t] = \mathbf{if}\ t\ \mathbf{then}\ t_1\, \mathbf{else}\ t_2$
(iv)	$[\_]$	$C[t] = t$

TABLE 2. Some examples of contexts.

Note that in Case (iv) of Table 2 the missing subterm is the whole term. In this case the context is said to be the *empty context*.

2.1. Operational Semantics of the Eager Language.

In order to present the operational semantics of the Eager language, we first define the set **Canonical Form** of the *canonical forms* of the Eager language. It is defined as follows:

$c ::= n \mid (c_1, c_2) \mid \lambda x.t$

where $c, c_1, c_2 \in$ **Canonical Form** and $\lambda x.t$ is a closed term. In what follows the variable c, possibly with subscripts, is supposed to range over the canonical forms.

The *eager operational semantics* relation $\rightarrow^e \subseteq$ **Term** $\times$ **Canonical Form** is defined by the deduction rules of Table 3. The identifiers t, t_0, t_1, and t_2 denote terms, and the identifiers c, c_1, and c_2 denote canonical forms.

The superscript e in $\rightarrow^e$ tells us that the operational semantics is for the Eager language. For reasons of simplicity, given any closed, typed term t of type and any canonical form c, we write $t \rightarrow c$, instead of $t \rightarrow^e c$. If $t \rightarrow^e c$ we also say that the eager operational evaluation of t yields the canonical form c.

Operational Semantics of the Eager Language.

$$c \rightarrow c$$

$$\frac{t_1 \rightarrow c_1 \quad t_2 \rightarrow c_2}{t_1 \,\mathbf{op}\, t_2 \rightarrow c}$$

where $c = c_1 \, op \, c_2$ and op is the semantic operation corresponding to **op**

$$\frac{t_0 \rightarrow 0 \qquad t_1 \rightarrow c_1}{\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \rightarrow c_1} \qquad \frac{t_0 \rightarrow n \quad t_2 \rightarrow c_2 \quad n \neq 0}{\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \rightarrow c_2}$$

$$\frac{t_1 \rightarrow c_1 \quad t_2 \rightarrow c_2}{(t_1, t_2) \rightarrow (c_1, c_2)}$$

$$\frac{t \rightarrow (c_1, c_2)}{\mathbf{fst}(t) \rightarrow c_1} \qquad \frac{t \rightarrow (c_1, c_2)}{\mathbf{snd}(t) \rightarrow c_2}$$

$$\frac{t_1 \rightarrow \lambda x.t \quad t_2 \rightarrow c_2 \quad t[c_2/x] \rightarrow c}{(t_1\, t_2) \rightarrow c}$$

$$\mathbf{rec}\, y.(\lambda x.t) \rightarrow \lambda x.(t[\mathbf{rec}\, y.(\lambda x.t)/y])$$

TABLE 3. Operational Semantics of the Eager Language. For reasons of simplicity, for any closed, typed term t of type τ and any canonical form c, we have written $t \rightarrow c$, instead of $t \rightarrow^e c$.

NOTE 2.1. The eager operational semantics rules define a so called *big-step semantics* in the sense that for every term t_1 and t_2, if $t_1 \rightarrow^e t_2$ then t_2 is a canonical value for the eager operational semantics.

If $t_1 \rightarrow^e t_2$ we will say that t_2 is the *eager operational value* of t_1. □

NOTE 2.2. In the definition of the eager operational semantics we do *not* have a context-rule of the form:

$$\frac{t \to c}{C[t] \to C[c]} \text{ for every context } C[\_].$$

Thus, in Table 3 on the preceding page if we replace the two rules with conclusions $\mathbf{fst}(t) \to c_1$ and $\mathbf{fst}(t) \to c_2$ by the axioms $\mathbf{fst}((c_1, c_2)) \to c_1$ and $\mathbf{fst}((c_1, c_2)) \to c_2$, respectively, we get a weaker deductive system. Indeed, for all t, c_1, c_2, in the new deductive system we cannot derive that if $t \to (c_1, c_2)$, then $\mathbf{fst}(t) \to c_1$. □

We have the following theorems whose proofs can be done by rule induction.

THEOREM 2.3. [**Determinism of the Eager Operational Semantics**]
$\forall$ terms t, $\forall$ canonical forms c_1, c_2, if $t \to^e c_1$ and $t \to^e c_2$ then $c_1 = c_2$.

THEOREM 2.4. [**The Eager Operational Semantics Preserves Types**]
$\forall$ terms $t : \tau$, $\forall$ canonical forms c, if $t \to^e c$ then $c : \tau$.

2.2. Operational Semantics of the Lazy Language.

In order to present the operational semantics of the Lazy language, we first define the set **Canonical Form** of the *canonical forms* of the Lazy language as follows:

$c ::= n \mid (t_1, t_2) \mid \lambda x.t$

where $c \in$ **Canonical Form** and t_1, t_2, and $\lambda x.t$ are closed terms. In what follows the variable c, possibly with subscripts, is supposed to range over the canonical forms.

The *lazy operational semantics* relation $\to^\ell \subseteq$ **Term** $\times$ **Canonical Form** is defined by the deduction rules of Table 4 on the following page. The identifiers t, t_0, t_1, and t_2 denote terms, and the identifiers c, c_1, and c_2 denote canonical forms.

The superscript ℓ in $\to^\ell$ tells us that the operational semantics is for the Lazy language. For reasons of simplicity, given any closed, typed term t of type and any canonical form c, we write $t \to c$, instead of $t \to^\ell c$. If $t \to^\ell c$ we also say that the lazy operational evaluation of t yields the canonical form c.

NOTE 2.5. The lazy operational semantics rules define *a big-step semantics* in the sense that for every term t_1 and t_2, if $t_1 \to^\ell t_2$ then t_2 is a canonical value for the lazy operational semantics.

If $t_1 \to^\ell t_2$ we will say that t_2 is the *lazy operational value* of t_1. □

NOTE 2.6. In the definition of the lazy operational semantics we do *not* have a context-rule of the form:

$$\frac{t \to c}{C[t] \to C[c]} \text{ for every context } C[\_].$$

□

NOTE 2.7. In the lazy operational semantics there are no rules for pairs with conclusion of the form $(t_1, t_2) \to (c_1, c_2)$ because for all terms t_1 and t_2, (t_1, t_2) is a canonical form. □

We have the following theorems whose proofs can be done by rule induction.

Operational Semantics of the Lazy Language.

$$c \to c$$

$$\frac{t_1 \to c_1 \quad t_2 \to c_2}{t_1 \,\mathbf{op}\, t_2 \to c}$$

where $c = c_1 \, op \, c_2$ and op is the semantic operation corresponding to **op**

$$\frac{t_0 \to 0 \qquad t_1 \to c_1}{\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \to c_1} \qquad \frac{t_0 \to n \quad t_2 \to c_2 \quad n \neq 0}{\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \to c_2}$$

$$\frac{t \to (t_1,t_2) \quad t_1 \to c_1}{\mathbf{fst}(t) \to c_1} \qquad \frac{t \to (t_1,t_2) \quad t_2 \to c_2}{\mathbf{snd}(t) \to c_2}$$

$$\frac{t_1 \to \lambda x.t \quad t[t_2/x] \to c}{(t_1\, t_2) \to c}$$

$$\frac{t[(\mathbf{rec}\, x.t)/x] \to c}{\mathbf{rec}\, x.t \to c}$$

TABLE 4. Operational Semantics of the Lazy Language. For reasons of simplicity, given a closed, typed term t and a canonical form c, we have written $t \to c$, instead of $t \to^{\ell} c$.

THEOREM 2.8. [**Determinism of the Lazy Operational Semantics**] $\forall$ terms t, $\forall$ canonical forms c_1, c_2, if $t \to^{\ell} c_1$ and $t \to^{\ell} c_2$ then $c_1 = c_2$.

THEOREM 2.9. [**The Lazy Operational Semantics Preserves Types**] $\forall$ terms $t{:}\tau$, $\forall$ canonical forms c, if $t \to^{\ell} c$ then $c{:}\tau$.

NOTE 2.10. Both in the eager operational semantics and lazy operational semantics the evaluation of the operations **op**'s is done by evaluating its arguments first. Analogously, the evaluation of the **if** t_0 **then** t_1 **else** t_2 is done by evaluating the condition t_0 first and then either the arm t_1 or the arm t_2, according to the value of t_0. □

NOTE 2.11. In the case of pairs there is a difference between the eager operational semantics and the lazy operational semantics, as we now indicate.
(i) In the eager semantics the canonical form of a pair is the pair of the canonical forms of the two components. For any pair t, in order to evaluate $\mathbf{fst}(t)$ we need to evaluate the canonical form of both the first and the second component of t. Analogously for the the evaluation of the second component $\mathbf{snd}(t)$.
(ii) In the lazy semantics any pair (t_1, t_2) is already in canonical form. For any pair t, in order to evaluate $\mathbf{fst}(t)$ we need to evaluate only the canonical form c_1 of t_1. Analogously for the evaluation of the second component $\mathbf{snd}(t)$.

In the case of function application there is a difference between the eager operational semantics and the lazy operational semantics, as we now indicate.
(i) In the eager operational semantics in order to evaluate the function application $(t_1\, t_2)$, we first evaluate t_1 and t_2 to their canonical forms, say c_1 and c_2, respectively, and then we evaluate the application c_1c_2.
(ii) In the lazy operational semantics in order to evaluate the function application $(t_1\, t_2)$, we first evaluate t_1 to its canonical form, say c_1, and then we evaluate the application c_1t_2.

Other differences between the eager operational semantics and the lazy operational semantics are in the syntax and semantics of the **rec** construct which are evident from the rules we have given above. □

NOTE 2.12. The eager operational rule for the **rec**, that is, the axiom

$$\mathbf{rec}\, y.(\lambda x.t) \;\rightarrow\; \lambda x.(t[\mathbf{rec}\, y.(\lambda x.t)/y])$$

follows from the usual semantics of recursion via unfolding. Indeed, the deduction rule one expects via unfolding is as follows:

$$\frac{(\lambda x.t)[(\mathbf{rec}\, y.(\lambda x.t))/y] \rightarrow c}{\mathbf{rec}\, y.(\lambda x.t) \;\rightarrow\; c}$$

and it reduces to $\mathbf{rec}\, y.(\lambda x.t) \;\rightarrow\; \lambda x.(t[\mathbf{rec}\, y.(\lambda x.t)/y])$ because:
(i) $(\lambda x.t)[(\mathbf{rec}\, y.(\lambda x.t))/y]$ is equal to $\lambda x.(t[(\mathbf{rec}\, y.(\lambda x.t))/y])$ (recall that x and y are distinct variables having distinct types), and
(ii) $\lambda x.(t[(\mathbf{rec}\, y.(\lambda x.t))/y]) \rightarrow \lambda x.(t[(\mathbf{rec}\, y.(\lambda x.t))/y])$ because any term of the form $\lambda x.t$ is a canonical form in the Eager language. □

2.3. Operational Evaluation in Linear Form.

A deduction, that is, a proof tree which justifies the operational evaluation of a given term, will often be written *in linear form*. This form is obtained by visiting the proof tree in a preorder way. Thus, for instance, instead of writing:

$$\frac{\dfrac{a \rightarrow a' \qquad b \rightarrow b'}{a + b \;\rightarrow\; c} \qquad\qquad d \;\rightarrow\; d'}{(a + b) + d \;\rightarrow\; e}$$

we will also write:

$$(a+b)+d \;\rightarrow\; (a'+b)+d \;\rightarrow\; (a'+b')+d \;\rightarrow\; c+d \;\rightarrow\; c+d' \;\rightarrow\; e.$$

Analogously, instead of writing:

$$\frac{t_1 \rightarrow \lambda x.t \qquad t_2 \rightarrow c_2 \qquad t[c_2/x] \rightarrow c}{(t_1\, t_2) \;\rightarrow\; c}$$

we will also write:

$$(t_1t_2) \rightarrow ((\lambda x.t)\, t_2) \rightarrow ((\lambda x.t)\, c_2) \rightarrow t[c_2/x] \rightarrow c.$$

In the case of the lazy operational semantics, instead of writing:

$$\frac{t[\mathbf{rec}\, x.t/x] \to c}{\mathbf{rec}\, x.\, t \to c}$$

we will also write:

$$\mathbf{rec}\, x.\, t \;\to\; t[\mathbf{rec}\, x.t/x] \;\to\; c.$$

REMARK 2.13. When the evaluation of a term t is written in linear form, the term u occurring in $t \to u$, need not be a canonical form. □

DEFINITION 2.14. [**Eager Canonical Form of a Term**] We will say that a term t has the *eager canonical form* c iff its eager operational evaluation in linear form is a sequence of the form: $t \to^e \ldots \to^e c$ and c is a term in eager canonical form.

DEFINITION 2.15. [**Lazy Canonical Form of a Term**] We will say that a term t has the *lazy canonical form* c iff its lazy operational evaluation in linear form is a sequence of the form: $t \to^\ell \ldots \to^\ell c$ and c is a term in lazy canonical form.

2.4. Eager Operational Semantics in Action: the Factorial Function.

Let us now show the evaluation, according to the eager operational semantics, of $(\mathit{fact}\ 2)$, where *fact* is the factorial function.

In order to evaluate $(\mathit{fact}\, 2)$ we have to consider the initial term:

$$((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x{-}1)))2).$$

Here is the evaluation of $(\mathit{fact}\, 2)$ according to the eager operational semantics, expressed in linear form:

$((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x{-}1)))2)$
$\to ((\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times ((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x{-}1)))\, (x{-}1)))\, 2)$
$\to \mathbf{if}\, 2\, \mathbf{then}\, 1\, \mathbf{else}\, 2 \times ((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x{-}1)))\, (2{-}1))$
$\to 2 \times ((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x{-}1)))\, (2{-}1))$ $(e2)$
$\to 2 \times ((\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times ((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x{-}1)))\, (x{-}1)))\, (2{-}1))$
$\to 2 \times ((\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times ((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x{-}1)))\, (x{-}1)))\, 1)$
$\to 2 \times (\mathbf{if}\, 1\, \mathbf{then}\, 1\, \mathbf{else}\, 1 \times ((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x{-}1)))\, (1{-}1)))$
$\to 2 \times (1 \times ((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x{-}1)))\, (1{-}1)))$ $(e1)$
$\to 2 \times (1 \times ((\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times$
$\qquad ((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x{-}1)))\, (x{-}1)))\, (1{-}1)))$
$\to 2 \times (1 \times ((\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times ((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x{-}1)))\, (x{-}1)))0))$
$\to 2 \times (1 \times (\mathbf{if}\, 0\, \mathbf{then}\, 1\, \mathbf{else}\, 0 \times ((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x{-}1)))\, (0{-}1))))$
$\to 2 \times (1 \times 1)$ $(e0)$
$\to 2 \times 1 \to 2.$

Note that the derivation from term $(e1)$ to term $(e0)$ is similar to the derivation from term $(e2)$ to term $(e1)$.

2.5. Lazy Operational Semantics in Action: the Factorial Function.

Here is the evaluation of $fact(2)$ according to the lazy operational semantics, expressed in linear form:

$((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x-1)))2)$
$\to ((\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times ((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x-1)))\,(x-1)))\,2)$
$\to \mathbf{if}\, 2\, \mathbf{then}\, 1\, \mathbf{else}\, 2 \times ((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x-1)))\,(2-1))$
$\to 2 \times ((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x-1)))\,(2-1))$ $(\ell 2)$
$\to 2\times((\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x\times((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x\times f(x–1)))\,(x–1)))\,(2–1))$
$\to 2\times(\mathbf{if}\, 2–1\, \mathbf{then}\, 1\, \mathbf{else}\, (2–1)\times(((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x\times f(x–1)))\,((2–1)–1))))$
$\to 2\times(\mathbf{if}\, 1\, \mathbf{then}\, 1\, \mathbf{else}\, (2-1)\times(((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x\times f(x-1)))\,((2-1)-1))))$
$\to 2 \times ((2-1) \times (((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x-1)))\,((2-1)-1))))$
$\to 2 \times (1 \times (((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x-1)))\,((2-1)-1))))$ $(\ell 1)$
$\to 2 \times (1 \times (\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x\times$
$\qquad ((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x-1)))\,(x-1))\,((2-1)-1)))$
$\to 2 \times (1 \times (\mathbf{if}\, ((2-1)-1)\, \mathbf{then}\, 1\, \mathbf{else}\, ((2-1)-1)\times$
$\qquad ((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x-1)))\,(((2-1)-1)-1))))$
$\to 2 \times (1 \times (\mathbf{if}\, (1-1)\, \mathbf{then}\, 1\, \mathbf{else}\, ((2-1)-1)\times$
$\qquad ((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x-1)))\,(((2-1)-1)-1))))$
$\to 2 \times (1 \times (\mathbf{if}\, 0\, \mathbf{then}\, 1\, \mathbf{else}\, ((2-1)-1)\times$
$\qquad ((\mathbf{rec}\, f.(\lambda x.\, \mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x-1)))\,(((2-1)-1)-1))))$
$\to 2 \times (1 \times 1)$ $(\ell 0)$
$\to 2 \times 1 \to 2$

Note that the derivation from term $(\ell 1)$ to term $(\ell 0)$ is similar to the derivation from term $(\ell 2)$ to term $(\ell 1)$.

2.6. Operational Semantics of the let-in construct.

Let us assume that we have the following syntactic construct:

$\mathbf{let}\ x \Leftarrow t_1\ \mathbf{in}\ t$

where x does not occur in t_1, with the following typing rule:

$$\frac{x : \tau_1 \qquad t_1 : \tau_1 \qquad t : \tau}{\mathbf{let}\, x \Leftarrow t_1\, \mathbf{in}\, t : \tau}$$

When considering the **let-in** construct from now on we will assume that the associated term $(\lambda x.t)\, t_1$ is closed and typed. For the **let-in** construct we have the following eager operational semantics rule:

$$\frac{t_1 \to c_1 \qquad t[c_1/x] \to c}{\mathbf{let}\ x \Leftarrow t_1\ \mathbf{in}\ t \to c}$$

and the following lazy operational semantics:

$$\frac{t[t_1/x] \to c}{\mathbf{let}\ x \Leftarrow t_1\ \mathbf{in}\ t \to c}$$

The constructs **let** $x \Leftarrow t_1$ **in** t and $((\lambda x.t)\, t_1)$ have the same eager operational semantics. Indeed, we have that:

$$\frac{t_1 \to c_1 \quad t[c_1/x] \to c}{\textbf{let } x \Leftarrow t_1 \textbf{ in } t \to c} \quad \text{and} \quad \frac{\lambda x.t \to \lambda x.t \quad t_1 \to c_1 \quad t[c_1/x] \to c}{((\lambda x.t)\, t_1) \to c}$$

and, obviously, $\lambda x.t \ \to \ \lambda x.t$ always holds because $\lambda x.t$ is a canonical form.

The constructs **let** $x \Leftarrow t_1$ **in** t and $((\lambda x.t)\, t_1)$ also have the same lazy operational semantics. Indeed, we have that:

$$\frac{t[t_1/x] \to c}{\textbf{let } x \Leftarrow t_1 \textbf{ in } t \to c} \quad \text{and} \quad \frac{\lambda x.t \to \lambda x.t \quad t[t_1/x] \to c}{((\lambda x.t)\, t_1) \to c}$$

and, obviously, $\lambda x.t \ \to \ \lambda x.t$ holds because $\lambda x.t$ is a canonical form.

3. Denotational Semantics of the Eager, Lazy1, and Lazy2 Languages

In this section we will present the denotational semantics of three languages: the Eager language, the Lazy1 language, and the Lazy2 language.

As already said, for all terms t and types τ, $t : \tau$ denotes that the type of t is τ. The domain of values of type τ is the cpo V_τ.

Given any term t of type τ, its canonical form c depends on τ. In Table 5 we have indicated the canonical forms of a term t of type τ in the Eager, Lazy1, and Lazy2 languages for: (i) the type *int*, (ii) the product type $\tau_1 \times \tau_2$, and (iii) the function type $\tau_1 \to \tau_2$. The corresponding semantic domains are indicated in Table 6.

In Table 5 we have that: (i) τ_1 is the type of the canonical form c_1, the term t_1 and the variable x, and (ii) τ_2 is the type of the canonical form c_2, the term t_2, and the term t.

Type τ	Eager Language	Lazy1 Language and Lazy2 Language
$\tau = int$	n	n
$= \tau_1 \times \tau_2$	(c_1, c_2)	(t_1, t_2)
$= \tau_1 \to \tau_2$	$\lambda x.t$ closed	$\lambda x.t$ closed

TABLE 5. The canonical forms for the Eager, Lazy1, and Lazy2 languages. n denotes an integer. c_1, c_2 denote canonical forms. t, t_1, t_2 denote terms. x denotes a variable of type τ_1, and t denotes a term of type τ_2.

Type τ	Eager Semantics	Lazy1 Semantics	Lazy2 Semantics
	$\rho \in \mathbf{Var} \to \bigcup_\tau V_\tau$	$\rho \in \mathbf{Var} \to \bigcup_\tau (V_\tau)_\perp$	$\rho \in \mathbf{Var} \to \bigcup_\tau V_\tau$
$\tau = int$ $= \tau_1 \times \tau_2$ $= \tau_1 \to \tau_2$	$V_\tau = N$ $= V_{\tau_1} \times V_{\tau_2}$ $= [V_{\tau_1} \to (V_{\tau_2})_\perp]$	$V_\tau = N$ $= (V_{\tau_1})_\perp \times (V_{\tau_2})_\perp$ $= [(V_{\tau_1})_\perp \to (V_{\tau_2})_\perp]$	$V_\tau = N_\perp$ $= V_{\tau_1} \times V_{\tau_2}$ $= [V_{\tau_1} \to V_{\tau_2}]$
	$[\![t]\!]^e \rho \in (V_\tau)_\perp$	$[\![t]\!]^{\ell 1} \rho \in (V_\tau)_\perp$	$[\![t]\!]^{\ell 2} \rho \in V_\tau$

TABLE 6. The semantic domains for the denotational semantics of the Eager, Lazy1, and Lazy2 languages for: (i) the type *int*, (ii) the product types, and (iii) the function types.

The eager denotational semantics is the function $[\![\_]\!]^e \in \mathit{Term} \times \mathit{Env} \to (V_\tau)_\perp$ where τ is the type of the term $t \in \mathit{Term}$ and *Env* is the set of environments, that is, $\mathbf{Var} \to \bigcup_\tau V_\tau$. Analogously, the lazy1 denotational semantics is the function $[\![\_]\!]^{\ell 1} \in \mathit{Term} \times \mathit{Env} \to (V_\tau)_\perp$ where τ is the type of the term $t \in \mathit{Term}$ and *Env* is the set of environments, that is, $\mathbf{Var} \to \bigcup_\tau (V_\tau)_\perp$. The lazy2 denotational semantics is the function $[\![\_]\!]^{\ell 2} \in \mathit{Term} \times \mathit{Env} \to V_\tau$ where τ is the type of the term $t \in \mathit{Term}$ and *Env* is the set of environments, that is, $\mathbf{Var} \to \bigcup_\tau V_\tau$.

As we have seen in the case of the language REC (see Remark 5.1 on page 174), the domains of the environments in the Lazy1 language (that are of the form $\bigcup_\tau V_\tau$) differ from those of the Eager language (that are of the form $\bigcup_\tau (V_\tau)_\perp$). This is due to the fact that in the case of the Lazy1 language, arguments of functions may be $\perp$'s, that is, values of non-terminating computations.

Then, this change of the domains is propagated from the environments $\rho \in \mathit{Env}$ both to the function types (that are of the form $\tau_1 \to \tau_2$) and to the product types (that are of the form $\tau_1 \times \tau_2$), because of the isomorphism between $[[F \times D] \to E]$ and $[F \to [D \to E]]$ (see page 88 and Figure 5 on page 89).

3.1. Denotational Semantics of the Eager Language.

In Table 7 on the next page we present the denotational semantics of the Eager language, also called *eager denotational semantics*. For simplicity, when understood from the context, we write $[\![\_]\!]$, instead of $[\![\_]\!]^e$.

Here are some notes on the definition of the eager denotational semantics of Table 7.

NOTE 3.1. The denotational semantics for the Eager language satisfies the following context-rule: for all context $C[\_]$, for all typable, closed terms t_1 and t_2 such that $C[t_1]$ and $C[t_2]$ are typable, closed terms,

if $[\![t_1]\!] \rho = [\![t_2]\!] \rho$ then $[\![C[t_1]]\!] \rho = [\![C[t_2]]\!] \rho$. □

In the right hand side of the semantic equation for $[\![\lambda x.t]\!] \rho$ the variable v is a fresh, new variable. Moreover, that right hand side cannot be simplified to $\lfloor \lambda x.[\![t]\!] \rho \rfloor$. If we make this wrong simplification, in fact, the eager denotational semantics of the

Denotational Semantics of the Eager language.

$$[\![n]\!]\rho = \lfloor n \rfloor$$
$$[\![x]\!]\rho = \lfloor \rho(x) \rfloor$$
$$[\![t_1 \,\mathbf{op}\, t_2]\!]\rho = [\![t_1]\!]\rho \; op_\perp \; [\![t_2]\!]\rho$$
$$[\![\mathbf{if}\, t_0 \,\mathbf{then}\, t_1 \,\mathbf{else}\, t_2]\!]\rho = Cond([\![t_0]\!]\rho, [\![t_1]\!]\rho, [\![t_2]\!]\rho)$$

$$[\![(t_1, t_2)]\!]\rho = let\, v_1 \Leftarrow [\![t_1]\!]\rho,\, v_2 \Leftarrow [\![t_2]\!]\rho \centerdot \lfloor (v_1, v_2) \rfloor$$
$$[\![\mathbf{fst}(t)]\!]\rho = let\, v \Leftarrow [\![t]\!]\rho \centerdot \lfloor \pi_1(v) \rfloor$$
$$[\![\mathbf{snd}(t)]\!]\rho = let\, v \Leftarrow [\![t]\!]\rho \centerdot \lfloor \pi_2(v) \rfloor$$

$$[\![t_1 t_2]\!]\rho = let\, \varphi \Leftarrow [\![t_1]\!]\rho,\, v \Leftarrow [\![t_2]\!]\rho \centerdot \varphi(v)$$
$$[\![\lambda x.t]\!]\rho = \lfloor \lambda v.[\![t]\!]\rho[v/x] \rfloor$$
$$[\![\mathbf{rec}\, y.(\lambda x.t)]\!]\rho = \lfloor fix\,(\lambda f.\lambda v.[\![t]\!]\rho[f/y,\, v/x]) \rfloor \qquad (\mathbf{rec}\text{-Glynn})$$
$$= fix\,(\lambda f.[\![\lambda x.t]\!]\rho[down(f)/y]) \qquad (\mathbf{rec}\text{-Alberto})$$

TABLE 7. Denotational Semantics of the Eager language. For simplicity, we have written $[\![\_]\!]$, instead of $[\![\_]\!]^e$.

application $(\lambda x.x)\,4$ in the environment ρ such that $\rho(x) = 1$, is $\lfloor 1 \rfloor$, instead of the expected value $\lfloor 4 \rfloor$. Indeed,

$[\![(\lambda x.x)\,4]\!]\rho =$
$= (\lambda x.([\![x]\!]\rho))\,4 =$
$= (\lambda x.\lfloor 1 \rfloor)\,4 = \lfloor 1 \rfloor.$

Note also that in the expression $(\lambda x.([\![x]\!]\rho))\,4$, the binder λx, which acts on the semantics, binds the variable x in $[\![x]\!]$, which is syntactic term, and this is an indication that the expression $\lambda x.([\![x]\!]\rho)$ is not correct. □

The *fix* operator in the expression (**rec**-Glynn) is different from the *fix* operator in the expression (**rec**-Alberto) because they belong to two different cpo's (see also Note 3.3 below). Actually, there is an operator $fix \in [[V_\tau \to V_\tau] \to V_\tau]$, for each cpo V_τ associated with the type τ. □

As already mentioned, instead of $fix(\lambda f.e)$, we will also write $\mu f.e$. By definition of the minimal fixpoint $fix(\lambda f.e)$ of the function $\lambda f.e$, we have that: $\mu f.e = (\lambda f.e)(\mu f.e)$. □

NOTE 3.2. The semantic function $op_\perp$ is the strict function in $[N_\perp \times N_\perp \to N_\perp]$ associated with **op**. Thus, for all x and $y \in N_\perp$,

$$x\, op_\perp\, y = \begin{cases} \perp & if\ x = \perp\ or\ y = \perp \\ \lfloor x' + y' \rfloor & if\ x = \lfloor x' \rfloor\ and\ y = \lfloor y' \rfloor\ for\ some\ x', y' \in N \end{cases}$$ □

NOTE 3.3. Let us assume that in the term $\mathbf{rec}\, y.(\lambda x.t)$ we have that:

(i) $y : \tau_1 \to \tau_2$
(ii) $x : \tau_1$
(iii) $t : \tau_2$

Then in Equations (**rec**-Glynn) and (**rec**-Alberto) we have that:

$$[\![\, \mathbf{rec}\, y.(\lambda x.t) \,]\!]\, \rho \in [V_{\tau_1} \to (V_{\tau_2})_\perp]_\perp$$
$$\rho \in \mathbf{Var} \to V_{\tau_1} \cup [V_{\tau_1} \to (V_{\tau_2})_\perp]$$

In Equation (**rec**-Glynn) we also have that:

$$f \in [V_{\tau_1} \to (V_{\tau_2})_\perp]$$
$$v \in V_{\tau_1}$$
$$[\![\, t \,]\!]\, \rho \in (V_{\tau_2})_\perp$$
$$\mathit{fix} \in [[V_{\tau_1} \to (V_{\tau_2})_\perp] \to [V_{\tau_1} \to (V_{\tau_2})_\perp]] \to [V_{\tau_1} \to (V_{\tau_2})_\perp]$$

In Equation (**rec**-Alberto) we also have that:

$$f \in [V_{\tau_1} \to (V_{\tau_2})_\perp]_\perp$$
$$[\![\, \lambda x.t \,]\!]\, \rho \in [V_{\tau_1} \to (V_{\tau_2})_\perp]_\perp$$
$$\mathit{down}(f) \in [V_{\tau_1} \to (V_{\tau_2})_\perp]$$
$$\mathit{fix} \in [[V_{\tau_1} \to (V_{\tau_2})_\perp]_\perp \to [V_{\tau_1} \to (V_{\tau_2})_\perp]_\perp] \to [V_{\tau_1} \to (V_{\tau_2})_\perp]_\perp$$ □

Now we will show that the (**rec**-Glynn) and the (**rec**-Alberto) definitions of the eager denotational semantics of $[\![\, \mathbf{rec}\, y.(\lambda x.t) \,]\!]\, \rho$ (see Table 7 on the preceding page), are equivalent in the sense that they define the same semantic value. It is enough to show that:

$$\lfloor \mathit{fix}(\lambda f.d) \rfloor = \mathit{fix}(\lambda f.\lfloor d[\mathit{down}(f)/f] \rfloor) \qquad (G)$$

where: (i) d is of the form $\lambda x.t$ for some $x \in V_{\tau_1}$ and some term $t \in (V_{\tau_2})_\perp$, and (ii) $d[\mathit{down}(f)/f]$ is the expression d where each occurrence of f has been replaced by $\mathit{down}(f)$.
Thus, on the left hand side of (G):

$$f \in [V_{\tau_1} \to (V_{\tau_2})_\perp]$$
$$d \in [V_{\tau_1} \to (V_{\tau_2})_\perp]$$
$$\mathit{fix} \in [[V_{\tau_1} \to (V_{\tau_2})_\perp] \to [V_{\tau_1} \to (V_{\tau_2})_\perp]] \to [V_{\tau_1} \to (V_{\tau_2})_\perp]$$

while on the right hand side of (G):

$$f \in [V_{\tau_1} \to (V_{\tau_2})_\perp]_\perp$$
$$d \in [V_{\tau_1} \to (V_{\tau_2})_\perp] \quad (\text{not } [V_{\tau_1} \to (V_{\tau_2})_\perp]_\perp)$$
$$\mathit{fix} \in [[V_{\tau_1} \to (V_{\tau_2})_\perp]_\perp \to [V_{\tau_1} \to (V_{\tau_2})_\perp]_\perp] \to [V_{\tau_1} \to (V_{\tau_2})_\perp]_\perp$$

Thus, in the equation (G) the two occurrences of *fix* denote two different fixpoint operators because they belong to two different cpo's.

We first show that:

$$\lfloor \mathit{fix}(\lambda f.d) \rfloor \sqsubseteq \mathit{fix}(\lambda f.\lfloor d[\mathit{down}(f)/f] \rfloor) \qquad (G1)$$

and then we show that:

$$\lfloor \mathit{fix}(\lambda f.d) \rfloor \sqsupseteq \mathit{fix}(\lambda f.\lfloor d[\mathit{down}(f)/f] \rfloor) \qquad (G2)$$

Proof of $(G1)$. Since $\lfloor \_ \rfloor$ is continuous, we have that $\lfloor \mathit{fix}(\lambda f.d) \rfloor = \mathit{fix} \lfloor (\lambda f.d) \rfloor$. Thus, we have to show that the following inequality holds in $[V_{\tau_1} \to (V_{\tau_2})_\perp]_\perp$:

$$\bigsqcup_{i \in \omega} \lfloor (\lambda f.d)^i(\perp) \rfloor \sqsubseteq \bigsqcup_{i \in \omega} (\lambda f.\lfloor d[\mathit{down}(f)/f] \rfloor)^i(\perp) \qquad (G1.1)$$

where: (i) $\perp$ on the left hand side is the function $\lambda x \in V_{\tau_1}.\, \perp\, (\in (V_{\tau_2})_\perp)$, while (ii) $\perp$ on the right hand side is the bottom element of $[V_{\tau_1} \to (V_{\tau_2})_\perp]_\perp$. We will prove $(G1.1)$ by showing that

for all $i \geq 0$, $\lfloor (\lambda f.d)^i(\perp) \rfloor \sqsubseteq \bigsqcup_{i \in \omega} (\lambda f.\lfloor d[\mathit{down}(f)/f] \rfloor)^i(\perp)$.

The proof is by induction on i.

(*Basis*) We have to show that $\lfloor \perp \rfloor \sqsubseteq \bigsqcup_{i \in \omega} (\lambda f.\lfloor d[\mathit{down}(f)/f] \rfloor)^i(\perp)$.
This holds because $\lfloor d[\mathit{down}(f)/f] \rfloor \neq \perp \in [V_{\tau_1} \to (V_{\tau_2})_\perp]_\perp$, and thus,

$\lfloor \perp \rfloor \sqsubseteq (\lambda f.\lfloor d[\mathit{down}(f)/f] \rfloor)\, (\perp)$.

To better understand this inequality, let us consider the types of its subexpressions. Let V_τ denote the cpo $[V_{\tau_1} \to (V_{\tau_2})_\perp]$. We have that on the left hand side $\perp$ is in V_τ, while on the right hand side the first two occurrences of f are in $(V_\tau)_\perp$, the third occurrence of f is in V_τ, and the occurrence of $\perp$ is in $(V_\tau)_\perp$.

(*Step*) Assume that $\lfloor (\lambda f.d)^i(\perp) \rfloor \sqsubseteq \mathit{fix}(\lambda f.\lfloor d[\mathit{down}(f)/f] \rfloor)$. We have to show that:

$\lfloor (\lambda f.d)((\lambda f.d)^i(\perp)) \rfloor \sqsubseteq \mathit{fix}(\lambda f.\lfloor d[\mathit{down}(f)/f] \rfloor)$.

Indeed, for the left hand side we have that:

$$\begin{aligned} &\lfloor (\lambda f.d)((\lambda f.d)^i(\perp)) \rfloor = \\ &= \lfloor d[(\lambda f.d)^i(\perp)/f] \rfloor = \{\text{by } \mathit{down}(\lfloor x \rfloor) = x\ \} = \\ &= \lfloor d[\mathit{down}(\lfloor (\lambda f.d)^i(\perp) \rfloor)/f] \rfloor. \end{aligned} \qquad (G1.2)$$

For the right hand side we have that:

$$\begin{aligned} &\mathit{fix}(\lambda f.\lfloor d[\mathit{down}(f)/f] \rfloor) = \\ &= (\lambda f.\lfloor d[\mathit{down}(f)/f] \rfloor)\ (\mathit{fix}(\lambda f.\lfloor d[\mathit{down}(f)/f] \rfloor)) = \\ &= \lfloor d[\mathit{down}(\mathit{fix}(\lambda f.\lfloor d[\mathit{down}(f)/f] \rfloor))/f] \rfloor. \end{aligned} \qquad (G1.3)$$

Now we have that $(G1.2) \sqsubseteq (G1.3)$, by induction hypothesis, and monotonicity of *down*, substitution, and $\lfloor \_ \rfloor$. This completes the proof of $(G1)$.

Proof of $(G2)$. We have to show that the following inequality holds in $[V_{\tau_1} \to (V_{\tau_2})_\perp]_\perp$:

$$\lfloor \bigsqcup_{i \in \omega} (\lambda f.d)^i(\perp) \rfloor \sqsupseteq \bigsqcup_{i \in \omega} (\lambda f.\lfloor d[\mathit{down}(f)/f] \rfloor)^i(\perp) \qquad (G2.1)$$

We will prove $(G2.1)$ by showing that:

for all $i \geq 0$, $\lfloor \bigsqcup_{i \in \omega} (\lambda f.d)^i(\perp) \rfloor \sqsupseteq (\lambda f.\lfloor d[\mathit{down}(f)/f] \rfloor)^i(\perp)$.

The proof is by induction on i.

(*Basis*) Obviously, we have that: $\lfloor \bigsqcup_{i \in \omega} (\lambda f.d)^i(\perp) \rfloor \sqsupseteq \perp \in [V_{\tau_1} \to (V_{\tau_2})_\perp]_\perp$.

(*Step*) Assume that $\lfloor \mathit{fix}(\lambda f.d) \rfloor \sqsupseteq (\lambda f.\lfloor d[\mathit{down}(f)/f] \rfloor)^i(\perp)$. We have to show that:

$\lfloor \mathit{fix}(\lambda f.d) \rfloor \sqsupseteq (\lambda f.\lfloor d[\mathit{down}(f)/f] \rfloor)\ ((\lambda f.\lfloor d[\mathit{down}(f)/f] \rfloor)^i(\perp))$.

Indeed, for the left hand side we have that:

$$\lfloor d[\mathit{fix}(\lambda f.d)/f]\rfloor = \{\text{by } \mathit{down}(\lfloor x\rfloor) = x\ \} =$$
$$= \lfloor d[\mathit{down}(\lfloor \mathit{fix}(\lambda f.d)\rfloor)\ /f]\rfloor. \qquad (G2.2)$$

For the right hand side we have that:

$$(\lambda f.\lfloor d[\mathit{down}(f)/f]\rfloor)\ ((\lambda f.\lfloor d[\mathit{down}(f)/f]\rfloor)^i(\bot)) =$$
$$=\lfloor d[\mathit{down}((\lambda f.\lfloor d[\mathit{down}(f)/f]\rfloor)^i(\bot))/f]\rfloor \qquad (G2.3)$$

Now we have that $(G2.2) \sqsupseteq (G2.3)$ by induction hypothesis, and monotonicity of *down*, substitution, and $\lfloor\_\rfloor$. This completes the proof of $(G2)$.

3.2. Computing the Factorial in the Eager Denotational Semantics.

Now let us see in action the eager denotational semantics and let us compute the value of $[\![(\mathbf{rec}\ \mathit{fact}.\ \lambda x.\ \mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x{\times}\mathit{fact}(x-1))\,(2)]\!]$ which is the factorial of 2. We have:

$$[\![(\mathbf{rec}\ \mathit{fact}.\ \lambda x.\ \mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x{\times}\mathit{fact}(x-1))\,(2)]\!]\ \rho =$$
$$= \mathit{let}\ \varphi \Leftarrow [\![\mathbf{rec}\ \mathit{fact}.\ \lambda x.\ \mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x{\times}\mathit{fact}(x-1)]\!]\,\rho,\ v \Leftarrow [\![2]\!]\,\rho\,\centerdot\,\varphi(v) =$$
$$= \mathit{let}\ \varphi \Leftarrow \lfloor \mu\psi.\lambda v.[\![\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x{\times}\mathit{fact}(x-1)]\!]$$
$$\rho[\psi/\mathit{fact},\ v/x]\rfloor,\ v \Leftarrow \lfloor 2\rfloor\,\rho\,\centerdot\,\varphi(v) =$$
$$= (\mu\psi.\lambda v.[\![\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x{\times}\mathit{fact}(x-1)]\!]\,\rho[\psi/\mathit{fact},\ v/x])\ 2 \qquad (E2)$$

Now we may proceed by computing by mathematical induction the minimal fixpoint $\mu\psi.\lambda v.[\![\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x{\times}\mathit{fact}(x-1)]\!]\,\rho[\psi/\mathit{fact},\ v/x]$.

A different way of proceeding is to use the following equation (see page 99):

$$\mu x.t\ =\ (\lambda x.t)(\mu x.t) \qquad (\mathit{FixUnfold})$$

This equation characterizes the minimal fixpoint $\mu x.t$ and allows us to unfold the fixpoint as many times as it is required by the value of the argument. We clarify this point by showing how to compute the factorial of 2 by performing the unfolding of the fixpoint

$$\mu\psi.\lambda v.[\![\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x{\times}\mathit{fact}(x-1)]\!]\,\rho[\psi/\mathit{fact},\ v/x]$$

occurring in Expression $(E2)$.

From $(E2)$ by applying $(\mathit{FixUnfold})$, we get:

$$(\lambda\psi.\lambda v.[\![\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x{\times}\mathit{fact}(x-1)]\!]\ \rho[\psi/\mathit{fact},\ v/x])$$
$$(\mu\psi.\lambda v.[\![\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x{\times}\mathit{fact}(x-1)]\!]\,\rho[\psi/\mathit{fact}, v/x])\,(2) =$$
$$= [\![\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x{\times}\mathit{fact}(x-1)]\!]$$
$$\rho[(\mu\psi.\lambda v.[\![\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x{\times}\mathit{fact}(x-1)]\!]\,\rho[\psi/\mathit{fact}, v/x])/\mathit{fact},\ 2/x] =$$
$$= \mathit{Cond}([\![x]\!]\rho[\ldots,\ 2/x],\ \ [\![1]\!]\rho[\ldots,\ 2/x],\ \ [\![x{\times}\mathit{fact}(x-1)]\!]\,\rho[\ldots,\ 2/x]) =$$
$$= \mathit{Cond}(\lfloor 2\rfloor,\ \ [\![1]\!]\rho[\ldots],\ \ [\![x{\times}\mathit{fact}(x-1)]\!]\,\rho[\ldots]) =$$
$$= [\![x{\times}\mathit{fact}(x{-}1)]\!]\,\rho[(\mu\psi.\lambda v.[\![\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x{\times}\mathit{fact}(x{-}1)]\!]$$
$$\rho[\psi/\mathit{fact}, v/x])/\mathit{fact},\ 2/x] =$$
$$= [\![x]\!]\,\rho[\ldots, 2/x]\ \times_\bot\ [\![\mathit{fact}(x-1)]\!]\,\rho[\ldots, 2/x] =$$
$$= \lfloor 2\rfloor\ \times_\bot\ (\mathit{let}\ \varphi \Leftarrow [\![\mathit{fact}]\!]\rho[\ldots, 2/x],\ v \Leftarrow [\![x-1]\!]\rho[\ldots, 2/x].\varphi(v)) =$$
$$= \lfloor 2\rfloor\ \times_\bot\ (\mathit{let}\ \varphi \Leftarrow \lfloor \mu\psi.\lambda v.[\![\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x{\times}\mathit{fact}(x-1)]\!]\,\rho[\psi/\mathit{fact}, v/x]\rfloor,$$
$$v \Leftarrow [\![x]\!]\rho[\ldots, 2/x]\ -_\bot\ [\![1]\!]\rho[\ldots, 2/x].\varphi(v) =$$

$$= \lfloor 2 \rfloor \times_\perp ((\mu\psi.\lambda v.[\![\textbf{if } x \textbf{ then } 1 \textbf{ else } x{\times}fact(x-1)]\!]\,\rho[\psi/fact, v/x]) \\ (down(\lfloor 2 \rfloor -_\perp \lfloor 1 \rfloor))) =$$

$$= \lfloor 2 \rfloor \times_\perp ((\mu\psi.\lambda v.[\![\textbf{if } x \textbf{ then } 1 \textbf{ else } x{\times}fact(x-1)]\!]\,\rho[\psi/fact, v/x])\,(1)) \qquad (E1)$$

Now if we compare the expressions $(E1)$ and $(E2)$, we see that, by performing from $(E1)$ a sequence of evaluation steps analogous to the sequence which leads from $(E2)$ to $(E1)$, we eventually get:

$$\lfloor 2 \rfloor \times_\perp (\lfloor 1 \rfloor \times_\perp ((\mu\psi.\lambda v.[\![\textbf{if } x \textbf{ then } 1 \textbf{ else } x{\times}fact(x-1)]\!]\,\rho[\psi/fact, v/x])\,(0)))$$

Then, after a few more evaluation steps we get:

$\lfloor 2 \rfloor \times_\perp (\lfloor 1 \rfloor \times_\perp \lfloor 1 \rfloor) = \lfloor 2 \rfloor$, as expected.

3.3. Two Equivalent Expressions in the Eager Denotational Semantics.

In this section we will show that the following equality holds in the eager denotational semantics:

$$[\![((\lambda f.(fx))\,(\textbf{rec}f.\lambda x.t)]\!]\,\rho = [\![((\textbf{rec}f.\lambda x.t)\,x)]\!]\,\rho \qquad (RecDef)$$

For the left hand side we have:

$$[\![((\lambda f.(fx))\,(\textbf{rec}f.\lambda x.t)]\!]\,\rho =$$
$$= let\ \varphi \Leftarrow [\![\lambda f.(fx)]\!]\,\rho,\ v \Leftarrow [\![\textbf{rec}f.(\lambda x.\,t)]\!]\,\rho \,.\, \varphi(v) =$$
$$= let\ \varphi \Leftarrow \lfloor \lambda g.[\![fx]\!]\,\rho[g/f] \rfloor,\ v \Leftarrow \lfloor \mu g.\lambda v.\,[\![t]\!]\,\rho[g/f,\, v/x] \rfloor \,.\, \varphi(v) =$$
$$= (\lambda g.[\![fx]\!]\,\rho[g/f])(\mu g.\lambda v.\,[\![t]\!]\,\rho[g/f,\, v/x]) =$$
$$= [\![fx]\!]\rho[(\mu g.\lambda v.\,[\![t]\!]\,\rho[g/f,\, v/x])/f] =$$
$$= let\ \varphi \Leftarrow \lfloor \mu g.\lambda v.\,[\![t]\!]\,\rho[g/f,\, v/x] \rfloor,\ v \Leftarrow \lfloor \rho(x) \rfloor .\, \varphi(v) =$$
$$= (\mu g.\lambda v.\,[\![t]\!]\,\rho[g/f,\, v/x])\,\rho(x).$$

For the right hand side we have:

$$[\![((\textbf{rec}f.(\lambda x.t))\,x)]\!]\,\rho =$$
$$= let\ \varphi \Leftarrow [\![\textbf{rec}f.(\lambda x.t)]\!]\,\rho,\ v \Leftarrow [\![x]\!]\,\rho \,.\, \varphi(v) =$$
$$= let\ \varphi \Leftarrow \lfloor \mu g.\lambda v.\,[\![t]\!]\,\rho[g/f,\, v/x] \rfloor,\ v \Leftarrow \lfloor \rho(x) \rfloor \,.\, \varphi(v) =$$
$$= (\mu g.\lambda v.\,[\![t]\!]\,\rho[g/f,\, v/x])\,\rho(x).$$

Thus, Equation $(RecDef)$ has been proved. □

NOTE 3.4. (Equation $(RecDef)$ is a consequence of the β-rule in lambda calculus. However, the β-rule does *not* hold, in general, in the eager denotational semantics as we will see later. □

As a consequence of the above Equation $(RecDef)$, we have, for instance, that:

$$[\![(\lambda fact.\,fact(2))\ (\textbf{rec}\ fact.\ \lambda x.\ \textbf{if } x \textbf{ then } 1 \textbf{ else } x{\times}fact(x-1))]\!]\rho =$$
$$= [\![(\textbf{rec}\ fact.\ \lambda x.\ \textbf{if } x \textbf{ then } 1 \textbf{ else } x{\times}fact(x-1))(2)]\!]\rho.$$

*Denotational Semantics of the Lazy*1 *Language.*

$$[\![\, n \,]\!]\, \rho \;=\; \lfloor n \rfloor$$
$$[\![\, x \,]\!]\, \rho \;=\; \rho(x)$$
$$[\![\, t_1 \,\mathbf{op}\, t_2 \,]\!]\, \rho \;=\; [\![\, t_1 \,]\!]\, \rho \;\; op_\perp \;\; [\![\, t_2 \,]\!]\, \rho$$
$$[\![\, \mathbf{if}\, t_0 \,\mathbf{then}\, t_1 \,\mathbf{else}\, t_2 \,]\!]\, \rho \;=\; \mathit{Cond}([\![\, t_0 \,]\!]\, \rho, [\![\, t_1 \,]\!]\, \rho, [\![\, t_2 \,]\!]\, \rho)$$
$$[\![\, (t_1, t_2) \,]\!]\, \rho \;=\; \lfloor ([\![\, t_1 \,]\!]\, \rho,\; [\![\, t_2 \,]\!]\, \rho) \rfloor$$
$$[\![\, \mathbf{fst}(t) \,]\!]\, \rho \;=\; \mathit{let}\, v \Leftarrow [\![\, t \,]\!]\, \rho \,\centerdot\, \pi_1(v)$$
$$[\![\, \mathbf{snd}(t) \,]\!]\, \rho \;=\; \mathit{let}\, v \Leftarrow [\![\, t \,]\!]\, \rho \,\centerdot\, \pi_2(v)$$
$$[\![\, t_1 t_2 \,]\!]\, \rho \;=\; \mathit{let}\, \varphi \Leftarrow [\![\, t_1 \,]\!]\, \rho \,\centerdot\, \varphi([\![t_2]\!]\rho)$$
$$[\![\, \lambda x.t \,]\!]\, \rho \;=\; \lfloor \lambda v.[\![\, t \,]\!]\, \rho\, [v/x] \rfloor$$
$$[\![\, \mathbf{rec}\, x.t \,]\!]\, \rho \;=\; \mathit{fix}(\lambda f.[\![\, t \,]\!]\, \rho[f/x])$$

TABLE 8. Denotational Semantics of the Lazy1 Language. For simplicity, we have written $[\![\_]\!]$, instead of $[\![\_]\!]^{\ell 1}$.

3.4. Denotational Semantics of the Lazy1 and Lazy2 Languages.

In Table 8 on page 215 and in Table 9 on page 216 we present the denotational semantics of the languages Lazy1 and Lazy2, also called *lazy*1 *denotational semantics* and *lazy*2 *denotational semantics*, respectively. For simplicity, when understood from the context, we write $[\![\_]\!]$, instead of $[\![\_]\!]^{\ell 1}$ or $[\![\_]\!]^{\ell 2}$, for the lazy1 and the lazy2 denotational semantics, respectively.

NOTE 3.5. As in the case of the denotational semantics for the Eager language, the denotational semantics for the languages Lazy1 and Lazy2 satisfy the following context-rule:

for all context $C[\_]$, for all typable, closed terms t_1 and t_2 such that $C[t_1]$ and $C[t_2]$ are typable, closed terms,

if $[\![\, t_1 \,]\!]\, \rho = [\![t_2]\!]\, \rho$ then $[\![\, C[t_1] \,]\!]\, \rho = [\![C[t_2]]\!]\, \rho$. □

3.5. Computing the Factorial in the Lazy1 Denotational Semantics.

Let us see in action the lazy1 denotational semantics and let us compute the value of $[\![(\mathbf{rec}\ \mathit{fact}.\ \lambda x.\ \mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x{\times}\mathit{fact}(x-1))\,(2)]\!]$. We have that:

$$[\![(\mathbf{rec}\ \mathit{fact}.\ \lambda x.\ \mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x{\times}\mathit{fact}(x-1))\,(2)]\!]\, \rho =$$
$$= \mathit{let}\ \varphi \Leftarrow [\![\mathbf{rec}\ \mathit{fact}.\ \lambda x.\ \mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x{\times}\mathit{fact}(x-1)]\!]\, \rho \,\centerdot\, \varphi([\![2]\!]\, \rho) =$$
$$= \mathit{let}\ \varphi \Leftarrow (\mu\psi.[\![\lambda x.\,\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x{\times}\mathit{fact}(x-1)]\!]\ \rho[\psi/\mathit{fact}]) \,\centerdot\, \varphi(\lfloor 2 \rfloor) =$$
$$= (\mathit{down}(\mu\psi.[\![\lambda x.\,\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x{\times}\mathit{fact}(x-1)]\!]\, \rho[\psi/\mathit{fact}]))\,(\lfloor 2 \rfloor) \qquad (L1.2)$$

where $\psi \in [N_\perp \to N_\perp]_\perp$.

Denotational Semantics of the Lazy2 Language.

$$[\![n]\!]\,\rho \;=\; \lfloor n \rfloor$$
$$[\![x]\!]\,\rho \;=\; \rho(x)$$
$$[\![t_1 \,\mathbf{op}\, t_2]\!]\,\rho \;=\; [\![t_1]\!]\,\rho \;\; op_\perp \;\; [\![t_2]\!]\,\rho$$
$$[\![\mathbf{if}\, t_0 \,\mathbf{then}\, t_1 \,\mathbf{else}\, t_2]\!]\,\rho \;=\; \mathit{Cond}([\![t_0]\!]\,\rho, [\![t_1]\!]\,\rho, [\![t_2]\!]\,\rho)$$
$$[\![(t_1, t_2)]\!]\,\rho \;=\; ([\![t_1]\!]\,\rho,\; [\![t_2]\!]\,\rho)$$
$$[\![\mathbf{fst}(t)]\!]\,\rho \;=\; \pi_1([\![t]\!]\,\rho)$$
$$[\![\mathbf{snd}(t)]\!]\,\rho \;=\; \pi_2([\![t]\!]\,\rho)$$
$$[\![t_1 t_2]\!]\,\rho \;=\; ([\![t_1]\!]\rho)([\![t_2]\!]\rho)$$
$$[\![\lambda x.t]\!]\,\rho \;=\; \lambda v.[\![t]\!]\,\rho\,[v/x]$$
$$[\![\mathbf{rec}\, x.t]\!]\,\rho \;=\; \mathit{fix}(\lambda f.[\![t]\!]\,\rho[f/x])$$

TABLE 9. Denotational Semantics of the Lazy2 Language. For simplicity, we have written $[\![\_]\!]$, instead of $[\![\_]\!]^{\ell 2}$.

Now, in order to compute the subexpression $(\mathit{down}(\ldots))$, note that:

$$\mu\psi.[\![\lambda x.\,\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x \times \mathit{fact}(x-1)]\!]\,\rho[\psi/\mathit{fact}] \neq \perp \in [N_\perp \to N_\perp]_\perp.$$

Indeed, $\mu\psi.[\![\lambda x.\,\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x \times \mathit{fact}(x-1)]\!]\,\rho[\psi/\mathit{fact}] =$

$$= \mathit{fix}\left(\lambda\psi.[\![\lambda x.\,\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x \times \mathit{fact}(x-1)]\!]\,\rho[\psi/\mathit{fact}]\right) =$$
$$= \mathit{fix}\left(\lambda\psi.\lfloor\lambda v[\![\lambda x.\,\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x \times \mathit{fact}(x-1)]\!]\,\rho[\psi/\mathit{fact}, v/x]\rfloor\right) \qquad (\dagger)$$

where $\mathit{fix} \in \left[[N_\perp \to N_\perp]_\perp \to [N_\perp \to N_\perp]_\perp\right]$, $\psi \in [N_\perp \to N_\perp]_\perp$, $v \in N_\perp$, and the value of Expression $(\dagger)$ is different from $\perp \in [N_\perp \to N_\perp]_\perp$, because of the occurrence of the lifting function $\lfloor\_\rfloor$.

Now, similarly to what we did for the minimal fixpoint in Expression $(E2)$ on page 213, instead of computing the minimal fixpoint in Expression $(L1.2)$, we proceed by using the following equation (see page 99):

$$\mu x.t \;=\; (\lambda x.t)(\mu x.t) \qquad (\mathit{FixUnfold})$$

Thus, from $(L1.2)$ by applying $(\mathit{FixUnfold})$, we get:

$$\mathit{down}\big((\lambda\psi.[\![\lambda x.\,\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x \times \mathit{fact}(x-1)]\!]\rho[\psi/\mathit{fact}])$$
$$(\mu\psi.[\![\lambda x.\,\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x \times \mathit{fact}(x-1)]\!]\,\rho[\psi/\mathit{fact}])\big)\,(\lfloor 2\rfloor) =$$
$$= \mathit{down}\big([\![\lambda x.\,\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x \times \mathit{fact}(x-1)]\!]$$
$$\rho[(\mu\psi.[\![\lambda x.\,\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x \times \mathit{fact}(x-1)]\!]\,\rho[\psi/\mathit{fact}])/\mathit{fact}]\big)\,(\lfloor 2\rfloor) =$$
$$= \mathit{down}\big(\lfloor\lambda v.\,[\![\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x \times \mathit{fact}(x-1)]\!]$$
$$\rho[(\mu\psi.[\![\lambda x.\,\mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\ x \times \mathit{fact}(x-1)]\!]\,\rho[\psi/\mathit{fact}])/\mathit{fact}, v/x]\rfloor\big)\,(\lfloor 2\rfloor) = (\dagger\dagger)$$

$= [\![\textbf{if } x \textbf{ then } 1 \textbf{ else } x{\times}fact(x-1)]\!]$
$\quad \rho[(\mu\psi.[\![\lambda x.\, \textbf{if } x \textbf{ then } 1 \textbf{ else } x \times fact(x{-}1)]\!]\, \rho[\psi/fact])/fact,\ \lfloor 2 \rfloor/x] =$
$= Cond([\![x]\!]\rho[\ldots/fact,\ \lfloor 2 \rfloor/x],\ [\![1]\!]\rho[\ldots/fact,\ \lfloor 2 \rfloor/x],$
$\quad [\![x{\times}fact(x-1)]\!]\, \rho[\ldots/fact,\ \lfloor 2 \rfloor/x]) =$
$= Cond(\lfloor 2 \rfloor,\ \lfloor 1 \rfloor,$
$\quad \lfloor 2 \rfloor \times_\perp \big(let\, \varphi \Leftarrow (\mu\psi.[\![\lambda x.\, \textbf{if } x \textbf{ then } 1 \textbf{ else } x \times fact(x{-}1)]\!]\, \rho[\psi/fact])\,\textbf{.}$
$\qquad\qquad \varphi(\lfloor 2 \rfloor -_\perp \lfloor 1 \rfloor)\big) =$
$= \lfloor 2 \rfloor \times_\perp \big(let\, \varphi \Leftarrow (\mu\psi.[\![\lambda x.\, \textbf{if } x \textbf{ then } 1 \textbf{ else } x{\times}fact(x{-}1)]\!]\, \rho[\psi/fact])\,\textbf{.}\ \varphi(\lfloor 1 \rfloor)\big) =$
$= \lfloor 2 \rfloor \times_\perp (down(\mu\psi.[\![\lambda x.\, \textbf{if } x \textbf{ then } 1 \textbf{ else } x \times fact(x{-}1)]\!]\, \rho[\psi/fact]))\, (\lfloor 1 \rfloor)$ $(L1.1)$

Now if we compare this expression $(L1.1)$ and the expression $(L1.2)$ on page 215, we see that, by performing from $(L1.2)$ a sequence of evaluation steps analogous to the sequence which leads from $(L1.2)$ to $(L1.1)$, we eventually get:

$\lfloor 2 \rfloor \times_\perp \big(\lfloor 1 \rfloor \times_\perp (down(\mu\psi.[\![\lambda x.\, \textbf{if } x \textbf{ then } 1 \textbf{ else } x{\times}fact(x{-}1)]\!]\, \rho[\psi/fact]))\, (\lfloor 0 \rfloor)\big)$

Then, after a few more evaluation steps we get:

$\lfloor 2 \rfloor \times_\perp (\lfloor 1 \rfloor \times_\perp \lfloor 1 \rfloor) = \lfloor 2 \rfloor$, as expected.

3.6. Computing the Factorial in the Lazy2 Denotational Semantics.

Let us see in action the lazy2 denotational semantics and let us compute the value of $[\![(\textbf{rec } fact.\ \lambda x.\ \textbf{if } x \textbf{ then } 1 \textbf{ else } x{\times}fact(x-1))\,(2)]\!]$. We have:

$[\![(\textbf{rec } fact.\ \lambda x.\ \textbf{if } x \textbf{ then } 1 \textbf{ else } x{\times}fact(x-1))\,(2)]\!]\ \rho =$
$= ([\![\textbf{rec } fact.\ \lambda x.\ \textbf{if } x \textbf{ then } 1 \textbf{ else } x{\times}fact(x-1)]\!]\ \rho)\ ([\![2]\!]\ \rho) =$
$= (\mu\psi.[\![\lambda x.\, \textbf{if } x \textbf{ then } 1 \textbf{ else } x{\times}fact(x-1)]\!]\, \rho[\psi/fact])\, (\lfloor 2 \rfloor)$ $(L2.2)$

Now, similarly to what we did for the minimal fixpoint in Expression $(L1.2)$ on page 215, instead of computing the minimal fixpoint in Expression $(L2.2)$, we proceed by using the following equation (see page 99):

$\mu x.t = (\lambda x.t)(\mu x.t)$ $(FixUnfold)$

Thus, from $(L2.2)$ by applying $(FixUnfold)$, we get:

$(\lambda\psi.[\![\lambda x.\, \textbf{if } x \textbf{ then } 1 \textbf{ else } x{\times}fact(x-1)]\!]\ \rho[\psi/fact])$
$\quad (\mu\psi.[\![\lambda x.\, \textbf{if } x \textbf{ then } 1 \textbf{ else } x{\times}fact(x-1)]\!]\, \rho[\psi/fact])\, (\lfloor 2 \rfloor) =$
$= [\![\lambda x.\, \textbf{if } x \textbf{ then } 1 \textbf{ else } x{\times}fact(x-1)]\!]$
$\quad \rho[(\mu\psi.[\![\lambda x.\, \textbf{if } x \textbf{ then } 1 \textbf{ else } x{\times}fact(x-1)]\!]\, \rho[\psi/fact])/fact]\, (\lfloor 2 \rfloor) =$
$= \lambda v.\, [\![\textbf{if } x \textbf{ then } 1 \textbf{ else } x{\times}fact(x-1)]\!]$
$\quad \rho[(\mu\psi.[\![\lambda x.\, \textbf{if } x \textbf{ then } 1 \textbf{ else } x{\times}fact(x-1)]\!]\, \rho[\psi/fact])/fact,\ v/x]\, (\lfloor 2 \rfloor).$

Note that with respect to the evaluation performed in the lazy1 denotational semantics (see Expression (††) on the facing page), in the above expression there is no lifting. The evaluation proceeds as in the case of the lazy1 semantics and we then get:

$[\![\textbf{if } x \textbf{ then } 1 \textbf{ else } x{\times}fact(x-1)]\!]$
$\quad \rho[(\mu\psi.[\![\lambda x.\, \textbf{if } x \textbf{ then } 1 \textbf{ else } x{\times}fact(x-1)]\!]\, \rho[\psi/fact])/fact,\ \lfloor 2 \rfloor/x] =$
$= Cond([\![x]\!]\rho[\ldots/fact,\ \lfloor 2 \rfloor/x],\ [\![1]\!]\rho[\ldots/fact,\ \lfloor 2 \rfloor/x],$
$\quad [\![x{\times}fact(x-1)]\!]\, \rho[\ldots/fact,\ \lfloor 2 \rfloor/x]) =$

$$= \mathit{Cond}(\lfloor 2\rfloor,\ \lfloor 1\rfloor,$$
$$\lfloor 2\rfloor \times_\perp (\mu\psi.[\![\lambda x.\,\textbf{if } x \textbf{ then } 1 \textbf{ else } x \times \mathit{fact}(x-1)]\!]\,\rho[\psi/\mathit{fact}])(\lfloor 2\rfloor -_\perp \lfloor 1\rfloor)) =$$
$$= \lfloor 2\rfloor \times_\perp (\mu\psi.[\![\lambda x.\,\textbf{if } x \textbf{ then } 1 \textbf{ else } x \times \mathit{fact}(x-1)]\!]\,\rho[\psi/\mathit{fact}])(\lfloor 1\rfloor)) \qquad (L2.1)$$

Now if we compare the expressions $(L2.1)$ and $(L2.2)$, we can see that, by performing from $(L2.2)$ a sequence of evaluation steps analogous to the sequence which leads from $(L2.2)$ to $(L2.1)$, we eventually get:

$$\lfloor 2\rfloor \times_\perp (\lfloor 1\rfloor \times_\perp ((\mu\psi.[\![\lambda x.\,\textbf{if } x \textbf{ then } 1 \textbf{ else } x \times \mathit{fact}(x-1)]\!]\,\rho[\psi/\mathit{fact}])\,(\lfloor 0\rfloor)))$$

Then, after a few more evaluation steps we get:

$\lfloor 2\rfloor \times_\perp (\lfloor 1\rfloor \times_\perp \lfloor 1\rfloor) = \lfloor 2\rfloor$, as expected.

EXERCISE 3.6. Show that in the eager denotational semantics we have that the FixUnfold equality holds (see page 99), that is, for all variables x, for all terms t, for all environments ρ, $\mathit{fix}\,([\![\lambda x.t]\!]^e\rho) = ([\![\lambda x.t]\!]^e\rho)\,(\mathit{fix}\,([\![\lambda x.t]\!]^e\rho))$ (see page 213).

Show that the FixUnfold equality holds also for the lazy1 denotational semantics (see page 216) and the lazy2 denotational semantics (see page 217). □

3.7. Denotational Semantics of the let-in construct.

Let us assume that we have the syntactic construct which we have introduced in Section 2.6 on page 207:

$$\textbf{let } x \Leftarrow t_1 \textbf{ in } t$$

where x does not occur in t_1.

We want to have the equivalence of the denotational semantics of $\textbf{let } x \Leftarrow t_1 \textbf{ in } t$ and the denotational semantics of $((\lambda x.t)t_1)$. Thus, in the eager denotational semantics we stipulate that:

$$[\![\,\textbf{let } x \Leftarrow t_1 \textbf{ in } t\,]\!]\,\rho = [\![t]\!]\,\rho[\mathit{down}([\![t_1]\!]\rho)/x]$$

and in the lazy1 and lazy2 denotational semantics we stipulate that:

$$[\![\,\textbf{let } x \Leftarrow t_1 \textbf{ in } t\,]\!]\,\rho = [\![t]\!]\,\rho[[\![t_1]\!]\rho/x].$$

4. The Alpha Rule

The α-rule does not hold in the eager operational semantics.
The α-rule does not hold in the lazy operational semantics.

The proof of these properties is based on the fact that for both the eager operational semantics and the lazy operational semantics, $\lambda x.t$ and $\lambda y.t[y/x]$ are different canonical forms. As usual, $t[y/x]$ denotes the term t where each free occurrence of the variable x is replaced by y.

The α-rule holds in the eager denotational semantics, in the lazy1 denotational semantics, and in the lazy2 denotational semantics.

Let us consider the case of the eager denotational semantics. The cases of the lazy1 and the lazy2 denotational semantics are similar. We have that:

$[\![\lambda x.t]\!]\,\rho \;=\; \lfloor \lambda v.[\![t]\!]\,\rho[v/x] \rfloor$,

$[\![\lambda y.t[y/x]]\!]\,\rho \;=\; \lfloor \lambda v.[\![t[y/x]]\!]\,\rho[v/y] \rfloor$,

where y is a fresh, new variable distinct from x, and the right hand sides of these two equations are equal.

5. The Beta Rule

The β-rule does not hold in the eager operational semantics.

It is *not* the case that, for all terms t and e, for all canonical forms c:

$((\lambda x.t)\,e) \;\to\; c \quad \text{iff} \quad t[e/x] \;\to\; c$

Indeed, take $t =_{def} 1$ and $e =_{def} ((\mathbf{rec}\, y.\lambda x.(y\, x))\, 5)$. We have the following Points (i) and (ii).

Point (i). No canonical form c exists such that $(\lambda x.1)e \to c$, because no canonical form c_2 exists such that $e \to c_2$. Indeed, since e is an application, by the following rules of the eager operational semantics (see Table 3 on page 202)

$$\frac{t_1 \to \lambda x.t \qquad t_2 \to c_2 \qquad t[c_2/x] \to c}{(t_1\, t_2) \to c} \qquad (app)$$

$$\mathbf{rec}\, y.(\lambda x.t) \;\to\; \lambda x.(t[\mathbf{rec}\, y.(\lambda x.t)/y]) \qquad (rec)$$

we have that:

there exists a canonical form c such that $((\mathbf{rec}\, y.\lambda \widetilde{x}.(y\, \widetilde{x}))\, 5) \to c$ iff {by (rec)}

there exists a canonical form c such that $\lambda x.((\mathbf{rec}\, y.\lambda \widetilde{x}.(y\, \widetilde{x}))\, x)\, 5 \to c$ iff {by (app)}

there exists a canonical form c such that $((\mathbf{rec}\, y.\lambda \widetilde{x}.(y\, \widetilde{x}))\, 5) \to c$

Thus, there is no proof that there exists a canonical form c such that:

$((\mathbf{rec}\, y.\lambda \widetilde{x}.(y\, \widetilde{x}))\, 5) \to c.$

Point (ii). We have that $t[e/x] \to 1[e/x] \to 1$ and 1 is a canonical form.

The β-rule holds in the lazy operational semantics.

It is the case that, for all terms t and e, for all canonical forms c:

$((\lambda x.t)\,e) \;\to\; c \quad \text{iff} \quad t[e/x] \;\to\; c$

because we have the following derived rule for the lazy operational semantics (see Table 4 on page 204) when t_1 is $\lambda x.t$ and t_2 is e:

$$\frac{t[e/x] \to c}{((\lambda x.t)\, e) \to c}$$

The β-rule does not hold in the eager denotational semantics.

We have that, for all terms t and e, for all environments ρ:

$[\![(\lambda x.t)\, e]\!]\, \rho \neq [\![t[e/x]]\!]\, \rho$

where $t[e/x]$ denotes the term t where all free occurrences of x which are bound in $\lambda x.t$, have been replaced by e. Indeed,

$[\![(\lambda x.t)e]\!]\rho =$

$= \mathit{let}\ \varphi \Leftarrow \lfloor \lambda\widetilde{v}.[\![t]\!]\rho[\widetilde{v}/x]\rfloor,\ v \Leftarrow [\![e]\!]\rho \,.\, \varphi(v) =$

$= \mathit{let}\ v \Leftarrow [\![e]\!]\rho \,.\, (\lambda\widetilde{v}.[\![t]\!]\rho[\widetilde{v}/x])(v) =$

$= \mathit{let}\ v \Leftarrow [\![e]\!]\rho \,.\, [\![t]\!]\rho[v/x]$

which may be different from $[\![t[e/x]]\!]\, \rho$ if $[\![e]\!]\rho = \bot$.

Indeed, take $t = 1$ and $e = ((\mathbf{rec}\, y.\lambda x.(y\, x))\, 5)$ with $y :\ \mathit{int} \to \mathit{int}$ and $x : \mathit{int}$. We have that:

$$[\![(\lambda x.1)\, ((\mathbf{rec}\, y.\lambda x.(y\, x))\, 5)]\!]\, \rho \ \neq\ [\![1[(\mathbf{rec}\, y.\lambda x.(y\, x))\, 5/x]]\!]\, \rho. \qquad (\dagger)$$

Left hand side of Equation (†) =

$= \mathit{let}\ v \Leftarrow [\![(\mathbf{rec}\, y.\lambda x.(y\, x))\, 5]\!]\rho \,.\, [\![1]\!]\rho[v/x] =$

$= \mathit{let}\ v \Leftarrow (\mathit{let}\ \varphi \Leftarrow [\![\mathbf{rec}\, y.\lambda x.(y\, x)]\!]\rho,\ \widetilde{v} \Leftarrow [\![5]\!]\rho \,.\, \varphi(\widetilde{v})) \,.\, [\![1]\!]\rho[v/x] =$

{by Remark 5.1 below}

$= \mathit{let}\ v \Leftarrow (\lambda x.\bot)\, 5 \,.\, [\![1]\!]\rho[v/x] =$

{by $(\lambda x.\bot)\, 5 = \bot$ and by the definition of the *let* construct}

$= \bot \in N_\bot.$

Right hand side of Equation (†) =

$= [\![1[(\mathbf{rec}\, y.\lambda x.(y\, x))\, 5/x]]\!]\, \rho = [\![1]\!]\, \rho = \lfloor 1 \rfloor.$

REMARK 5.1. Here we prove that $[\![\mathbf{rec}\, y.\lambda x.(y\, x)]\!]\rho = \lfloor \lambda x.\bot \rfloor \in [N \to N_\bot]_\bot$.

We have that: $[\![\mathbf{rec}\, y.\lambda x.(yx)]\!]\rho \in [N \to N_\bot]_\bot$. We also have that:

$[\![\mathbf{rec}\, y.\lambda x(yx)]\!]\rho =$

$= \lfloor \mu\varphi.\lambda v.\ \mathit{let}\ y' \Leftarrow [\![y]\!]\rho[v/x,\, \varphi/y],\ x' \Leftarrow [\![x]\!]\rho[v/x,\, \varphi/y] \,.\, y'(x') \rfloor =$

$= \lfloor \mu\varphi.\lambda v.\varphi(v) \rfloor\ = \lfloor \mathit{fix}(\lambda\varphi.\lambda v.\varphi(v)) \rfloor =$ {see Equation (††) below} $=$

$= \lfloor \lambda x.\bot \rfloor \in [N \to N_\bot]_\bot.$

Let us explain this last step by proving Equation (††).

For $\varphi \in [N \to N_\bot]_\bot$, for $v \in N$, we have that $\mathit{fix}(\lambda\varphi.\lambda v.\varphi(v)) = \lambda x.\bot$, with $\lambda x.\bot \in [N \to N_\bot]$, because we have that:

$\bigsqcup_{n\geq 0}(\lambda\varphi.\lambda v.\, \varphi(v))^n(\bot) = \bigsqcup_{n\geq 0} \tau^n(\bot)$ for $\tau = \lambda\varphi.\lambda v.\varphi(v)$ with $\bot \in [N \to N_\bot]_\bot$.

$\tau^0(\bot) = \lambda x.\bot \in [N \to N_\bot]$ and

$\tau^1(\bot) = \lambda v.((\lambda x.\bot)(v))$ which is $\lambda v.\bot \in [N \to N_\bot]$,

where $\bot$ in the arguments of τ^0 and τ^1 on the left hand sides belongs to $[N \to N_\bot]_\bot$, and $\bot$ on the right hand sides belongs to $N_\bot$.

Thus, $\bigsqcup_{n\geq 0}(\lambda\varphi.\lambda v.\, \varphi(v))^n(\bot) = \lambda v.\bot \in [N \to N_\bot]$. (††) □

The β-rule holds in the lazy1 denotational semantics.

We have that, for all terms t and e, for all environments ρ:
$[\![(\lambda x.t)\, e]\!]\, \rho =$
$= \mathit{let}\ \varphi \Leftarrow \lfloor \lambda d.[\![t]\!]\rho[d/x] \rfloor \centerdot \varphi([\![e]\!]\rho) =$
$= (\lambda d.[\![t]\!]\rho[d/x]\, ([\![e]\!]\rho) =$
$= [\![t]\!]\rho[[\![e]\!]\rho/x].$

The β-rule holds in the lazy2 denotational semantics.

We have that, for all terms t and e, for all environments ρ:
$[\![(\lambda x.t)\, e]\!]\, \rho = (\lambda d.[\![t]\!]\rho[d/x])\, ([\![e]\!]\rho) = [\![t]\!]\rho[[\![e]\!]\rho/x].$

6. The Eta Rule

The η-rule does not hold in the eager operational semantics.

Let us consider the term $\lambda x.(f x)$, where x is not free in f. We have that this term is a canonical form and it may be the case that there is no canonical form c such that $f \to c$.
Consider, for instance, $f =_{def} ((\mathbf{rec}\, y.\lambda x.(y\, x))\, 5)$. Indeed,

$f \to c$ iff $\lambda x.((\mathbf{rec}\, y.\lambda \widetilde{x}.(y\, \widetilde{x}))\, x)\, 5 \to c$ iff $((\mathbf{rec}\, y.\lambda \widetilde{x}.(y\, \widetilde{x}))\, 5) \to c$

and thus, there is no proof that there exists a canonical form c such that $f \to c$.

The η-rule does not hold in the lazy operational semantics.

The proof is as in the case of the eager operational semantics, but now we have to consider the term $f =_{def} \mathbf{rec}\, w.w$, instead of the term $f =_{def} ((\mathbf{rec}\, y.\lambda x.(y\, x))\, 5)$.

The η-rule does not hold in the eager denotational semantics.

Let us consider the following terms: (i) $x : \sigma$, (ii) $f : \sigma \to \tau$, and (iii) $\lambda x.(f x) : \sigma \to \tau$, where x is not free in f.
We have that:

$$[\![f]\!]\rho,\ [\![\lambda x.(f x)]\!]\rho\ \in [V_\sigma \to (V_\tau)_\perp]_\perp, \quad \rho(x) \in V_\sigma, \quad \text{and} \quad [\![x]\!]\rho \in (V_\sigma)_\perp.$$

Let us assume that $[\![f]\!]\rho = \perp$, with $\perp \in [V_\sigma \to (V_\tau)_\perp]_\perp$. We have that:
$[\![\lambda x.(f x)]\!]\, \rho =$
$= \lfloor \lambda w.\ \mathit{let}\ \varphi \Leftarrow [\![f]\!]\rho[w/x],\ v \Leftarrow [\![x]\!]\rho[w/x] \centerdot \varphi(v) \rfloor =$
$= \lfloor \lambda w.\ \mathit{let}\ \varphi \Leftarrow [\![f]\!]\rho,\ v \Leftarrow \lfloor w \rfloor \centerdot \varphi(v) \rfloor =$
$= \lfloor \lambda w.\ \mathit{let}\ \varphi \Leftarrow [\![f]\!]\rho \centerdot \varphi(w) \rfloor \quad (\text{where } w \in V_\sigma \text{ and } \varphi \in [V_\sigma \to (V_\tau)_\perp]) =$ (1)
$= \{\text{by } [\![f]\!]\rho = \perp\} =$
$= \lfloor \lambda w.\perp \rfloor$
which is different from $\perp \in [V_\sigma \to (V_\tau)_\perp]_\perp$.

Note that if we assume that $[\![f]\!]\rho \neq \perp$ with $\perp \in [V_\sigma \to (V_\tau)_\perp]_\perp$ then the η-rule holds in the eager denotational semantics. Indeed,

$[\![\lambda x.(fx)]\!]\,\rho = \{\text{see (1) above}\} =$
$= \lfloor \lambda w.\ let\ \varphi \Leftarrow [\![f]\!]\rho \centerdot \varphi(w) \rfloor = \{\text{evaluating the } let \text{ expression}\} =$
$= \lfloor \lambda w.(down([\![f]\!]\rho)\, w) \rfloor = \{\text{the } \eta\text{-rule holds in mathematics}\} =$
$= \lfloor down([\![f]\!]\rho) \rfloor = \{\text{by } [\![f]\!]\rho \neq \bot\} =$
$= [\![f]\!]\rho.$

The η-rule does not hold in the lazy1 denotational semantics.

Let us consider the following terms: (i) $x : \sigma$, (ii) $f : \sigma \to \tau$, and (iii) $\lambda x.(fx) : \sigma \to \tau$, where x is not free in f.
We have that:

$$[\![f]\!]\rho,\ [\![\lambda x.(fx)]\!]\rho \in [(V_\sigma)_\perp \to (V_\tau)_\perp]_\perp, \quad \rho(x) \in V_\sigma, \quad \text{and} \quad [\![x]\!]\rho \in (V_\sigma)_\perp.$$

Let us assume that $[\![f]\!]\rho = \bot$, with $\bot \in [(V_\sigma)_\perp \to (V_\tau)_\perp]_\perp$. We have that:
$[\![\lambda x.(fx)]\!]\,\rho =$
$= \lfloor \lambda d.\ [\![fx]\!]\rho[d/x] \rfloor \quad (\text{where } d \in (V_\sigma)_\perp \text{ and } \varphi \in [V_\sigma \to (V_\tau)_\perp]) =$
$= \lfloor \lambda d.\ let\ \varphi \Leftarrow [\![f]\!]\rho[d/x] \centerdot \varphi([\![x]\!]\rho[d/x]) \rfloor \quad (\text{where } \varphi \in [(V_\sigma)_\perp \to (V_\tau)_\perp]) =$
$= \lfloor \lambda d.\ let\ \varphi \Leftarrow [\![f]\!]\rho \centerdot \varphi(d) \rfloor =$ (2)
$= \lfloor \lambda d.\bot \rfloor, \text{ with } \lfloor \lambda d.\bot \rfloor \in [(V_\sigma)_\perp \to (V_\tau)_\perp]_\perp$
which is different from $\bot$ of $[(V_\sigma)_\perp \to (V_\tau)_\perp]_\perp$.

Note that if we assume that $[\![f]\!]\rho \neq \bot$ with $\bot \in [(V_\sigma)_\perp \to (V_\tau)_\perp]_\perp$ then the η-rule holds in the lazy1 denotational semantics. Indeed,
$[\![\lambda x.(fx)]\!]\,\rho = \{\text{see (2) above}\} =$
$= \lfloor \lambda d.\ let\ \varphi \Leftarrow [\![f]\!]\rho \centerdot \varphi(d) \rfloor = \{\text{evaluating the } let \text{ expression}\} =$
$= \lfloor \lambda d.(down([\![f]\!]\rho)\, d) \rfloor = \{\text{the } \eta\text{-rule holds in mathematics}\} =$
$= \lfloor down([\![f]\!]\rho) \rfloor = \{\text{because } [\![f]\!]\rho \neq \bot\} =$
$= [\![f]\!]\rho.$

The η-rule holds in the lazy2 denotational semantics.

Let us consider the following terms: (i) $x : \sigma$, (ii) $f : \sigma \to \tau$, and (iii) $\lambda x.(fx) : \sigma \to \tau$, where x is not free in f.
We have that:

$$[\![f]\!]\rho,\ [\![\lambda x.(fx)]\!]\rho \in [V_\sigma \to V_\tau], \quad \rho(x) \in V_\sigma, \quad \text{and} \quad [\![x]\!]\rho \in V_\sigma.$$

We also have that:
$[\![\lambda x.(fx)]\!]\,\rho =$
$= \lambda d.([\![fx]\!]\rho[d/x]) =$
$= \lambda d.(([\![f]\!]\rho[d/x])\,([\![x]\!]\rho[d/x])) =$
$= \lambda d.(([\![f]\!]\rho)\, d) = \{\text{the } \eta\text{-rule holds in mathematics}\} =$
$= [\![f]\!]\rho.$

7. The Fixpoint Operators

In this section we study various fixpoint operators in higher order, typed functional languages. We also show that in those languages we can dispose of the **rec** construct in favour of lambda abstraction and function application.

Indeed, as indicated by Equation (E) on this page and Equation (L) on the current page, we have that any term t *with* **rec** constructs can be replaced by a term t' *without* **rec** constructs, such that t and t' have the same denotational semantics.

In the Eager language, given the variable $x:\tau_1$, the term $t:\tau_2$, the term $\lambda x.t:\tau$, and the variable $y:\tau$, with $\tau = \tau_1 \to \tau_2$, there exists a term $R:(\tau\to\tau)\to\tau$ such that for any environment ρ:

$$[\![R\,(\lambda y.\lambda x.t)]\!]\rho = [\![\mathbf{rec}\,y.(\lambda x.t)]\!]\rho \tag{E}$$

We will show that in the Eager language the term R can be taken to be the term $\mathbf{rec}\,w.\lambda f.\lambda x.((f(wf))\,x)$ where $w:(\tau\to\tau)\to\tau$, $f:\tau\to\tau$, and $x:\tau_1$, with $\tau=\tau_1\to\tau_2$. Since in R there are no free variables, in Equation (E) the value of the environment ρ is irrelevant. Other choices for R are possible.

Note that Equation (E) does *not* hold operationally, because as the reader may verify, it is not the case that for all canonical forms c, we have that $R\,(\lambda y.\lambda x.t) \to^e c$ iff $\mathbf{rec}\,y.(\lambda x.t) \to^e c$.

In the Lazy1 language and the Lazy2 language, given the terms $y:\tau$, and $t:\tau$, there exists a term $R:(\tau\to\tau)\to\tau$ such that for any environment ρ:

$$[\![R\,(\lambda y.t)]\!]\rho = [\![\mathbf{rec}\,y.t]\!]\rho \tag{L}$$

We will show that in the Lazy1 language and the Lazy2 language the term R can be taken to be $\mathbf{rec}\,w.\lambda f.(f(wf))$ where $w:(\tau\to\tau)\to\tau$ and $f:\tau\to\tau$. Since in R there are no free variables, in Equation (L) the value of the environment ρ is irrelevant. Also in this case other choices for R are possible.

Note that Equation (L) does *not* hold operationally, because as the reader may verify, it is not the case that for all canonical forms c, we have that $R\,(\lambda y.t) \to^\ell c$ iff $\mathbf{rec}\,y.t \to^\ell c$.

In the Eager language we have that:

$$\begin{aligned}[\![R]\!]\rho = {} & \lfloor \lambda\varphi.\, \lfloor \mathit{fix}(\mathit{down}\circ\varphi)\rfloor\rfloor \in [[V_\tau \to (V_\tau)_\perp] \to (V_\tau)_\perp]_\perp \\ & \text{with } \varphi \in [V_\tau \to (V_\tau)_\perp],\quad \mathit{fix} \in [[V_\tau \to V_\tau] \to V_\tau], \text{ and} \\ & \mathit{down} \in [(V_\tau)_\perp \to V_\tau].\end{aligned} \tag{RE}$$

In the Lazy1 language we have that:

$$\begin{aligned}[\![R]\!]\rho = {} & \lfloor \lambda\varphi.\, \mathit{fix}(\mathit{down}\,\varphi)\rfloor \in [[(V_\tau)_\perp \to (V_\tau)_\perp]_\perp \to (V_\tau)_\perp]_\perp \\ & \text{with } \varphi \in [(V_\tau)_\perp \to (V_\tau)_\perp]_\perp,\ \mathit{fix} \in [[(V_\tau)_\perp \to (V_\tau)_\perp] \to (V_\tau)_\perp], \text{ and} \\ & \mathit{down} \in [[(V_\tau)_\perp \to (V_\tau)_\perp]_\perp \to [(V_\tau)_\perp \to (V_\tau)_\perp]].\end{aligned} \tag{RL1}$$

In the Lazy2 language we have that:

$$[\![R]\!]\rho = \mathit{fix} \in [[V_\tau \to V_\tau] \to V_\tau] \tag{RL2}$$

In the case of the Lazy2 language we assume that the cpo V_τ is a cpo with a bottom element. This hypothesis is required, in particular, in the proof of (RL2) (see page 226), where we need the bottom element of the cpo $[[V_\tau \to V_\tau] \to V_\tau]$.

The terms R are called *fixpoint operators* because for any given term $F : \tau \to \tau$, for the Eager language we have that:

$$[\![RF]\!]\rho = [\![F(RF)]\!]\rho \in (V_\tau)_\perp \quad \text{if } \mathit{down}([\![F]\!]\rho) \neq (\lambda x.\perp) \in [V_\tau \to (V_\tau)_\perp] \tag{EF}$$

for the Lazy1 language we have that:

$$[\![RF]\!]\rho = [\![F(RF)]\!]\rho \in (V_\tau)_\perp \tag{LF1}$$

and for the Lazy2 language we have that:

$$[\![RF]\!]\rho = [\![F(RF)]\!]\rho \in V_\tau \tag{LF2}$$

In (EF) the condition $\mathit{down}([\![F]\!]\rho) \neq (\lambda x.\perp) \in [V_\tau \to (V_\tau)_\perp]$ is equivalent to the conjunction of the following two conditions:
(i) $[\![F]\!]\rho \neq \perp \in [V_\tau \to (V_\tau)_\perp]_\perp$ where $\perp$ is the bottom element in $[V_\tau \to (V_\tau)_\perp]_\perp$, and
(ii) $[\![F]\!]\rho \neq (\lambda x.\perp) \in [V_\tau \to (V_\tau)_\perp]$ where $x \in V_\tau$ and $\perp$ is the bottom element in $(V_\tau)_\perp$.
First we show (RE), (RL1), and (RL2), then we show (E) and (L) and, finally, we show (EF), (LF1), and (LF2).

Proof of (RE). We have to show that $[\![R]\!]\rho = \lfloor \lambda\varphi.\, \lfloor \mathit{fix}(\mathit{down} \circ \varphi) \rfloor \rfloor$.
We have that $[\![R]\!]\rho \in [[V_\tau \to (V_\tau)_\perp] \to (V_\tau)_\perp]_\perp$.
Since R is $\mathbf{rec}\, w.\lambda f.\lambda x.((f(wf))\, x)$, we have that:
$[\![R]\!]\rho = \lfloor \mathit{fix}\, (\lambda u.\lambda\varphi.\lfloor \mathit{down}\, (\mathit{let}\, v \Leftarrow u(\varphi) \centerdot (\varphi\, v)) \rfloor) \rfloor$
with $\varphi \in [V_\tau \to (V_\tau)_\perp]$, $u \in A$, and $\mathit{fix} \in [[A \to A] \to A]$,
where A stands for $[[V_\tau \to (V_\tau)_\perp] \to (V_\tau)_\perp]$.
Thus,
$[\![R]\!]\rho = \lfloor \bigsqcup_{n\in\omega} \chi^n(\perp) \rfloor$ with $\chi = \lambda u.\lambda\varphi.\lfloor \mathit{down}\, (\mathit{let}\, v \Leftarrow u(\varphi) \centerdot (\varphi\, v)) \rfloor$ and
$\chi \in [[[V_\tau \to (V_\tau)_\perp] \to (V_\tau)_\perp] \to [[V_\tau \to (V_\tau)_\perp] \to (V_\tau)_\perp]]$.
We have that:

$$\begin{aligned}
\chi^0(\perp) &= \perp \qquad \text{with both } \perp\text{'s belonging to } [[V_\tau \to (V_\tau)_\perp] \to (V_\tau)_\perp] \\
&= \lambda\varphi.\perp \quad \text{with } \perp \in (V_\tau)_\perp \\
\chi^1(\perp) &= \lambda\varphi.\lfloor \mathit{down}\, (\mathit{let}\, v \Leftarrow \perp \centerdot (\varphi\, v)) \rfloor = \\
&= \lambda\varphi.\lfloor \mathit{down}\, (\varphi\perp) \rfloor = \\
&= \lambda\varphi.\lfloor (\mathit{down} \circ \varphi)\perp \rfloor
\end{aligned}$$

where on the left hand side $\bot \in [[V_\tau \to (V_\tau)_\bot] \to (V_\tau)_\bot]$ and on the right hand sides $\bot \in (V_\tau)_\bot$.

$$\begin{aligned}\chi^2(\bot) &= \lambda\varphi.\lfloor down\,(let\, v \Leftarrow \lambda\widetilde{\varphi}.\lfloor(down \circ \widetilde{\varphi})\bot\rfloor(\varphi)\,.\,(\varphi\, v))\rfloor = \\ &= \lambda\varphi.\lfloor down\,(let\, v \Leftarrow \lfloor(down \circ \varphi)\bot\rfloor\,.\,(\varphi\, v))\rfloor = \\ &= \lambda\varphi.\lfloor down\,(\varphi((down \circ \varphi)\bot))\rfloor = \\ &= \lambda\varphi.\lfloor(down \circ \varphi)((down \circ \varphi)\bot)\rfloor = \\ &= \lambda\varphi.\lfloor(down \circ \varphi)^2(\bot)\rfloor\end{aligned}$$

where on the left hand side $\bot \in [[V_\tau \to (V_\tau)_\bot] \to (V_\tau)_\bot]$ and on the right hand sides $\bot \in (V_\tau)_\bot$,

and, by induction, one can show that for all $n \geq 0$, we have that:

$$\chi^n(\bot) = \lambda\varphi.\lfloor(down \circ \varphi)^n\bot)\rfloor$$

where on the left hand side $\bot \in [[V_\tau \to (V_\tau)_\bot] \to (V_\tau)_\bot]$ and on the right hand side $\bot \in (V_\tau)_\bot$. Thus,

$$\begin{aligned}&[\![R]\!]\rho = \lfloor \bigsqcup_{n\in\omega} \chi^n(\bot)\rfloor = \\ &= \lfloor \bigsqcup_{n\in\omega} \lambda\varphi.\lfloor((down \circ \varphi)^n\bot)\rfloor\rfloor = \{\text{lub's of functions are computed pointwise}\} = \\ &= \lfloor \lambda\varphi.\bigsqcup_{n\in\omega} \lfloor((down \circ \varphi)^n\bot)\rfloor\rfloor = \{\text{by continuity of } \lfloor\_\rfloor\} = \\ &= \lfloor \lambda\varphi.\lfloor \bigsqcup_{n\in\omega} (down \circ \varphi)^n\bot)\rfloor\rfloor = \{\text{by definition of } fix = \lambda f.\ \bigsqcup_{n\in\omega}\ f^n(\bot)\} = \\ &= \lfloor \lambda\varphi.\lfloor fix\,(down \circ \varphi)\rfloor\rfloor.\end{aligned}$$

□

Proof of (RL1). We have to show that $[\![R]\!]\rho = \lfloor\lambda\varphi.\, fix(down\,\varphi)\rfloor$.

We have that $[\![R]\!]\rho \in [[(V_\tau)_\bot \to (V_\tau)_\bot]_\bot \to (V_\tau)_\bot]_\bot$.

Since R is $\mathbf{rec}\, w.\lambda f.(f(wf))$, we have that:

$$[\![R]\!]\rho = fix(\lambda u.\lfloor\lambda\varphi.(down\,(\varphi))((down(u))\varphi)\rfloor)$$

with $\varphi \in [(V_\tau)_\bot \to (V_\tau)_\bot]_\bot$ and $u \in [[(V_\tau)_\bot \to (V_\tau)_\bot]_\bot \to (V_\tau)_\bot]_\bot$.

(Note that the two *down* operators are different because they have different types.)

Thus,

$[\![R]\!]\rho = \bigsqcup_{n\in\omega} \chi^n(\bot)$ with $\chi = \lambda u.\lfloor\lambda\varphi.(down\,(\varphi))((down(u))\varphi)\rfloor$ and

$\chi \in [[[(V_\tau)_\bot \to (V_\tau)_\bot]_\bot \to (V_\tau)_\bot]_\bot \ \to\ [[(V_\tau)_\bot \to (V_\tau)_\bot]_\bot \to (V_\tau)_\bot]_\bot]$.

We have that:

$$\begin{aligned}\chi^0(\bot) &= \bot \quad \text{with both } \bot\text{'s belonging to } [[(V_\tau)_\bot \to (V_\tau)_\bot]_\bot \to (V_\tau)_\bot]_\bot \\ \chi^1(\bot) &= \lfloor\lambda\varphi.(down\,(\varphi)\bot)\rfloor\end{aligned}$$

where on the left hand side $\bot \in [[(V_\tau)_\bot \to (V_\tau)_\bot]_\bot \to (V_\tau)_\bot]_\bot$ and on the right hand sides $\bot \in (V_\tau)_\bot$.

$$\begin{aligned}\chi^2(\bot) &= \lfloor\lambda\varphi.(down\,(\varphi))((down(\lfloor\lambda\widetilde{\varphi}.(down\,(\widetilde{\varphi})\bot)\rfloor))\varphi)\rfloor = \\ &= \lfloor\lambda\varphi.(down\,(\varphi))((\lambda\widetilde{\varphi}.(down\,(\widetilde{\varphi})\bot))\varphi)\rfloor = \\ &= \lfloor\lambda\varphi.(down\,(\varphi))((down\,(\varphi)\bot))\rfloor = \\ &= \lfloor\lambda\varphi.(down\,(\varphi))^2(\bot)\rfloor\end{aligned}$$

where on the left hand side $\bot \in [[(V_\tau)_\bot \to (V_\tau)_\bot]_\bot \to (V_\tau)_\bot]_\bot$ and on the right hand sides $\bot \in (V_\tau)_\bot$. By induction on n, one can show that for all $n \geq 0$,

$$\chi^n(\bot) = \lfloor\lambda\varphi.(down(\varphi))^n(\bot)\rfloor$$

where on the left hand side $\bot \in [[(V_\tau)_\bot \to (V_\tau)_\bot]_\bot \to (V_\tau)_\bot]_\bot$ and on the right hand side $\bot \in (V_\tau)_\bot$. Thus,

$$\begin{aligned}&[\![R]\!]\rho = \textstyle\bigsqcup_{n\in\omega}\chi^n(\bot) = \\ &= \textstyle\bigsqcup_{n\in\omega}\lfloor\lambda\varphi.(down(\varphi))^n(\bot)\rfloor = \{\text{by continuity of } \lfloor\_\rfloor\} = \\ &= \lfloor\textstyle\bigsqcup_{n\in\omega}\lambda\varphi.(down(\varphi))^n(\bot)\rfloor = \{\text{lub's of functions are computed pointwise}\} = \\ &= \lfloor\lambda\varphi.\textstyle\bigsqcup_{n\in\omega}(down(\varphi))^n(\bot)\rfloor = \{\text{by definition of } \mathit{fix} = \lambda f.\textstyle\bigsqcup_{n\in\omega} f^n(\bot)\} = \\ &= \lfloor\lambda\varphi.\mathit{fix}(down(\varphi))\rfloor.\end{aligned}$$

□

Proof of (RL2). We have to show that $[\![R]\!]\rho = \mathit{fix}$, where $R:(\tau\to\tau)\to\tau$.

We have that $[\![R]\!]\rho \in [[V_\tau \to V_\tau] \to V_\tau]$.

We also have that R is $\mathbf{rec}\,w.\lambda f.(f(wf))$ where $f:\tau\to\tau$. Thus,

$$\begin{aligned}&[\![R]\!]\rho = [\![\mathbf{rec}\,w.\lambda f.(f(wf))]\!]\rho = \\ &\quad = \mathit{fix}\,(\lambda\widetilde{w}.[\![\lambda f.(f(wf))]\!]\rho[\widetilde{w}/w] = \\ &\quad = \mathit{fix}\,(\lambda\widetilde{w}.\lambda\widetilde{f}.[\![f(wf)]\!]\rho[\widetilde{w}/w,\ \widetilde{f}/f] = \{\text{by renaming of bound variables}\} = \\ &\quad = \mathit{fix}\,(\lambda w.\lambda f.f(wf)).\end{aligned}$$

Hence,

$$\begin{aligned}&[\![R]\!]\rho = \textstyle\bigsqcup_{n\in\omega}\chi^n(\bot) \text{ with } \chi = \lambda w.\lambda f.f(wf) \text{ and} \\ &\quad \chi \in [[[V_\tau \to V_\tau] \to V_\tau] \ \to\ [[V_\tau \to V_\tau] \to V_\tau]].\end{aligned}$$

We have that:

$$\begin{aligned}\chi^0(\bot) &= \bot \qquad \text{with both } \bot\text{'s belonging to } [[V_\tau \to V_\tau] \to V_\tau] \\ \chi^1(\bot) &= (\lambda w.\lambda f.f(wf))\,(\bot) = \lambda f.\bot \\ \chi^2(\bot) &= (\lambda w.\lambda f.f(wf))\,(\lambda f.\bot) = \lambda f.f(\bot) \\ \chi^3(\bot) &= (\lambda w.\lambda f.f(wf))\,(\lambda f.f(\bot)) = \lambda f.f(f(\bot))\end{aligned}$$

where all $\bot$'s belong to $\bot \in [[V_\tau \to V_\tau] \to V_\tau]$, except those which occur in expressions of the form $\lambda f.f^n(\bot)$, for some $n \geq 0$: those $\bot$'s belong to V_τ (recall that: (i) $f \in [V_\tau \to V_\tau]$), and (ii) we have assumed that V_τ is a cpo with bottom) By induction on n, one can show that for all $n \geq 1$, $\chi^n(\bot) = \lambda f.f^{n-1}(\bot)$. Thus, since $\chi^0(\bot) = \bot$, we have that:

$$[\![R]\!]\rho = \textstyle\bigsqcup_{n\in\omega}\chi^n(\bot) = \bigsqcup_{n\in\omega}\lambda f.f^n(\bot) = \mathit{fix}.$$

□

Proof of (E). For the Eager language we have to show that:

$$[\![R\ (\lambda y.\lambda x.t)]\!]\rho = [\![\mathbf{rec}\, y.\lambda x.t]\!]\rho \tag{E}$$

with $R = \mathbf{rec}\, w.\lambda f.\lambda x.((f(wf))x)$ and $R : (\tau \to \tau) \to \tau$. By (RE) on page 223 and the definition of the eager denotational semantics of $\mathbf{rec}\, y.\lambda x.t$, we have to show that

$$let\ \widetilde{\varphi} \Leftarrow \lfloor \lambda\varphi.\ \lfloor fix(down \circ \varphi)\rfloor\rfloor,\ v \Leftarrow [\![\lambda y.\lambda x.t]\!]\rho \centerdot \widetilde{\varphi}(v) = \lfloor fix\ (\lambda u.\lambda v.[\![\, t\,]\!]\, \rho[u/y,\, v/x])\rfloor,$$

that is,

$$let\ v \Leftarrow \lfloor \lambda u.[\![\lambda x.t]\!]\rho[u/y]\rfloor \centerdot (\lambda\varphi.\ \lfloor fix(down \circ \varphi)\rfloor)(v) = \lfloor fix\ (\lambda u.\lambda v.[\![\, t\,]\!]\, \rho[u/y,\, v/x])\rfloor.$$

Now, this last equality holds because:

$$\begin{aligned}
&let\ v \Leftarrow \lfloor \lambda u.[\![\lambda x.t]\!]\rho[u/y]\rfloor \centerdot (\lambda\varphi.\ \lfloor fix(down \circ \varphi)\rfloor)(v) = \\
&\quad = \lambda\varphi.\lfloor\, fix(down \circ \varphi)\rfloor\, (\lambda u.[\![\lambda x.t]\!]\rho[u/y]) = \\
&\quad = \lfloor\, fix(down \circ (\lambda u.[\![\lambda x.t]\!]\rho[u/y]))\rfloor = \\
&\quad = \lfloor\, fix(down \circ (\lambda u.\lfloor \lambda v.[\![t]\!]\rho[u/y,\, v/x]\rfloor))\rfloor = \{\text{see Equation } (\dagger) \text{ below}\} = \\
&\quad = \lfloor\, fix\ (\lambda u.\lambda v.[\![\, t\,]\!]\, \rho[u/y,\, v/x])\rfloor.
\end{aligned}$$

Thus, the proof is completed if the following equation holds between elements of the cpo $[V_\tau \to V_\tau]$:

$$down \circ (\lambda u.\lfloor \lambda v.[\![t]\!]\rho[u/y,\, v/x]\rfloor) = \lambda u.\lambda v.[\![\, t\,]\!]\, \rho[u/y,\, v/x] \tag{$\dagger$}$$

where $u : \tau$, $v : \tau_1$, and $t : \tau_2$, with $\tau = \tau_1 \to \tau_2$. Indeed, for all r and s, we have that:

$$\begin{aligned}
(down \circ (\lambda u.\lfloor r\rfloor))s &= down((\lambda u.\lfloor r\rfloor)s) = down\lfloor r[s/u]\rfloor = r[s/u] = \\
&= (\lambda u.r)s
\end{aligned} \tag{$\dagger\dagger$}$$

Now, having derived the Equation ($\dagger\dagger$): $(down \circ (\lambda u.\lfloor r\rfloor))s = (\lambda u.r)s$, we may instantiate it by considering r to be the term $\lambda v.[\![t]\!]\rho[u/y,\, v/x]$, and we get the desired Equation ($\dagger$) above. □

Proof of (L) *for the Lazy*1 *language.* We have to show that:

$$[\![R\ (\lambda y.t)]\!]\rho = [\![\mathbf{rec}\, y.t]\!]\rho \tag{L}$$

with $R = \mathbf{rec}\, w.\lambda f.(f(wf))$ and $R : (\tau \to \tau) \to \tau$. By (RL1) on page 223 and the definition of the lazy1 denotational semantics of $\mathbf{rec}\, y.t$, we have to show that:

$$let\ \widetilde{\varphi} \Leftarrow \lfloor \lambda\varphi.\, fix(down\, \varphi)\rfloor \centerdot \widetilde{\varphi}([\![\lambda y.t]\!]\rho) = fix\ (\lambda u.[\![\, t\,]\!]\, \rho[u/y]).$$

This equality holds because:

$$\begin{aligned}
&let\ \widetilde{\varphi} \Leftarrow \lfloor \lambda\varphi.\, fix(down\, \varphi)\rfloor \centerdot \widetilde{\varphi}([\![\lambda y.t]\!]\rho) = \{\text{by definition of the } \textit{let} \text{ construct}\} = \\
&= fix(down\, ([\![\lambda y.t]\!]\rho)) = \{\text{by definition of } [\![\lambda y.t]\!]\rho\} = \\
&= fix(down\, \lfloor \lambda u.[\![t]\!]\rho[u/y]\rfloor) = \{\text{by } down\lfloor x\rfloor = x\} = \\
&= fix\ (\lambda u.[\![\, t\,]\!]\, \rho[u/y]).
\end{aligned}$$

□

Proof of (L) *for the Lazy*2 *language.* We have to show that:

$$[\![R\ (\lambda y.t)]\!]\rho = [\![\mathbf{rec}\, y.t]\!]\rho \tag{L}$$

with $R = \mathbf{rec}\, w.\lambda f.(f(wf))$ and $R : (\tau \to \tau) \to \tau$. By (RL2) on page 224 and the definition of the lazy2 denotational semantics of $\mathbf{rec}\, y.t$, we have to show that:

$$fix([\![\lambda y.t]\!]\rho) = fix(\lambda u.[\![\, t\,]\!]\, \rho[u/y]).$$

This equality holds in the lazy2 denotational semantics because:
$[\![\lambda y.t]\!]\rho = \lambda u.[\![\, t \,]\!]\, \rho[u/y]$. □

Proof of (EF). Consider $F{:}\tau\to\tau$ and $R{:}(\tau\to\tau)\to\tau$.
For the Eager language we have that:
$[\![F]\!]\rho \in [V_\tau \to (V_\tau)_\perp]_\perp$
$[\![R]\!]\rho \in [[V_\tau \to (V_\tau)_\perp] \to (V_\tau)_\perp]_\perp$
$\mathit{fix} \in [[V_\tau \to V_\tau] \to V_\tau]$
$\varphi \in [V_\tau \to (V_\tau)_\perp]$
$(\mathit{down} \circ \varphi) \in [V_\tau \to V_\tau]$
$[\![R]\!]\rho = \lfloor \lambda\varphi.\lfloor \mathit{fix}(\mathit{down}\circ\varphi)\rfloor\rfloor$

We have to show that:
$$[\![RF]\!]\rho = [\![F(RF)]\!]\rho \quad \text{if } \mathit{down}([\![F]\!]\rho) \neq (\lambda x.\perp) \in [V_\tau \to (V_\tau)_\perp]. \tag{EF}$$
For the left hand side of (EF) we have that:
$[\![RF]\!]\rho = \mathit{let}\ r \Leftarrow [\![R]\!]\rho,\ v \Leftarrow [\![F]\!]\rho \,\centerdot\, r(v) =$
{by definition of the *let* construct}
$= (\mathit{down}([\![R]\!]\rho))\ (\mathit{down}([\![F]\!]\rho)) =$
{by definition of $[\![R]\!]\rho$}
$= \lfloor \mathit{fix}(\mathit{down}\circ(\mathit{down}([\![F]\!]\rho)))\rfloor =$
{by definition of *fix* and $\circ$}
$= \Big\lfloor \mathit{down}\Big(\big(\mathit{down}([\![F]\!]\rho)\big)\ \big(\mathit{fix}(\mathit{down}\circ(\mathit{down}([\![F]\!]\rho)))\big)\Big)\Big\rfloor =$ (†1)
{by $\lfloor \mathit{down}(x)\rfloor = x$ for $x \neq \perp \in (V_\tau)_\perp$}
$= \big(\mathit{down}([\![F]\!]\rho)\big)\big(\mathit{fix}\ (\mathit{down}\circ(\mathit{down}([\![F]\!]\rho)))\big).$ (†2)

This last step from Expression (†1) to Expression (†2) is justified by the fact that the argument of the leftmost *down* in (†1) is different from $\perp \in (V_\tau)_\perp$ because, by hypothesis, $\mathit{down}([\![F]\!]\rho)) \neq (\lambda x.\perp) \in [V_\tau \to (V_\tau)_\perp]$.

For the right hand side of (EF) we have that:
$[\![F(RF)]\!]\rho = \mathit{let}\ f \Leftarrow [\![F]\!]\rho,\ x \Leftarrow (\mathit{let}\ r \Leftarrow [\![R]\!]\rho,\ v \Leftarrow [\![F]\!]\rho \,\centerdot\, r(v)).\ f(x)) =$
{by definition of the *let* construct}
$= (\mathit{down}([\![F]\!]\rho))\ (\mathit{down}((\mathit{down}([\![R]\!]\rho))\ (\mathit{down}([\![F]\!]\rho)))) =$
{by definition of $[\![R]\!]\rho$}
$= (\mathit{down}([\![F]\!]\rho))\ (\mathit{down}((\mathit{down}\lfloor\lambda\varphi.\lfloor\mathit{fix}(\mathit{down}\circ\varphi\rfloor\rfloor)\ (\mathit{down}([\![F]\!]\rho)))) =$
{by $\mathit{down}(\lfloor x\rfloor) = x$ and function application}
$= (\mathit{down}([\![F]\!]\rho))\ (\mathit{down}(\lfloor\mathit{fix}(\mathit{down}\circ(\mathit{down}([\![F]\!]\rho))\rfloor)) =$
{by $\mathit{down}(\lfloor x\rfloor) = x$}
$= (\mathit{down}([\![F]\!]\rho))\ (\mathit{fix}(\mathit{down}\circ(\mathit{down}([\![F]\!]\rho)))).$ (†3)
Since (†2) = (†3) the proof is completed. □

Proof of (LF1). Consider $F:\tau\to\tau$ and $R:(\tau\to\tau)\to\tau$.

For the Lazy1 language we have that:

$[\![F]\!]\rho \in [(V_\tau)_\perp \to (V_\tau)_\perp]_\perp$

$[\![R]\!]\rho \in [[(V_\tau)_\perp \to (V_\tau)_\perp]_\perp \to (V_\tau)_\perp]_\perp$

$\mathit{fix} \in [[(V_\tau)_\perp \to (V_\tau)_\perp] \to (V_\tau)_\perp]$

$[\![R]\!]\rho = \lfloor \lambda\varphi.\mathit{fix}(\mathit{down}\,\varphi)\rfloor$

We have to show that:

$[\![RF]\!]\rho = [\![F(RF)]\!]\rho \in (V_\tau)_\perp$.

This equality holds because we have that:

$$
\begin{aligned}
&[\![RF]\!]\rho = \mathit{let}\,\varphi \Leftarrow [\![R]\!]\rho \,\centerdot\, \varphi([\![F]\!]\rho) = \\
&\quad = \mathit{let}\,\widetilde{\varphi} \Leftarrow \lfloor \lambda\varphi.\mathit{fix}(\mathit{down}\,\varphi)\rfloor \,\centerdot\, \widetilde{\varphi}([\![F]\!]\rho) = \\
&\quad = (\lambda\varphi.\mathit{fix}(\mathit{down}\,\varphi))([\![F]\!]\rho) = \\
&\quad = \mathit{fix}(\mathit{down}\,([\![F]\!]\rho)), \qquad (\dagger 4)\\
&[\![F(RF)]\!]\rho = \mathit{let}\,\varphi \Leftarrow [\![F]\!]\rho \,\centerdot\, \varphi([\![RF]\!]\rho) = \{\text{see } (\dagger 4)\} = \\
&\quad = \mathit{let}\,\varphi \Leftarrow [\![F]\!]\rho \,\centerdot\, \varphi(\mathit{fix}(\mathit{down}\,([\![F]\!]\rho))) = \\
&\quad = (\mathit{down}\,([\![F]\!]\rho))\,(\mathit{fix}(\mathit{down}\,([\![F]\!]\rho))), \text{ and}
\end{aligned}
$$

for all x, $\mathit{fix}\,x = x\,(\mathit{fix}\,x)$. □

Proof of (LF2). Consider $F:\tau\to\tau$ and $R:(\tau\to\tau)\to\tau$.

For the Lazy2 language we have that:

$[\![F]\!]\rho \in [V_\tau \to V_\tau]$

$[\![R]\!]\rho \in [[V_\tau \to V_\tau] \to V_\tau]$

$\mathit{fix} \in [[V_\tau \to V_\tau] \to V_\tau]$

$[\![R]\!]\rho = \mathit{fix}$

Since in the Lazy2 language we have that $[\![\, t_1t_2 \,]\!]\,\rho = ([\![t_1]\!]\rho)([\![t_2]\!]\rho)$, we have to show that $[\![RF]\!]\rho = [\![F(RF)]\!]\rho$, that is,

$\mathit{fix}([\![F]\!]\rho) = ([\![F]\!]\rho)\,(\mathit{fix}([\![F]\!]\rho))$

Indeed, this equality holds because for any $x \in [V_\tau \to V_\tau]$ we have that $\mathit{fix}\,x = x\,(\mathit{fix}\,x)$. □

7.1. Eager Operational Semantics of Fixpoint Operators.

Let R denote the fixpoint operator $\mathbf{rec}\,y.(\lambda f.\lambda x.((f(yf))x))$ of the eager operational semantics.

We show that in the eager operational semantics the following two points hold.
Point (i): for all terms $F:\tau\to\tau$, if F has no canonical form then both RF and $F(RF)$ have no canonical form, and
Point (ii): for all terms $F:\tau\to\tau$, if F has a canonical form then RF has a canonical form, while $F(RF)$ may or may not have a canonical form.

Point (i) is immediate because in the eager operational semantics the evaluation of an application (t_1t_2) requires that both subterms t_1 and t_2 have canonical forms.

In order to show Point (ii), let us consider the term $F =_{def} \lambda u.t$ which is a canonical form. We have that:

$$\begin{array}{l} RF =_{def} (\mathbf{rec}\, y.(\lambda f.\lambda x.((f(yf))x)))\,(\lambda u.t) \rightarrow \\ \quad \rightarrow (\lambda f.\lambda x.((f(yf))x)\,[\mathbf{rec}\, y.\,(\lambda g\lambda v.((g(yg))v))\,/\,y])\;(\lambda u.t) \rightarrow \\ \quad \rightarrow \lambda x.(((\lambda u.t)(y(\lambda u.t)))x)\,[\mathbf{rec}\, y.\,(\lambda g\lambda v.((g(yg))v))\,/\,y] \rightarrow \\ \quad \rightarrow \lambda x.(((\lambda u.t)((\mathbf{rec}\, y.\,(\lambda g\lambda v.((g(yg))v)))(\lambda u.t)))x) \qquad (\alpha) \end{array}$$

which is a canonical form. Let us call it α. We also have that:

$F(RF) =_{def} (\lambda u.t)\,((\mathbf{rec}\, y.(\lambda f.\lambda x.((f(yf))x)))\,(\lambda u.t)) \rightarrow \ldots \rightarrow (\lambda u.t)\,\alpha \rightarrow t[\alpha/u]$

If in $F =_{def} \lambda u.t$ we take t to be 1 then $RF \rightarrow \alpha[1/t]$ and $F(RF) \rightarrow 1$. Thus, we get the two distinct canonical forms $\alpha[1/t]$ and 1. We have that RF and $F(RF)$ have the same canonical form if in F we take t to be u. In that case, in fact, the terms α and $t[\alpha/u]$ are both equal to $\alpha[u/t]$.

If in F we take t to be $((\mathbf{rec}\, y.(\lambda f.(f(yf))))\,(\lambda v.t_1))$ for some term t_1 then RF has the canonical form α with $((\mathbf{rec}\, y.(\lambda f.(f(yf))))\,(\lambda v.t_1))$, instead of t, and $F(RF)$ has no canonical form in the eager operational semantics. Indeed, in the eager operational semantics the application:

$(\mathbf{rec}\, y.(\lambda f.(f(yf))))\,(\lambda v.t_1)$

where $\mathbf{rec}\, y.(\lambda f.(f(yf)))$ is the fixpoint operator of the lazy operational semantics, has no canonical form.

7.2. Lazy Operational Semantics of Fixpoint Operators.

We show that in the lazy operational semantics for all terms $F : \tau \rightarrow \tau$ and all canonical forms c, $RF \rightarrow c$ iff $F(RF) \rightarrow c$, where R denotes the fixpoint operator $\mathbf{rec}\, y.\lambda f.(f(yf))$ of the lazy operational semantics. Indeed, we have that:

$RF \rightarrow c$

{by definition of R}

iff $(\mathbf{rec}\, y.\lambda f.(f(yf)))\,F \rightarrow c$

{by the operational rule for **rec**}

iff $\lambda f.(f((\mathbf{rec}\, y.\,\lambda g.(g(yg)))f))\,F \rightarrow c$

{by the operational rule for function application}

iff $F((\mathbf{rec}\, y.\,(\lambda g.g(yg)))F) \rightarrow c$

{by definition of R}

iff $F(RF) \rightarrow c$.

7.3. Fixpoint Operators in Type Free, Higher Order Languages.

If we consider type free, higher order languages where self-application is allowed, we can express the fixpoint operators without using the **rec** operator. In particular, for a language with eager operational semantics we have the fixpoint operator:

$Y^* =_{def} \lambda f.\big(\lambda z.\big((\lambda x.f(\lambda y.xxy))\,(\lambda x.f(\lambda y.xxy))z\big)\big)$

and for a language with lazy operational semantics we have the fixpoint operator:

$Y =_{def} \lambda f.\big((\lambda x.f(xx))\,(\lambda x.f(xx))\big)$.

Note that in the definitions of both Y^* and Y the self application of x to itself is allowed because we consider type free languages. Also recall that application is left associative, that is, for instance, xxy stands for $((xx)y)$.

By using the rules of the operational semantics of the Eager language (see page 202) and by writing Δ^*, instead of $\lambda x.\tau(\lambda y.xxy)$, we have that:
for any abstraction $\tau =_{def} \lambda f.t$, for any canonical form c, c_1,

$$Y^*\tau\, c \to^e c_1 \qquad (\alpha 0)$$

$$\text{iff } (\lambda z.\Delta^*\Delta^* z)\, c \to^e c_1, \qquad (\alpha 1)$$

$$\text{iff } \Delta^*\Delta^* c \to^e c_1$$

$$\text{iff } \tau(\lambda y.\Delta^*\Delta^* y)\, c \to^e c_1. \qquad (\alpha 2)$$

From $(\alpha 0)$, $(\alpha 1)$, and $(\alpha 2)$, we get that $Y^*\tau$ behaves as $\tau(Y^*\tau)$ and, thus, Y^* behaves as a fixpoint operator in a type free version of the Eager language.

Exercise 7.1. Show that (see also Section 2.4 on page 206):

$$Y^*(\lambda f.\lambda x.\mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x-1))\, 2 \to^e 2 \qquad \square$$

Similarly, by using the rules of the operational semantics of the Lazy language (see page 204) and by writing Δ, instead of $\lambda x.\tau(xx)$, we have that:
for any abstraction $\tau =_{def} \lambda f.t$, for any canonical form c, c_1,

$$Y\tau\, c \to^\ell c_1 \qquad (\beta 0)$$

$$\text{iff } \Delta\Delta\, c \to^\ell c_1 \qquad (\beta 1)$$

$$\text{iff } \tau(\Delta\Delta)\, c \to^\ell c_1. \qquad (\beta 2)$$

From $(\beta 0)$, $(\beta 1)$, and $(\beta 2)$, we get that $Y\tau$ behaves as $\tau(Y\tau)$ and, thus, Y behaves as a fixpoint operator in a type free version of the Lazy language.

Exercise 7.2. Show that (see also Section 2.5 on page 207):

$$Y(\lambda f.\lambda x.\mathbf{if}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, x \times f(x-1))\, 2 \to^\ell 2. \qquad \square$$

8. Adequacy

In this section we establish a correspondence between the operational and the denotational semantics for the three languages: (i) the Eager language, (ii) the Lazy1 language, and (iii) the Lazy2 language.

Let us introduce the following definitions. For any closed, typable term t of type τ, we say that:

(i) t *converges in the eager operational semantics*, denoted $t \downarrow^e$, iff there exists a canonical form c such that $t \to^e c$, where $\to^e$ denotes the eager operational semantics relation,

(ii) t *converges in the lazy operational semantics*, denoted $t \downarrow^\ell$, iff there exists a canonical form c such that $t \to^\ell c$, where $\to^\ell$ denotes the lazy operational semantics relation, and

(iii) t *converges in the eager denotational semantics*, denoted $t \Downarrow^e$, iff there exists $v \in V_\tau$ such that for all environments ρ, $[\![t]\!]^e\, \rho = \lfloor v \rfloor$,

(iv) t *converges in the lazy*1 *denotational semantics*, denoted $t \Downarrow^{\ell 1}$, iff there exists $v \in V_\tau$ such that for all environments ρ, $[\![t]\!]^{\ell 1}\,\rho = \lfloor v \rfloor$.

The definition of convergence in the lazy2 denotational semantics, denoted $t \Downarrow^{\ell 2}$, cannot be given in a way which is similar to that of convergence in the lazy1 denotational semantics (see Point (iv) above), because for any term t of type τ, for any environment ρ, $[\![t]\!]^{\ell 2}\,\rho$ is of type V_τ, and not of type $(V_\tau)_\perp$ and, in general, we do not assume the existence of a bottom element in the cpo V_τ.

Thus, we introduce the following definition. For any closed, typable term t of type τ, we say that:

(v) t *converges in the lazy*2 *denotational semantics*, denoted $t \Downarrow^{\ell 2}$, iff if $\tau = int$ then there exists $n \in N$ such that for all environments ρ, $[\![t]\!]^{\ell 2}\,\rho = \lfloor n \rfloor$.

Now we introduce the three notions of adequacy of the denotational semantics with respect to the operational semantics for: (i) the Eager language, (ii) the Lazy1 language, and (iii) the Lazy2 language. These three definitions follow the same pattern and they can be derived one from the other by making the changes indicated in the following Table 10.

	Eager		Lazy1		Lazy2	
operational semantics and operational convergence	$\rightarrow^e$	$\downarrow^e$	$\rightarrow^\ell$	$\downarrow^\ell$	$\rightarrow^\ell$	$\downarrow^\ell$
denotational semantics and denotational convergence	$[\![\_]\!]^e$	$\Downarrow^e$	$[\![\_]\!]^{\ell 1}$	$\Downarrow^{\ell 1}$	$[\![\_]\!]^{\ell 2}$	$\Downarrow^{\ell 2}$

TABLE 10. Notations for operational semantics, denotational semantics, operational convergence, and denotational convergence. Note that Lazy1 and Lazy2 languages have the same operational semantics.

DEFINITION 8.1. **[Adequacy of the Eager Denotational Semantics]** The eager denotational semantics $[\![\_]\!]^e$ is said to be *adequate* w.r.t. the eager operational semantics $\rightarrow^e$ iff

(A^e) for all closed, typed terms t, $t \downarrow^e$ iff $t \Downarrow^e$, and

($\overrightarrow{B^e}$) for all closed, typed terms t, there exists a canonical form c such that $t \rightarrow^e c$ implies for all environments ρ, $[\![t]\!]^e\,\rho = [\![c]\!]^e\,\rho$.

DEFINITION 8.2. **[Adequacy of the Lazy1 Denotational Semantics]** The lazy1 denotational semantics $[\![\_]\!]^{\ell 1}$ is said to be *adequate* w.r.t. the lazy operational semantics $\rightarrow^\ell$ iff

($A^{\ell 1}$) for all closed, typed terms t, $t \downarrow^\ell$ iff $t \Downarrow^{\ell 1}$, and

($\overrightarrow{B^{\ell 1}}$) for all closed, typed terms t, there exists a canonical form c such that $t \rightarrow^\ell c$ implies for all environments ρ, $[\![t]\!]^{\ell 1}\,\rho = [\![c]\!]^{\ell 1}\,\rho$.

DEFINITION 8.3. **[Adequacy of the Lazy2 Denotational Semantics]** The lazy2 denotational semantics $[\![\_]\!]^{\ell 2}$ is said to be *adequate* w.r.t. the lazy operational semantics $\rightarrow^{\ell}$ iff

$(A^{\ell 2})$ for all closed term t of type τ, $t \downarrow^{\ell}$ iff $t \Downarrow^{\ell 2}$, and

$(\overrightarrow{B^{\ell 2}})$ for all closed, typed terms t, there exists a canonical form c such that $t \rightarrow^{\ell} c$ implies for all environments ρ, $[\![t]\!]^{\ell 2}\,\rho = [\![c]\!]^{\ell 2}\,\rho$.

The following theorem shows that if in the definitions of the properties $(\overrightarrow{B^{e}})$, $(\overrightarrow{B^{\ell 1}})$, and $(\overrightarrow{B^{\ell 2}})$ we replace 'implies' by 'iff', then adequacy does not hold.

THEOREM 8.4. For some closed, typed terms t_1 and t_2 both of type $int \rightarrow int$,

(i) $[\![t_1]\!]^{e}\,\rho = [\![t_2]\!]^{e}\,\rho$ does not imply $t_1 \rightarrow^{e} t_2$,

(ii) $[\![t_1]\!]^{\ell 1}\,\rho = [\![t_2]\!]^{\ell 1}\,\rho$ does not imply $t_1 \rightarrow^{\ell} t_2$, and

(iii) $[\![t_1]\!]^{\ell 2}\,\rho = [\![t_2]\!]^{\ell 2}\,\rho$ does not imply $t_1 \rightarrow^{\ell} t_2$.

PROOF. (i) Let us consider the terms $\lambda x.x+0$ and $\lambda x.x$.
We have that: $[\![\lambda x.x+0]\!]^{e}\,\rho = [\![\lambda x.x]\!]^{e}\,\rho$. However, since in the eager operational semantics $\lambda x.x+0$ and $\lambda x.x$ are distinct canonical forms, it is not the case that $\lambda x.x+0 \rightarrow^{e} \lambda x.x$.

The proofs of (ii) and (iii) are analogous to the proof of (i). □

Let us also introduce the following notations. For the superscript $^{\square}$ which is ^{e}, or $^{\ell 1}$, or $^{\ell 2}$,

(i) let $(\overrightarrow{A^{\square}})$ denote the formula $(A^{\square})$ with 'implies', instead of 'iff', and

(ii) let $(B^{\square})$ denote the formula $(\overrightarrow{B^{\square}})$ with 'iff', instead of 'implies'.

Let $A^{e}int$, $B^{e}int$, $A^{\ell 1}int$, $B^{\ell 1}int$, $A^{\ell 2}int$, and $B^{\ell 2}int$ denote the properties derived from the properties A^{e}, B^{e}, $A^{\ell 1}$, $B^{\ell 1}$, $A^{\ell 2}$, and $B^{\ell 2}$, respectively, by assuming that the quantifications are over terms of type *int* only.

THEOREM 8.5. We have that:

(i) $(A^{e}int)$ is equivalent to $(B^{e}int)$,

(ii) $(A^{\ell 1}int)$ is equivalent to $(B^{\ell 1}int)$, and

(iii) $(A^{\ell 2}int)$ is equivalent to $(B^{\ell 2}int)$.

PROOF. *Point* (i). We have that:
$(A^{e}int)$ is: for all closed terms t of type *int*,
$\big(\exists n \in N,\ t \rightarrow^{e} n\big)$ iff $\big(\exists m \in N,\ \forall \rho,\ [\![t]\!]^{e}\rho = \lfloor m \rfloor\big)$
$(B^{e}int)$ is: for all closed terms t of type *int*, $\exists n \in N$, $\big(t \rightarrow^{e} n$ iff $\forall \rho,\ [\![t]\!]^{e}\rho = \lfloor n \rfloor\big)$.
We have to show that: (i.1) $(A^{e}int)$ implies $(B^{e}int)$, and (i.2) $(B^{e}int)$ implies $(A^{e}int)$.
Point (i.1) follows from the fact which we now show, that $(A^{e}int)$ implies
for all closed terms t of type *int*, for all $n \in N$, $\big(t \rightarrow^{e} n$ iff $\forall \rho,\ [\![t]\!]^{e}\rho = \lfloor n \rfloor\big)$.
Take any t of type *int*. Take any n. We have to show that:
(i.1.1) $t \rightarrow^{e} n$ implies $(\forall \rho,\ [\![t]\!]^{e}\rho = \lfloor n \rfloor)$ and
(i.1.2) $(\forall \rho,\ [\![t]\!]^{e}\rho = \lfloor n \rfloor)$ implies $t \rightarrow^{e} n$.

Point (i.1.1). Assume that $t \rightarrow^e n$. By Lemma 11.11 [**19**, page 191] we have that $\forall\rho$, $[\![t]\!]^e\rho = \lfloor n \rfloor$.
Point (i.1.2). Take any ρ. Assume that $[\![t]\!]^e\rho = \lfloor n \rfloor$. By Corollary 11.15 [**19**, page 200] we have that $\exists m$, $t \rightarrow^e m$. Now we have that $m = n$ because, by absurdum, if $t \rightarrow^e m$ and $t \rightarrow^e n$ and $m \neq n$ we get by (i.1.1) that for all ρ, $[\![t]\!]^e\rho = \lfloor m \rfloor = \lfloor n \rfloor$ and $m \neq n$. This is impossible because $\lambda t.[\![t]\!]^e\rho$ is a function.
Point (i.2) is obvious.

Point (ii). The proof is similar to that of Point (i) by referring to Lemma 11.20 [**19**, page 205] and Corollary 11.24 [**19**, page 209], instead of Lemma 11.11 and Corollary 11.15, respectively.

Point (iii). The proof is similar to that of Point (i) by referring to Exercise 11.28 (2) [**19**, page 217] and Exercise 11.28 (3) [**19**, page 217], instead of Lemma 11.11 and Corollary 11.15, respectively. □

As a consequence of Theorem 8.4, the above Theorem 8.5 cannot be extended to the case where the term t has a type which is not *int*. In particular, for a closed term t of type $int \rightarrow int$, we have that (A^e) holds, while (B^e) does *not* hold.

Indeed, let us consider the term $\lambda x.t_1$, for some variable x and term t_1, both of type *int*. Then $\lambda x.t_1 \downarrow^e$ holds (because $\lambda x.t_1$ is an abstraction) and $\lambda x.t_1 \Downarrow^e$ holds (because for all ρ, $[\![\lambda x.t_1]\!]^e\rho = \lfloor \lambda v.[\![t_1]\!]^e\rho[v/x] \rfloor$). Similarly for $\ell 1$, instead of e. For $\ell 2$, instead of e, we have that $(A^{\ell 2})$ holds because both $\lambda x.t_1 \downarrow^{\ell 2}$ and $\lambda x.t_1 \Downarrow^{\ell 2}$ hold.

We have the following theorems which we state without proofs.

THEOREM 8.6. [**Adequacy of the Eager Denotational Semantics**] The eager denotational semantics $[\![\_]\!]^e$ is adequate w.r.t. the eager operational semantics $\rightarrow^e$.

PROOF. It is based on the fact that for any closed term $t : \tau$, there exists a canonical form c of the Eager language such that $t \rightarrow^e c$ *iff* $\exists v \in V_\tau \, \forall\rho \, [\![t]\!]^e \, \rho = \lfloor v \rfloor$.

The *only-if* part is a consequence of the fact that: (i) for all canonical forms $c : \tau$ of the Eager language, $[\![c]\!]^e \, \rho \neq \bot \in (V_\tau)_\bot$ (see Lemma 11.8 [**19**, page 190]), and (ii) for all terms t and canonical forms c, if $t \rightarrow^e c$ then $\forall\rho \, [\![t]\!]^e \, \rho = [\![c]\!]^e \, \rho$ (see Lemma 11.11 [**19**, page 191]).

The *if* part is shown in Lemma 11.14 [**19**, page 195] whose proof uses the technique of logical relations. □

THEOREM 8.7. [**Adequacy of the Lazy1 Denotational Semantics**] The lazy1 denotational semantics $[\![\_]\!]^{\ell 1}$ is adequate w.r.t. the lazy operational semantics $\rightarrow^\ell$.

PROOF. It is based on the fact that for any closed term $t : \tau$, there exists a canonical form c of the Lazy1 language such that $t \rightarrow^\ell c$ *iff* $\exists v \in V_\tau \, \forall\rho \, [\![t]\!]^{\ell 1} \, \rho = \lfloor v \rfloor$.

The *only-if* part is a consequence of the fact that: (i) for all canonical forms $c : \tau$ of the Lazy1 language, $[\![c]\!]^{\ell 1} \, \rho \neq \bot \in (V_\tau)_\bot$ (see Lemma 11.19 [**19**, page 204]), and (ii) for all terms t and canonical forms c, if $t \rightarrow^\ell c$ then $\forall\rho \, [\![t]\!]^{\ell 1} \, \rho = [\![c]\!]^{\ell 1} \, \rho$ (see Lemma 11.20 [**19**, page 205]).

The *if* part is shown in Lemma 11.23 [**19**, page 206] whose proof uses the technique of logical relations. □

As a consequence of Theorems 8.5, 8.6, and 8.7, and Exercise 11.28 (3) [**19**, page 217], we have the following corollary.

COROLLARY 8.8. For every closed term t of type int and $n \in N$,

(i) $(B^e int)$: $t \to^e n$ iff $[\![t]\!]^e\,\rho = [\![n]\!]^e\,\rho$,

(ii) $(B^{\ell 1} int)$: $t \to^\ell n$ iff $[\![t]\!]^{\ell 1}\,\rho = [\![n]\!]^{\ell 1}\,\rho$, and

(iii) $(B^{\ell 2} int)$: $t \to^\ell n$ iff $[\![t]\!]^{\ell 2}\,\rho = [\![n]\!]^{\ell 2}\,\rho$.

Recall that for any $n \in N$, for any environment ρ, we have that: $[\![n]\!]^e\,\rho = [\![n]\!]^{\ell 1}\,\rho = [\![n]\!]^{\ell 2}\,\rho = \lfloor n \rfloor$.

We have the following result.

THEOREM 8.9. (i) $(\overrightarrow{B^e})$ implies $(\overrightarrow{A^e})$. (ii) $(\overrightarrow{B^{\ell 1}})$ implies $(\overrightarrow{A^{\ell 1}})$.
(iii) $(\overrightarrow{B^{\ell 2}})$ implies $(\overrightarrow{A^{\ell 2}})$.

PROOF. *Point* (i). First note that $(\overrightarrow{B^e})$: $\forall t\ \exists$ a canonical form c, $t \to^e c$ implies $[\![t]\!]^e\,\rho = [\![c]\!]^e\,\rho$, is equivalent to: (1) $\forall t\ \forall$ canonical form c, $t \to^e c$ implies $[\![t]\!]^e\,\rho = [\![c]\!]^e\,\rho$, because the relation $\to^e$ is deterministic. In order to prove Point (i), we assume (1) and (2): $\forall t\ \forall$ canonical forms c, $t \to^e c$, and we have to show: $\forall$ terms $t : \tau$, $\exists$ a value $v \in V_\tau$, $\forall$ environments ρ, $[\![t]\!]^e\,\rho = \lfloor v \rfloor \in (V_\tau)_\perp$. From (1) by (2) we get: $\exists$ a canonical form c, $[\![t]\!]^e\,\rho = [\![c]\!]^e\,\rho$. By Lemma 11.8 (ii) [**19**, page 190], we have that $[\![c]\!]^e\,\rho \neq \perp \in (V_\tau)_\perp$. This completes the proof of Point (i).
Point (ii). Analogous to the proof of Point (i) using Lemma 11.19 (ii) [**19**, page 204].
Point (iii). It is obvious if t of a type different from int. If t is of type int, the proof follows from Exercise 11.28 (3) [**19**, page 217]. □

Let us consider the following two terms of the Lazy language:
(i) $\mathbf{rec}\, w.w$, also denoted Ω, where the variable w is of type $int \to int$ and the term $\mathbf{rec}\, w.w$ is of type $int \to int$, and
(ii) $\lambda x.((\mathbf{rec}\, w.w)\, x)$, also denoted $\lambda x.(\Omega\, x)$, where the variable x is of type int, and the variable w and the term $\lambda x.((\mathbf{rec}\, w.w)\, x)$ are both of type $int \to int$.

We have the following fact and theorem.

FACT 8.10. For any environment ρ, $[\![\Omega]\!]^{\ell 2}\rho = [\![\lambda x.(\Omega x)]\!]^{\ell 2}\rho$ (both sides belong to $[N_\perp \to N_\perp]$).

PROOF. This lemma is a consequence of the fact that the η-rule holds in the Lazy2 language (see page 222). We have that for any ρ,

$[\![\Omega]\!]^{\ell 2}\rho = [\![\mathbf{rec}\, w.w]\!]^{\ell 2}\rho = \mathit{fix}(\lambda d.[\![w]\!]^{\ell 2}\rho[d/w]) =$

$= \mathit{fix}(\lambda d.d)$, with $\mathit{fix} \in [[[N_\perp \to N_\perp] \to [N_\perp \to N_\perp]] \ \to\ [N_\perp \to N_\perp]] =$

$= \bigsqcup_{n \geq 0} (\lambda d.d)^n(\perp)$, where $\perp$ is the function $\lambda n \in N_\perp.\perp \in N_\perp =$

$= \lambda n.\perp$, with $n \in N_\perp$ and $\perp \in N_\perp$.

Recall that $\lambda n.\perp$ is the bottom element in $[N_\perp \to N_\perp]$. □

REMARK 8.11. For any ρ, $[\![\Omega]\!]^{\ell 1}\rho \neq [\![\lambda x.(\Omega x)]\!]^{\ell 1}\rho$ (both sides belong to $[N_\perp \to N_\perp]_\perp$). Indeed, for the left hand side we have that:

$[\![\Omega]\!]^{\ell 1}\rho = [\![\mathbf{rec}\, w.w]\!]^{\ell 1}\rho = \mathit{fix}(\lambda d.[\![w]\!]^{\ell 1}\rho[d/w]) =$

$= \mathit{fix}(\lambda d.d)$, with $\mathit{fix} \in [[[N_\perp \to N_\perp]_\perp \to [N_\perp \to N_\perp]_\perp] \ \to\ [N_\perp \to N_\perp]_\perp] =$

$= \bigsqcup_{n\geq 0}(\lambda d.d)^n(\bot)$, where $\bot \in [N_\bot \to N_\bot]_\bot =$
$= \bot$, with $\bot \in [N_\bot \to N_\bot]_\bot$.

For the right hand side we have that:

$[\![\lambda x.(\Omega x)]\!]^{\ell 1}\rho = \lfloor \lambda v.([\![\Omega\, x]\!]^{\ell 1}\rho[v/x])\rfloor \in [N_\bot \to N_\bot]_\bot$ with $v \in N_\bot =$
$= \lfloor \lambda v.(let\ \varphi \Leftarrow [\![\Omega]\!]^{\ell 1}\rho[v/x]\,.\,\varphi([\![x]\!]^{\ell 1}\rho[v/x]))\rfloor =$
$= \{$since $[\![\Omega]\!]^{\ell 1}\rho = \bot \in [N_\bot \to N_\bot]_\bot$, as we have shown above, we have that
$\varphi = (\lambda n.\bot) \in [N_\bot \to N_\bot]\} =$
$= \lfloor \lambda v.((\lambda n.\bot)([\![x]\!]^{\ell 1}\rho[v/x]))\rfloor =$
$= \lfloor \lambda v.\bot\rfloor$, with $\lfloor \lambda v.\bot\rfloor \in [N_\bot \to N_\bot]_\bot$. □

THEOREM 8.12. **[The Lazy2 Denotational Semantics is not adequate]** The lazy2 denotational semantics $[\![\_]\!]^{\ell 2}$ is *not* adequate w.r.t. the lazy operational semantics $\to^\ell$.

PROOF. Let us assume, by absurdum, that the lazy2 denotational semantics $[\![\_]\!]^{\ell 2}$ is adequate w.r.t. the lazy operational semantics $\to^\ell$. In particular, we assume that:

$\Omega \downarrow^\ell$ iff $\Omega \Downarrow^{\ell 2}$ and (†1)

$\lambda x.(\Omega x) \downarrow^\ell$ iff $\lambda x.(\Omega x) \Downarrow^{\ell 2}$. (†2)

We have that:

	$\Omega \downarrow^\ell$	{by (†1)}
iff	$\Omega \Downarrow^{\ell 2}$	{by definition of $\Downarrow^{\ell 2}$, being Ω and $\lambda x.(\Omega x)$ of type $int \to int$}
iff	$\lambda x.(\Omega x) \Downarrow^{\ell 2}$	{by (†2)}
iff	$\lambda x.(\Omega x) \downarrow^\ell$	

Now, it cannot be the case that: $\Omega \downarrow^\ell$ iff $\lambda x.(\Omega x) \downarrow^\ell$, because:
(i) $\lambda x.(\Omega x)$ is a lazy canonical form, being an abstraction, and thus, $\lambda x.(\Omega x) \to \lambda x.(\Omega x)$, and
(ii) no lazy canonical form c exists such that $\Omega \to^\ell c$ because the only way of deducing $\mathbf{rec}\, w.w \to^\ell c$ is to prove $\mathbf{rec}\, w.w \to^\ell c$ (see the lazy operational rule for **rec** on page 204). □

Note that, if we consider a different notion of denotational convergence in the Lazy2 language (for example, a notion which assumes that a term t of type τ is convergent if its semantic value $[\![t]\!]^{\ell 2}\rho$, for some environment ρ, belongs to a fixed subset of the cpo V_τ), we still have that the lazy2 denotational semantics is not adequate w.r.t. the lazy2 operational semantics, because by Fact 8.10 on the preceding page, we have that for any ρ, $[\![\Omega]\!]^{\ell 2}\rho = [\![\lambda x.(\Omega x)]\!]^{\ell 2}\rho$.

In Table 11 on the next page we sum up the main notions and results we have presented in this Section 8. In that table we consider a closed, typed term t of type τ. The expression $t : any$ means that the term t is of any type, that is, t is typed, but we do not know its type, while the expression $t : int$ means that the term t is of type int.

In that table the down-arrow $\downarrow$ and the right-arrow $\to$ should be labeled by: (i) the superscript ^{e} if we consider the Eager language, and (ii) the superscript $^\ell$ if we consider the Lazy1 and Lazy2 languages.

Property		Eager $\Box = e$	Lazy1 $\Box = \ell 1$	Lazy2 $\Box = \ell 2$
1. $A^{\Box}$:	$t : any \quad t\downarrow$ iff $t \Downarrow$	yes	yes	no
2. $\overrightarrow{B^{\Box}}$:	$t : any \quad t \to c$ implies $[\![t]\!]\rho = [\![c]\!]\rho$	yes $^{(*)}$	yes $^{(*)}$	yes $^{(*)}$
3. $A^{\Box} int$:	$t : int \quad t\downarrow$ iff $t \Downarrow$	yes	yes	yes
4. $B^{\Box} int$:	$t : int \quad t \to n$ iff $[\![t]\!]\rho = \lfloor n \rfloor$	yes	yes	yes

TABLE 11. The arrows $\downarrow$, $\Downarrow$, and $\to$, and the semantic brackets $[\![\_]\!]$ should be labelled by the superscript ^{e}, or $^{\ell}$, or $^{\ell 1}$, or $^{\ell 2}$, according to the languages Eager, Lazy1, and Lazy2. By Theorem 8.5 on page 233 we have that $A^{\Box} int$ iff $B^{\Box} int$, for $\Box = {}^{e}$, or $^{\ell 1}$, or $^{\ell 2}$. The denotational semantics of the languages Eager and Lazy1 are adequate w.r.t. their operational semantics (that is, $A^{e} \wedge \overrightarrow{B^{e}}$ and $A^{\ell 1} \wedge \overrightarrow{B^{\ell 1}}$ hold), while the denotational semantics of the language Lazy2 is *not* (see the 'no' entry). If in row 2 we replace 'implies' by 'iff', then the three 'yes' entries marked with $^{(*)}$ become 'no', because in the Eager, Lazy1, and Lazy2 languages we have that $[\![\lambda x.x]\!]\rho = [\![\lambda x.x+0]\!]\rho$ holds, while $\lambda x.x \to \lambda x.x+0$ does not hold.

The down-arrow $\Downarrow$ and the semantic brackets $[\![\_]\!]$ should be labeled by: (i) the superscript ^{e} if we consider the Eager language, (ii) the superscript $^{\ell 1}$ if we consider the Lazy1 language, and (iii) the superscript $^{\ell 2}$ if we consider the Lazy2 language.

The superscript $^{\Box}$ should be: (i) ^{e} if we consider the Eager language, (ii) $^{\ell 1}$ if we consider the Lazy1 language, and (iii) $^{\ell 2}$ if we consider the Lazy2 language.

Adequacy of the denotational semantics with respect to the operational semantics is defined to be the conjunction of Property $A^{\Box}$ and Property $\overrightarrow{B^{\Box}}$, for the superscript $^{\Box}$ equal to ^{e}, or $^{\ell 1}$, or $^{\ell 2}$. Note that in Property $\overrightarrow{B^{\Box}}$ the term t is of *any* type and not necessarily of type *int* (as in Property $B^{\Box} int$). Thus, in particular, Table 11 indicates that: (i) the eager denotational semantics is adequate with respect to the eager operational semantics, (ii) the lazy1 denotational semantics is adequate with respect to the lazy operational semantics, while (iii) the lazy2 denotational semantics is *not* adequate with respect to the lazy operational semantics. Indeed, in the case of the lazy2 denotational semantics Property $A^{\ell 2}$ holds only for terms t of type *int* (see the entry 'yes' for Property $A^{\ell 2} int$), and not for terms of any type.

9. Half Abstraction and Full Abstraction

Let us consider a generic, higher order operational semantics relation, denoted $\_ \to \_$, and a generic, higher order denotational semantics function, denoted $[\![\_]\!]\,\rho$. For every closed, typed term t, we write $t\downarrow$ iff there exists a canonical form c such that $t \to c$.

Let us also consider the following two formulas, which are assumed to have as parameters the two terms t_1 and t_2 of the same type:

$Op(t_1, t_2)$ which holds iff for every context $C[\_]$ such that $C[t_1]$ and $C[t_2]$ are typable, closed terms, $C[t_1]\downarrow$ iff $C[t_2]\downarrow$

$Den(t_1, t_2)$ which holds iff for every environment ρ, $[\![t_1]\!]\rho = [\![t_2]\!]\rho$.

The name of the predicate *Op* derives from the fact that its definition refers to the *operational semantics*. Analogously, the name of the predicate *Den* derives from the fact that its definition refers to the *denotational semantics*.

The following definitions relate the convergence of the operational semantics to the equality of the denotational semantics. These definitions are meaningful because through contexts we can establish equality of denotational values as we now explain. For instance, in the Eager language, for any term t of type *int*, for any $n \in N$, we have that:

$t \rightarrow^e n$ iff $(\textbf{if } (t-n) \textbf{ then } 0 \textbf{ else } \Omega) \rightarrow^e 0$
iff $[\![t]\!]^e\rho = \lfloor n \rfloor$.

Analogous properties hold for the Lazy1 and Lazy2 languages.

DEFINITION 9.1. [**Half Abstraction of a Denotational Semantics with respect to an Operational Semantics**] A denotational semantics $[\![\_]\!]$ is said to be *half abstract* w.r.t. the observation of convergence of the operational semantics $\rightarrow$ (or simply, w.r.t. the operational semantics $\rightarrow$) if

for all terms t_1 and t_2, $Op(t_1, t_2)$ *if* $Den(t_1, t_2)$.

DEFINITION 9.2. [**Full Abstraction of a Denotational Semantics with respect to an Operational Semantics**] A denotational semantics $[\![\_]\!]$ is said to be *fully abstract* w.r.t. the observation of convergence of the operational semantics $\rightarrow$ (or simply, w.r.t. the operational semantics $\rightarrow$) if

for all terms t_1 and t_2, $Op(t_1, t_2)$ *if and only if* $Den(t_1, t_2)$.

THEOREM 9.3. [**The Eager Denotational Semantics is Half Abstract**] For the eager denotational semantics $[\![\_]\!]^e$ (which is adequate w.r.t. $\rightarrow^e$), we have that for all terms t_1 and t_2 of the same type, $Op(t_1, t_2)$ *if* $Den(t_1, t_2)$.

PROOF. Let us consider the terms t_1 and t_2 of type τ. If $[\![t_1]\!]^e\rho = [\![t_2]\!]^e\rho$ we get that: for every context $C[\_]$, $[\![C[t_1]]\!]^e\rho = [\![C[t_2]]\!]^e\rho$. Now there are two cases.
Case (i): $[\![C[t_1]]\!]^e\rho \neq \perp \in (V_\tau)_\perp$ and Case (ii): $[\![C[t_1]]\!]^e\rho = \perp \in (V_\tau)_\perp$.
In Case (i), from $[\![C[t_1]]\!]^e\rho = [\![C[t_2]]\!]^e\rho \neq \perp$, by adequacy, we get that: $C[t_1]\downarrow$ and $C[t_2]\downarrow$.
In Case (ii), from $[\![C[t_1]]\!]^e\rho = [\![C[t_2]]\!]^e\rho = \perp$, by adequacy, we get that: $\neg\, C[t_1]\downarrow$ and $\neg\, C[t_2]\downarrow$.
Thus, in both cases we have $C[t_1]\downarrow$ iff $C[t_2]\downarrow$. □

THEOREM 9.4. [**The Lazy1 Denotational Semantics is Half Abstract**] For the lazy1 semantics $[\![\_]\!]^{\ell 1}$ (which is adequate w.r.t. $\rightarrow^\ell$), we have that for all terms t_1 and t_2 of the same type, $Op(t_1, t_2)$ *if* $Den(t_1, t_2)$.

PROOF. Similar to the proof of Theorem 9.3. □

THEOREM 9.5. [**The Lazy2 Denotational Semantics is not Half Abstract**] For the lazy2 semantics $[\![\_]\!]^{\ell 2}$ (which is *not* adequate w.r.t. $\rightarrow^{\ell}$), it is *not* the case that for all terms t_1 and t_2 of the same type, $Op(t_1, t_2)$ *if* $Den(t_1, t_2)$.

PROOF. Let us consider the terms Ω and $\lambda x.(\Omega\, x)$ both of type $int \rightarrow int$. We have that:
(i) $[\![\Omega]\!]^{\ell 2}\rho = [\![\lambda x.(\Omega x)]\!]^{\ell 2}\rho$,
(ii) $\lambda x.(\Omega x) \rightarrow^{\ell} \lambda x.(\Omega x)$, and
(iii) no lazy operational canonical form c exists such that $\Omega \rightarrow^{\ell} c$.

Thus, by Point (i), $Den(\Omega,\ \lambda x.(\Omega x))$ holds, and by Points (ii) and (iii), it is not the case that $\Omega \downarrow^{\ell}$ iff $\lambda x.(\Omega x) \downarrow^{\ell}$, and thus, $Op(\Omega,\ \lambda x.(\Omega x))$ does *not* hold because if we take the context $C[\_]$ to be empty context, it is not the case that $C[\Omega] \downarrow^{\ell}$ iff $C[\lambda x.(\Omega x)] \downarrow^{\ell}$. □

We have the following theorems. The proofs of Theorem 9.6 and Theorem 9.7 are omitted.

THEOREM 9.6. [**The Eager Denotational Semantics is Not Fully Abstract**] The eager denotational semantics $[\![\_]\!]^{e}$ is *not* fully abstract w.r.t. the eager operational semantics $\rightarrow^{e}$.

THEOREM 9.7. [**The Lazy1 Denotational Semantics is Not Fully Abstract**] The lazy1 denotational semantics $[\![\_]\!]^{\ell 1}$ is *not* fully abstract w.r.t. the lazy operational semantics $\rightarrow^{\ell}$.

THEOREM 9.8. [**The Lazy2 Denotational Semantics is Not Fully Abstract**] The lazy2 denotational semantics $[\![\_]\!]^{\ell 2}$ is *not* fully abstract w.r.t. the lazy operational semantics $\rightarrow^{\ell}$.

PROOF. It is a consequence of Theorem 9.5. □

Table 12 and Table 13 below summarize the results of Sections 5, 6, 8, and 9. In those tables the entry 'yes' means that the rule (or the property) holds, while the entry 'no' means that the rule (or the property) does not hold.

Operational Semantics	α-rule	β-rule	η-rule
Eager	no	no	no
Lazy	no	yes	no

TABLE 12. Validity of the α-rule, β-rule, and η-rule in the operational semantics of the Eager and Lazy languages.

Denotational Semantics	α-rule	β-rule	η -rule	$A^{\Box} int$ $\wedge B^{\Box} int$	adequacy: $A^{\Box} \wedge \overrightarrow{B^{\Box}}$	abstraction half	abstraction full
Eager	yes	no	no	yes	yes	yes	no
Lazy1	yes	yes	no	yes	yes	yes	no
Lazy2	yes	yes	yes	yes	no	no	no

TABLE 13. Validity of the α-rule, β-rule, η-rule, adequacy, half abstraction, and full abstraction in the eager, lazy1, and lazy2 denotational semantics. Properties $A^{\Box}$, $\overrightarrow{B^{\Box}}$, $A^{\Box} int$, and $B^{\Box} int$ are defined in Table 11 on page 237, for $\Box = {}^{e}$ or ${}^{\ell 1}$ or ${}^{\ell 2}$. Half abstraction and full abstraction are defined in Definition 9.1 on page 238 and Definition 9.2 on page 238, respectively.

EXERCISE 9.9. Recall that for any type τ and σ, any variable x of type τ, any expression e of type σ, and any term t of type τ, we stipulate that $\mathbf{let}\, x \Leftarrow t\, \mathbf{in}\, e$ stands for $(\lambda x.e)\, t$. In particular, $\mathbf{let}\, x \Leftarrow t\, \mathbf{in}\, x$ stands for $(\lambda x.x)\, t$.

(1) Show that in the Eager language $[\![\mathbf{let}\, x \Leftarrow t\, \mathbf{in}\, x]\!]\, \rho = [\![t]\!]\, \rho$. Both values belong to $(V_\tau)_\perp$.

Solution. We have that $[\![\mathbf{let}\, x \Leftarrow t\, \mathbf{in}\, x]\!]\, \rho =$

$= [\![(\lambda x.x)t]\!]\, \rho =$
$= let\, \varphi \Leftarrow [\![\lambda x.x]\!]\, \rho,\, v \Leftarrow [\![t]\!]\, \rho \centerdot (\varphi\, v) =$
$= let\, \varphi \Leftarrow \lfloor \lambda \widetilde{v}.[\![x]\!]\, \rho[\widetilde{v}/x] \rfloor,\, v \Leftarrow [\![t]\!]\, \rho \centerdot (\varphi\, v) =$
$= let\, v \Leftarrow [\![t]\!]\, \rho \centerdot ((\lambda \widetilde{v}.\lfloor \widetilde{v} \rfloor)\, v) =$
$= let\, v \Leftarrow [\![t]\!]\, \rho \centerdot \lfloor v \rfloor$, which belongs to $(V_\tau)_\perp$.

Now, $let\, v \Leftarrow [\![t]\!]\, \rho \centerdot \lfloor v \rfloor = [\![t]\!]\, \rho$ because: (i) if $[\![t]\!]\, \rho = \perp \in (V_\tau)_\perp$ then both sides are $\perp$ and, otherwise, (ii) if $[\![t]\!]\, \rho \neq \perp$ then both sides are equal because $\lfloor down([\![t]\!]\, \rho) \rfloor = [\![t]\!]\, \rho$ (see page 91). Note that, in general, $[\![\mathbf{let}\, x \Leftarrow t\, \mathbf{in}\, e]\!]\, \rho \neq [\![t[e/x]]\!]\, \rho$ (see page 220).

(2) Show that in the Lazy1 language $[\![\mathbf{let}\, x \Leftarrow t\, \mathbf{in}\, x]\!]\, \rho = [\![t]\!]\, \rho$. Both values belong to $(V_\tau)_\perp$.

Solution. We have that $[\![\mathbf{let}\, x \Leftarrow t\, \mathbf{in}\, x]\!]\, \rho =$

$= [\![(\lambda x.x)t]\!]\, \rho =$
$= let\, \varphi \Leftarrow [\![\lambda x.x]\!]\, \rho \centerdot (\varphi\, [\![t]\!]\, \rho) =$
$= let\, \varphi \Leftarrow \lfloor \lambda v.v \rfloor \centerdot (\varphi\, [\![t]\!]\, \rho) =$
$= (\lambda v.v)\, ([\![t]\!]\, \rho) =$
$= [\![t]\!]\, \rho$, which belongs to $(V_\tau)_\perp$.

(3) Show that in the Lazy2 language $[\![\mathbf{let}\, x \Leftarrow t\, \mathbf{in}\, x]\!]\, \rho = [\![t]\!]\, \rho$. Both values belong to V_τ.

Solution. We have that $[\![\mathbf{let}\, x \Leftarrow t\, \mathbf{in}\, x]\!]\, \rho =$

$= [\![(\lambda x.x)t]\!]\, \rho =$

$= (\llbracket \lambda x.x \rrbracket\, \rho)\, (\llbracket t \rrbracket\, \rho) =$
$= (\lambda v.v)\, (\llbracket t \rrbracket\, \rho) =$
$= \llbracket t \rrbracket\, \rho$, which belongs to V_τ. □

EXERCISE 9.10. Assume that $x, y : \tau_1$, $e : \tau_2$, and $f : \tau_1 \to \tau_2$. Assume also that y does not occur free in $\mathbf{rec}\, f.\, (\lambda x.e)$.

(1) Show that in the Eager language $\llbracket \mathbf{rec}\, f.\, (\lambda x.e) \rrbracket\, \rho = \llbracket \lambda y.((\mathbf{rec}\, f.\, (\lambda x.e))\, y) \rrbracket\, \rho$. Both values belong to $[V_{\tau_1} \to (V_{\tau_2})_\perp]_\perp$. Recall that, in general, the η-rule does not hold in the Eager language.

Solution. We have that $\llbracket \mathbf{rec}\, f.\, (\lambda x.e) \rrbracket\, \rho =$
$= \lfloor \mathit{fix}(\lambda \widetilde{f}.\lambda v.\llbracket e \rrbracket\, \rho[\widetilde{f}/f, v/x]) \rfloor$. (†1)
We also have that $\llbracket \lambda y.((\mathbf{rec}\, f.\, (\lambda x.e))\, y) \rrbracket\, \rho =$
$= \lfloor \lambda v.\llbracket (\mathbf{rec}\, f.\, (\lambda x.e))\, y \rrbracket\, \rho[v/y] \rfloor =$
$= \lfloor \lambda v.\mathit{let}\, \varphi \Leftarrow \lfloor \mathit{fix}(\lambda \widetilde{f}.\lambda \widetilde{v}.\llbracket e \rrbracket\, \rho[\widetilde{f}/f, \widetilde{v}/x, v/y]) \rfloor,\, u \Leftarrow \lfloor v \rfloor \centerdot (\varphi\, u) \rfloor =$
$= \{$since y does not occur free in e and for all z, $\mathit{down}(\lfloor z \rfloor) = z\} =$
$= \lfloor \lambda v.(\mathit{fix}(\lambda \widetilde{f}.\lambda \widetilde{v}.\llbracket e \rrbracket\, \rho[\widetilde{f}/f, \widetilde{v}/x])\, v) \rfloor$
which is equal to (†1) because the η-rule holds in mathematics.

(2) Show that in the Lazy1 language $\llbracket \mathbf{rec}\, f.\, (\lambda x.e) \rrbracket\, \rho = \llbracket \lambda y.((\mathbf{rec}\, f.\, (\lambda x.e))\, y) \rrbracket\, \rho$. Both values belong to $[(V_{\tau_1})_\perp \to (V_{\tau_2})_\perp]_\perp$. Recall that, in general, the η-rule does not hold in the Lazy1 language.

Solution. We have that $\llbracket \mathbf{rec}\, f.\, (\lambda x.e) \rrbracket\, \rho =$
$= \mathit{fix}(\lambda \widetilde{f}.\llbracket \lambda x.e \rrbracket\, \rho[\widetilde{f}/f])$. (†2)
We also have that $\llbracket \lambda y.((\mathbf{rec}\, f.\, (\lambda x.e))\, y) \rrbracket\, \rho =$
$= \lfloor \lambda v.\llbracket (\mathbf{rec}\, f.\, (\lambda x.e))\, y \rrbracket\, \rho[v/y] \rfloor =$
$= \lfloor \lambda v.\mathit{let}\, \varphi \Leftarrow \mathit{fix}(\lambda \widetilde{f}.\llbracket \lambda x.e \rrbracket\, \rho[\widetilde{f}/f, v/y]) \centerdot (\varphi\, \llbracket y \rrbracket\, \rho[v/y]) \rfloor =$
$= \lfloor \lambda v.\mathit{let}\, \varphi \Leftarrow \mathit{fix}(\lambda \widetilde{f}.\llbracket \lambda x.e \rrbracket\, \rho[\widetilde{f}/f, v/y]) \centerdot (\varphi\, v) \rfloor =$
$= \{$since y does not occur free in $e\} =$
$= \lfloor \lambda v.\mathit{let}\, \varphi \Leftarrow \mathit{fix}(\lambda \widetilde{f}.\llbracket \lambda x.e \rrbracket\, \rho[\widetilde{f}/f]) \centerdot (\varphi\, v) \rfloor =$
$= \{$since $\mathit{fix}(\lambda \widetilde{f}.\llbracket \lambda x.e \rrbracket\, \rho[\widetilde{f}/f]) \neq \perp \in [(V_{\tau_1})_\perp \to (V_{\tau_2})_\perp]_\perp\} =$
$= \lfloor \lambda v.(\mathit{down}(\mathit{fix}(\lambda \widetilde{f}.\llbracket \lambda x.e \rrbracket\, \rho[\widetilde{f}/f]))\, v) \rfloor =$
$= \{$since the η-rule holds in mathematics$\} =$
$= \lfloor \mathit{down}(\mathit{fix}(\lambda \widetilde{f}.\llbracket \lambda x.e \rrbracket\, \rho[\widetilde{f}/f])) \rfloor$
which is equal to (†2) because: (i) $\mathit{fix}(\lambda \widetilde{f}.\llbracket \lambda x.e \rrbracket\, \rho[\widetilde{f}/f]) \neq \perp \in [(V_{\tau_1})_\perp \to (V_{\tau_2})_\perp]_\perp$, and (ii) for all $z \neq \perp$, $\lfloor \mathit{down}(z) \rfloor = z$.

(3) Show that in the Lazy2 language $\llbracket \mathbf{rec}\, f.\, (\lambda x.e) \rrbracket\, \rho = \llbracket \lambda y.((\mathbf{rec}\, f.\, (\lambda x.e))\, y) \rrbracket\, \rho$. Both values belong to $[V_{\tau_1} \to V_{\tau_2}]$.

Solution. The equality to be shown follows from the fact that the η-rule holds in the Lazy2 language. □

CHAPTER 8

Implementation of Operational Semantics

In this chapter we present four Prolog programs which implement, respectively, the operational semantics of:
(i) the imperative language IMP introduced in Section 1 on page 117 (see the program in Section 1 on this page),
(ii) the first order functional language REC introduced in Section 1 on page 165, both in the call-by-value and call-by-name regime (see the program in Section 2 on page 247),
(iii) the higher order, typed functional language Eager introduced in Section 1.1 on page 199 (see the program in Section 3 on page 251), and
(iv) the higher order, typed functional language Lazy introduced in Section 1.2 on page 200 (see the program in Section 4 on page 256).

1. Operational Semantics of the Imperative Language IMP

In this section we consider the simple imperative language IMP which we have introduced in Section 1 on page 117 and we provide its operational semantics via a Prolog program, called `imp.pl`.

In that program we use the *non-ground representation* for terms and, in particular, the expression variable X is represented as the Prolog variable `X`. On the contrary, in the ground representation for terms, the expression variable X is represented as the term: `var(X)` using the unary term constructor `var`.

Since every deduction rule which defines the operational semantics has a single conclusion (see Section 2.1 on page 118, Section 2.2 on page 118, and Section 2.3 on page 119), the operational semantics of the language IMP can easily be implemented via a Prolog program with definite clauses [**1**].

In order to to parse and interpret the input string, we use the so called Definite Clause Grammars [**1**]. In that formalism, having defined a syntactic category, say `prog(P)`, we need to call a Prolog goal of the form: `prog(P,"...",[])`, for binding the Prolog variable `P` to the term obtained by parsing the string `"..."`. Note that the Prolog predicate `prog` requires a third argument which is an empty list.

In the program `imp.pl` below, the reader will find the definition of the syntax of the language IMP and many comments that will help him to understand the Prolog code. He will also find some examples of program execution. In particular, after entering the Prolog system, in order to compute the factorial of 5, which is 120, we have to type:

```
f(5,F).
```

and we will get:

```
[[x,0],[y,120]]
F = 120
```

Note that the variable identifiers in IMP are written using lower case letters such as `x` or `y`. The list `[[x,0],[y,120]]` shows the store at the end of the computation: the variable `x` is bound to `0` and the variable `y` is bound to `120`.

In the program `imp.pl` we do not perform the type-checking of the input and, indeed, we assumed that the input is well-typed.

```
/**
 * ========================================================================
 *      OPERATIONAL SEMANTICS OF THE IMPERATIVE LANGUAGE IMP
 *
 * filename: imp.pl
 * Non-ground representation:
 * the expression variable X is represented as the Prolog variable X.
 * ------------------------------------------------------------------------
 * Use of a DEFINITE CLAUSE GRAMMAR for the input.
 * Do not insert blank spaces in the input string! For reasons of clarity
 * in the grammar below we have indicated the blank spaces (not to be
 * inserted) as '_'.
 * Note that we can add extra round parentheses '(' and ')' around
 * programs, arithmetic expressions, and boolean expressions.
 * program    p ::= skip  |  x:=e  |  (p;p)  |  if_b_then_p_else_p  |
 *                  while_b_do_p   |  (p)
 *
 * arithmetic expression
 *            e ::= n  |  x  |  (e+e)  |  (e-e)  |  (e*e)  |  (e)
 *
 * boolean expression (or formula)
 *            b ::= tt |  ff |  e=e  |  e=<e  |  -b  |  (b*b)  |
 *                  (b+b)  |  (b)
 *
 * variable   x ::= a | ... | z
 * digit      n ::= 0 | 1 | ... | 9
 * ========================================================================
 */
prog(P) --> "skip",                              {P = skip}.

/* Note the non-ground representation of the variable X:
 * the expression variable X is represented as the Prolog variable X,
 * not as the term var(X).                                              */
prog(P) --> expvar(X), ":=", exp(E),             {P = asg(X,E)}.

prog(P) --> "(", prog(P1), ";", prog(P2), ")",   {P = conc(P1,P2)}.
prog(P) --> "if", form(B), "then", prog(P1), "else", prog(P2),
                                                 {P = ite(B,P1,P2)}.
prog(P) --> "while", form(B), "do", prog(P1),    {P = wl(B,P1)}.
prog(P) --> "(", prog(P1), ")",                  {P = P1}.

form(B) --> exp(B1), "=", exp(B2),               {B = eq(B1,B2)}.
form(B) --> exp(B1), "=<", exp(B2),              {B = leq(B1,B2)}.
form(B) --> "(", form(B1), "*", form(B2), ")",   {B = and(B1,B2)}.
form(B) --> "(", form(B1), "+", form(B2), ")",   {B = or(B1,B2)}.
form(B) --> "-", form(B1),                       {B = not(B1)}.
form(B) --> "(", form(B1), ")",                  {B = B1}.
```

```
form(B) --> true(B).
form(B) --> false(B).
true(X)  --> [C,C], {C=116, name(X,[C,C])}. % name("tt", [116,116]) holds
false(X) --> [C,C], {C=102, name(X,[C,C])}. % name("ff", [102,102]) holds

exp(E) --> "(", exp(E1), "+", exp(E2), ")",     {E = plus(E1,E2)}.
exp(E) --> "(", exp(E1), "-", exp(E2), ")",     {E = minus(E1,E2)}.
exp(E) --> "(", exp(E1), "*", exp(E2), ")",     {E = mult(E1,E2)}.
exp(E) --> "(", exp(E1), ")",                   {E = E1}.
exp(E) --> expvar(E).
exp(E) --> digit(E).

expvar(X) --> [C], {"a"=<C, C=<"z", name(X,[C])}.
                                /* e.g., expvar(X,"y",[]) gives X = y */
digit(X)  --> [C], {"0"=<C, C=<"9", X is C - "0"}.
                                /* e.g., digit(X,"5",[])  gives X = 5 */
/* ------------------------------------------------------------------------
 * lookup(Location, Store, ValueAtLocation)                               */

lookup(Loc, [[Loc,V]|_],  V) :- !.
lookup(Loc, [[Loc1,_]|T], V) :- \+ (Loc = Loc1), lookup(Loc,T,V), !.
/* ------------------------------------------------------------------------
 * update([Location, NewValue], Store, NewStore) adds to the store a new
 * [Location, Value] pair, if Location is not already in the store.    */

update([Loc,NewV], [[Loc, _]|T], [[Loc, NewV]|T]) :- !.
update([Loc,NewV], [[Loc1, V1]|T],  [[Loc1, V1]|T1])  :- \+ (Loc = Loc1),
       update([Loc,NewV], T, T1).
update([Loc,NewV], [], [[Loc,NewV]]).

/* ------------------------------------------------------------------------
 * Evaluation of arithmetic expressions: eval(Aexp, Store, Value)
 * Note the non-ground representation:
 * the expression variable X is represented as the Prolog variable X.  */

eval(N,_,N) :- integer(N), !.
eval(X,S,N) :- lookup(X,S,N), !.
eval(plus(T0,T1),S,N)  :- eval(T0,S,N0), eval(T1,S,N1), N is N0 + N1.
eval(minus(T0,T1),S,N) :- eval(T0,S,N0), eval(T1,S,N1), N is N0 - N1.
eval(mult(T0,T1),S,N)  :- eval(T0,S,N0), eval(T1,S,N1), N is N0 * N1.

/* ------------------------------------------------------------------------
 * Evaluation of boolean expressions: beval(Bexp, Store, Value)
 * tt stands for true. ff stands for false.                            */

beval(tt,_,tt).
beval(ff,_,ff).
beval(eq(A0,A1),S,tt)  :- eval(A0,S,N0), eval(A1,S,N1), N0 =:= N1.
beval(eq(A0,A1),S,ff)  :- eval(A0,S,N0), eval(A1,S,N1), N0 =\= N1.
beval(leq(A0,A1),S,tt) :- eval(A0,S,N0), eval(A1,S,N1), N0 =< N1.
beval(leq(A0,A1),S,ff) :- eval(A0,S,N0), eval(A1,S,N1), \+ (N0 =< N1).
beval(not(B),S,ff)     :- beval(B,S,tt).
beval(not(B),S,tt)     :- beval(B,S,ff).
beval(or(B0,B1),S,tt)  :- beval(B0,S,T0), beval(B1,S,T1), (T0;T1).
beval(or(B0,B1),S,ff)  :- beval(B0,S,T0), beval(B1,S,T1), \+ (T0;T1).
beval(and(B0,B1),S,tt) :- beval(B0,S,T0), beval(B1,S,T1), (T0,T1).
beval(and(B0,B1),S,ff) :- beval(B0,S,T0), beval(B1,S,T1), \+ (T0,T1).
```

```
/* ------------------------------------------------------------------
 * Execution of commands: exec(Command, Store, NewStore)              */
exec(skip,S,S).
exec(asg(X,A),S,S1)    :- eval(A,S,V), update([X,V],S,S1).
exec(conc(C0,C1),S,S2) :- exec(C0,S,S1), exec(C1,S1,S2).
exec(ite(B,C0,_),S,S1) :- beval(B,S,tt), exec(C0,S,S1).
exec(ite(B,_,C1),S,S1) :- beval(B,S,ff), exec(C1,S,S1).
exec(wl(B,_),S,S)      :- beval(B,S,ff).
exec(wl(B,C),S,Sn)     :- beval(B,S,tt), exec(C,S,S1),
                          exec(wl(B,C),S1,Sn).
/**
 * ------------------------------------------------------------------
 * Now we give three examples of execution of our Prolog program.
 * factorial (First Version).
 * For the query ‘f(5,F).’ we get: [[x,0],[y,120]]
 *                                  F = 120
 * ------------------------------------------------------------------ */
f(N,F) :- exec(conc(asg(x,N),
               conc(asg(y,1),
               wl(leq(1,x), conc( asg(y,mult(y,x)), asg(x, minus(x,1))) )
               )), [], S),
          lookup(y,S,F),       % we compute the value of the factorial
          write(S).            % we write the final store
/**
 * ------------------------------------------------------------------
 * factorial (Second Version).
 * For the query ‘f1(5,F).’ we get: [[x,0],[y,120]]
 *                                  F = 120
 * In the string for P below, we may want to write:
 * "(x:=N;(y:=1;while(1=<x)do((y:=(y*x));(x:=(x-1)))))", instead of
 * "(y:=1;while(1=<x)do((y:=(y*x));(x:=(x-1))))". But, if we do so,
 * we get: "asg(x,’N’)", instead of "asg(x,N)", that is, we get the
 * character N, instead of the Prolog variable N. To overcome this
 * difficulty, we take the initial store to be [[x,N]], instead of [],
 * and, indeed, in [[x,N]] the variable x is bound to N, not to ’N’.
 * ------------------------------------------------------------------ */

f1(N,F) :- prog(P,"(y:=1;while(1=<x)do((y:=(y*x));(x:=(x-1))))",[]),
           exec(P, [[x,N]], S),
           lookup(y,S,F),      % we compute the value of the factorial
           write(S).           % we write the final store
/**
 * ------------------------------------------------------------------
 * greatest common divisor.
 * For the query ‘gcd(18,30,D).’  we get: [[x,6],[y,6]]
 *                                         D = 6
 * ------------------------------------------------------------------ */
gcd(M,N,D) :- exec(conc(asg(x,M),
               conc(asg(y,N),
               wl(not(eq(x,y)),
                  ite(leq(x,y), asg(y,minus(y,x)), asg(x,minus(x,y)) ) )
               )), [], S),
          lookup(x,S,D), % the value of the gcd is the value of x (and y)
          write(S).      % we write the final store
/* ================================================================== */
```

2. Operational Semantics of the First Order Language REC

In this section we consider the simple first order functional language REC that we have introduced in Section 1 on page 165, and we provide its operational semantics via a Prolog program, called `rec.pl`.

We have two operational semantics: (i) the operational semantics for the *call-by-value* regime (see Table 1 on page 167), and (ii) the operational semantics for the *call-by-name* regime (see Table 4 on page 173).

Our Prolog program `rec.pl` implements the deduction rules provided in those Tables 1 and 4.

For reasons of simplicity, we assume that the declarations of the language REC can introduce unary functions only. Obviously, as usual, we have the binary arithmetic operations $+$, $-$, and $\times$.

In our program `rec.pl`, in order to parse and interpret the input string, we use Definite Clause Grammars [**1**]. In that formalism, having defined a syntactic category, say `binding(B)`, we need to call a Prolog goal of the form: `binding(B,"...",[])`, for binding the Prolog variable `B` to the term obtained by parsing the string `"..."`. Note that the Prolog predicate `binding` requires a third argument which is an empty list.

```
/**
 * ========================================================================
 *             OPERATIONAL SEMANTICS FOR THE LANGUAGE REC
 *
 * filename: rec.pl
 * ------------------------------------------------------------------------
 * Use of a DEFINITE CLAUSE GRAMMAR for the input.
 *
 * Do not insert blank spaces in the input string! For reasons of clarity
 * in the grammar below we have indicated the blank spaces (not to be
 * inserted) as '_'.
 * Note that we can add extra round parentheses '(' and ')' around terms.
 * term     t ::= n  |  x  |  (t+t)  |  (t-t)  |  (t*t)  |
 *                 if_t_then_t_else_t |
 *                 (i_t)  |         % i is a function identifier.
 *                 (t)             % we consider unary functions only.
 *
 * variable x ::= a | ... | z
 *
 * number   n ::= 0 | 1 | ... | 9  % note: one digit only.
 *
 * function-identifier             % note: long function-identifiers.
 *          i ::= letter (letter+digit)*
 *
 * binding  i(x)=t        % note: one argument only.
 *                        % e.g., binding for factorial:
 *                                fact1(x)=if(x)then(1)else(x*fact1(x-1))
 * definitions
 *          [binding, binding, ...]
 * ========================================================================
 */
binding(B) --> fun(F), "(", termvar(X), ")=", term(T),
                                              {B = [F,var(X),T]}.
```

```
term(T) --> "if", term(T0), "then", term(T1), "else", term(T2),
                                          {T = ite(T0,T1,T2)}.
term(T) --> "(", term(T1), ")",           {T = T1}.
term(T) --> fun(F), term(T1),             {T = fun(F,T1)}.

term(T) --> "(", term(T1), "+", term(T2), ")",  {T = plus(T1,T2)}.
term(T) --> "(", term(T1), "-", term(T2), ")",  {T = minus(T1,T2)}.
term(T) --> "(", term(T1), "*", term(T2), ")",  {T = mult(T1,T2)}.
term(T) --> termvar(X),                   {T = var(X)}.
term(T) --> digit(T).

/* ------------------------------------------------------------------------
 * a function identifier is a sequence of one letter followed by 0 or
 * more letters or digits.                                                 */

fun(F)  --> letter(F1), fun1(F2),
          {name(F1,N1), name(F2,N2), append(N1,N2,N3), name(F,N3)}.
fun(F)  --> letter(F1), {F = F1}.
fun1(F) --> letterdigit(F1), fun1(F2),
          {name(F1,N1), name(F2,N2), append(N1,N2,N3), name(F,N3)}.
fun1(F) --> letterdigit(F).

letterdigit(F) --> letter(F).
letterdigit(F) --> digit(F).

termvar(X) --> letter(X).
letter(X) --> [C], {"a"=<C, C=<"z", name(X,[C])}.
                                /* e.g., letter(X,"y",[]) gives X = y */
digit(X)  --> [C], {"0"=<C, C=<"9", X is C - "0"}.
                                /* e.g., digit(X,"5",[])  gives X = 5 */
/* ------------------------------------------------------------------------
 * Appending lists                                                         */

append([],L,L).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
/* ------------------------------------------------------------------------
 * lookup(FunctionName, FunctionArg, FunctionBody, FunctionEnv)            */

lookup(Name,var(X),Body,[[Name,var(X),Body]|_]).
lookup(Name,var(X),Body,[[Name1,_,_]|T]) :- \+ (Name = Name1),
                                            lookup(Name,var(X),Body,T).

/* ------------------------------------------------------------------------
 * substituting ‘Value’ for ‘Variable’ in ‘Term’, thereby deriving
 * ‘NewTerm’: subst(Variable, Value, Term, NewTerm)                        */

subst(_,_,N,N) :- integer(N).
subst(var(X),V,var(X),V).
subst(var(X),_,var(Y),var(Y)) :- X \= Y.
subst(var(X),V,plus(T0,T1),plus(NT0,NT1)) :-
         subst(var(X),V,T0,NT0), subst(var(X),V,T1,NT1).
subst(var(X),V,minus(T0,T1),minus(NT0,NT1)) :-
         subst(var(X),V,T0,NT0), subst(var(X),V,T1,NT1).
subst(var(X),V,mult(T0,T1),mult(NT0,NT1)) :-
         subst(var(X),V,T0,NT0), subst(var(X),V,T1,NT1).
subst(var(X),V,ite(T0,T1,T2),ite(NT0,NT1,NT2)) :-
         subst(var(X),V,T0,NT0), subst(var(X),V,T1,NT1),
         subst(var(X),V,T2,NT2).
```

```
subst(var(X),V,fun(Name,Arg),fun(Name,NArg)) :- subst(var(X),V,Arg,NArg).

/* ------------------------------------------------------------------
 *          CALL-BY-VALUE: eval_v(Term, FunctionEnv, Value)
 * The evaluation of the closed term 'Term' in the function environment
 * 'FunctionEnv' returns in the call-by-value regime the value 'Value'.*/

eval_v(N,_,N) :- integer(N).
eval_v(plus(T0,T1),FEnv,N)   :- eval_v(T0,FEnv,N0), eval_v(T1,FEnv,N1),
                                N is N0 + N1.
eval_v(minus(T0,T1),FEnv,N)  :- eval_v(T0,FEnv,N0), eval_v(T1,FEnv,N1),
                                N is N0 - N1.
eval_v(mult(T0,T1),FEnv,N)   :- eval_v(T0,FEnv,N0), eval_v(T1,FEnv,N1),
                                N is N0 * N1.
eval_v(ite(T0,T1,_),FEnv,N)  :- eval_v(T0,FEnv,0),  eval_v(T1,FEnv,N).
eval_v(ite(T0,_,T2),FEnv,N)  :- eval_v(T0,FEnv,N1), N1 =\= 0,
                                eval_v(T2,FEnv,N).
eval_v(fun(Name,Arg),FEnv,N) :- eval_v(Arg,FEnv,V),
                                lookup(Name,var(X),Body,FEnv),
                                subst(var(X),V,Body,NewBody),
                                eval_v(NewBody,FEnv,N).

/* ------------------------------------------------------------------
 *          CALL-BY-NAME: eval_n(Term, FunctionEnv, Value)
 * The evaluation of the closed term 'Term' in the function environment
 * 'FunctionEnv' returns in the call-by-name regime the value 'Value'. */

eval_n(N,_,N) :- integer(N).
eval_n(plus(T0,T1),FEnv,N)   :- eval_n(T0,FEnv,N0), eval_n(T1,FEnv,N1),
                                N is N0 + N1.
eval_n(minus(T0,T1),FEnv,N)  :- eval_n(T0,FEnv,N0), eval_n(T1,FEnv,N1),
                                N is N0 - N1.
eval_n(mult(T0,T1),FEnv,N)   :- eval_n(T0,FEnv,N0), eval_n(T1,FEnv,N1),
                                N is N0 * N1.
eval_n(ite(T0,T1,_),FEnv,N)  :- eval_n(T0,FEnv,0),  eval_n(T1,FEnv,N).
eval_n(ite(T0,_,T2),FEnv,N)  :- eval_n(T0,FEnv,N1), N1 =\= 0,
                                eval_n(T2,FEnv,N).
eval_n(fun(Name,Arg),FEnv,N) :- lookup(Name,var(X),Body,FEnv),
                                subst(var(X),Arg,Body,NewBody),
                                eval_n(NewBody,FEnv,N).
/** ------------------------------------------------------------------
 *                          Various tests
 * ------------------------------------------------------------------ */
/*           Evaluation 'by value' of factorial                      */

f1_v(N,F) :- eval_v(fun(f,N),
          [[f,var(x),ite(var(x),1,mult(var(x),fun(f,minus(var(x),1))))]],
          F).

f2_v(N,F) :- binding(B,"fact1(x)=if(x)then(1)else(x*fact1(x-1))",[]),
          eval_v(fun(fact1,N),[B],F).

/* ------------------------------------------------------------------
 *           Evaluation 'by name' of factorial                      */

f3_n(N,F):- eval_n(fun(f,N),
          [[f,var(x),ite(var(x),1,mult(var(x),fun(f,minus(var(x),1))))]],
          F).
```

```
/* ------------------------------------------------------------------------
 *           Evaluation 'by value'.
 * Terminating function composition: f(x) = x;  g(x) = x+1;            */

gf1_v(N,V) :- eval_v(fun(g,fun(f,N)),
              [[f,var(x),var(x)], [g,var(x),plus(var(x),1)]],
              V).       % V = N+1

gf2_v(N,V) :- binding(B1,"f(x)=x",[]), bind(B2,"g(x)=(x+1)",[]),
              eval_v(fun(g,fun(f,N)),
              [B1,B2],
              V).       % V = N+1

/* ------------------------------------------------------------------------
 *           Evaluation 'by value'.
 * Nonterminating function composition: f(x) = f(x);  g(x) = x+1;      */

gf3_v(N,V) :- eval_v(fun(g,fun(f,N)),
              [[f,var(x),fun(f,var(x))], [g,var(x),plus(var(x),1)]],
              V).       % computation diverges

/* ------------------------------------------------------------------------
 *           Evaluation 'by name'.
 * Terminating function composition: f(x) = f(x);  g(x) = 1;           */

gf4_n(N,V) :- eval_n(fun(g,fun(f,N)),
              [[f,var(x),fun(f,var(x))], [g,var(x),1]],
              V).       % V = 1
/* ======================================================================== */
```

3. Operational Semantics of the Higher Order Language Eager

In this section we consider the higher order typed functional language Eager that we have introduced in Section 1.1 on page 199. The operational semantics of the Eager language has been defined in Table 3 on page 202. The following Prolog program `eager.pl` implements the deduction rules provided in that table.

In our program `eager.pl`, in order to parse and interpret the input string, we use Definite Clause Grammars [**1**]. In that formalism, having defined a syntactic category, say `term(T)`, we need to call a Prolog goal of the form: `term(T,"...",[])`, for binding the Prolog variable `T` to the term obtained by parsing the string `"..."`. Note that the Prolog predicate `term` requires a third argument which is an empty list.

```
/**
 * ========================================================================
 *      EAGER EVALUATION OF A HIGHER ORDER TYPED FUNCTIONAL LANGUAGE
 *
 * filename: eager.pl
 * Variables are introduced 'without types', but we assume that
 * all terms are well-typed.
 * ------------------------------------------------------------------------
 * Use of a DEFINITE CLAUSE GRAMMAR for the input.

 * Do not insert blank spaces in the input string! For reasons of clarity
 * in the grammar below we have indicated the blank spaces (not to be
 * inserted) as '_'.
 * Note that we can add extra round parentheses '(' and ')' around terms.
 * term     t ::= n  |  x  |  (t+t)  |  (t-t)  |  (t*t)  |
 *                 if_t_then_t_else_t |
 *                 <t,t>  |  p1(t)  |  p2(t)  |
 *                 \\x.t  |  (t_t)  |
 *                 let_x=t_in_t  |
 *                 rec_f.\\x.t   |
 *                 (t)
 *
 * variable x ::= a  |  ...  |  z
 * number   n ::= d  |  #dd...d
 * digit    d ::= 0  |  1  |  ...  |  9
 * ========================================================================
 */
term(T) --> "if", term(T0), "then", term(T1), "else", term(T2),
                                                {T = ite(T0,T1,T2)}.
term(T) --> "<", term(T1), ",", term(T2), ">",  {T = pair(T1,T2)}.
term(T) --> "p1(", term(T1), ")",               {T = fst(T1)}.
term(T) --> "p2(", term(T1), ")",               {T = snd(T1)}.

/* the character \ (lambda) is denoted by \\.
   Thus, in the input string, one should use \\.                        */

term(T) --> "\\", termvar(X), ".", term(T1),    {T = lam(var(X),T1)}.
term(T) --> "(", term(T1), term(T2), ")",       {T = app(T1,T2)}.
term(T) --> "let", termvar(X), "=", term(T1), "in", term(T2),
                                                {T = let(var(X),T1,T2)}.
term(T) --> "rec", termvar(X), ".", term(T1),   {T = rec(var(X),T1)}.
term(T) --> "(", term(T1), ")",                 {T = T1}.
```

```
term(T) --> "(", term(T1), "+", term(T2), ")",  {T = plus(T1,T2)}.
term(T) --> "(", term(T1), "-", term(T2), ")",  {T = minus(T1,T2)}.
term(T) --> "(", term(T1), "*", term(T2), ")",  {T = mult(T1,T2)}.
term(T) --> termvar(X),                         {T = var(X)}.

term(N) --> digit(N).
term(T) --> "#", number(T).      /* e.g., #14 represents the number 14 */
                                 /*       2 is represented by 2 or #2  */

/* Since the 2nd clause for number/1 is left recursive the order
 * of the clauses is important: basis case first.                      */

number(N) --> digit(N).
number(N) --> number(N1), digit(D),    {N is (N1*10)+D}.

/* A function identifier is a sequence of one letter followed by 0 or
 * more letters or digits.                                             */

termvar(F) --> "(", termvar(F1), ")",  {F = F1}.
termvar(F) --> letter(F1), termvar1(F2),
               {name(F1,N1), name(F2,N2), append(N1,N2,N3), name(F,N3)}.
termvar(F)  --> letter(F1), {F = F1}.
termvar1(F) --> letterdigit(F1), termvar1(F2),
          {name(F1,N1), name(F2,N2), append(N1,N2,N3), name(F,N3)}.
termvar1(F) --> letterdigit(F).

letterdigit(F) --> letter(F).
letterdigit(F) --> digit(F).

letter(X) --> [C], {"a"=<C, C=<"z", name(X,[C])}.
                                 /* e.g., letter(X,"y",[]) gives X = y */
digit(X)  --> [C], {"0"=<C, C=<"9", X is C - "0"}.
                                 /* e.g., digit(X,"5",[])  gives X = 5 */
/* ---------------------------------------------------------------------
 * Generator of new symbols.                                           */

:- assert(current_index(0)).      /* this is executed at compile time. */
gensym(NewName) :-
   (current_index(I) -> retract(current_index(I)) ; I = 0),
   I1 is I + 1, assert(current_index(I1)),
   name(I1,NameI1), name(NewName,[120, 95 | NameI1]).  /* 120 95 is x_ */


/* ---------------------------------------------------------------------
 * Appending lists.                                                    */

append([],L,L).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).


/* ---------------------------------------------------------------------
 * Deleting all occurrences of an element from a list.                 */

del(_,[],[]).
del(X,[X|L],T)    :- !, del(X,L,T).
del(X,[Y|Ys],[Y|Zs]) :- del(X,Ys,Zs).
```

```
/* ------------------------------------------------------------------------
 * Member of a list.                                                      */

member(X,[X|_]) :- !.
member(X,[_|T]) :- member(X,T).

/* ------------------------------------------------------------------------
 * Union of two sets represented as lists without repetitions.
 * Unfortunately, the order of the elements in a list is significant.
 * Be careful about this!
 * For sets the order of the elements is not significant.                 */

union([],Ys,Ys).
union([X|Xs],Ys,Zs) :- member(X,Ys), !, union(Xs,Ys,Zs).
union([X|Xs],Ys,[X|Zs]) :- union(Xs,Ys,Zs).

/* ------------------------------------------------------------------------
 * Free Variables.                                                        */

freeV(N,[]) :- integer(N).
freeV(var(X),[X]).
freeV(plus(T1,T2),S)   :- freeV(T1,S1),  freeV(T2,S2),
        union(S1,S2,S).
freeV(minus(T1,T2),S)  :- freeV(T1,S1),  freeV(T2,S2),
        union(S1,S2,S).
freeV(mult(T1,T2),S)   :- freeV(T1,S1),  freeV(T2,S2),
        union(S1,S2,S).
freeV(ite(T0,T1,T2),S) :- freeV(T0,S0), freeV(T1,S1),
        freeV(T2,S2), union(S0,S1,S01), union(S01,S2,S).
freeV(pair(T1,T2),S)   :- freeV(T1,S1), freeV(T2,S2),
        union(S1,S2,S).
freeV(fst(T),S) :- freeV(T,S).
freeV(snd(T),S) :- freeV(T,S).
freeV(lam(var(X),B),S) :- freeV(B,FB), del(X,FB,S).
freeV(app(T1,T2),S)    :- freeV(T1,S1), freeV(T2,S2), union(S1,S2,S).
freeV(let(var(X),T1,T2),S) :- freeV(T1,S1), freeV(T2,S2),
        del(X,S2,S21), union(S1,S21,S).
freeV(rec(var(F), lam(var(X),B)), S) :-
        freeV(lam(var(X),B),S1), del(F,S1,S).

/* ------------------------------------------------------------------------
 * Substitution of 'Value' for Variable in 'Term', thereby deriving
 * 'NewTerm': subst(Variable, Value, Term, NewTerm)                       */

subst(var(_),_,N,N) :- integer(N).
subst(var(X),V,var(X),V).
subst(var(X),_,var(Y),var(Y)) :- X \= Y.
subst(var(X),V,plus(T0,T1),plus(NT0,NT1)) :-
        subst(var(X),V,T0,NT0), subst(var(X),V,T1,NT1).
subst(var(X),V,minus(T0,T1),minus(NT0,NT1)) :-
        subst(var(X),V,T0,NT0), subst(var(X),V,T1,NT1).
subst(var(X),V,mult(T0,T1),mult(NT0,NT1)) :-
        subst(var(X),V,T0,NT0), subst(var(X),V,T1,NT1).
subst(var(X),V,ite(T0,T1,T2),ite(NT0,NT1,NT2)) :-
        subst(var(X),V,T0,NT0), subst(var(X),V,T1,NT1),
        subst(var(X),V,T2,NT2).
subst(var(X),V,pair(T1,T2),pair(NT1,NT2)) :-
        subst(var(X),V,T1,NT1), subst(var(X),V,T2,NT2).
```

```
subst(var(X),V,fst(T),fst(NT)) :- subst(var(X),V,T,NT).
subst(var(X),V,snd(T),snd(NT)) :- subst(var(X),V,T,NT).
subst(var(X),V,app(T1,T2),app(NT1,NT2)) :-
        subst(var(X),V,T1,NT1), subst(var(X),V,T2,NT2).


/* ------------------------------------------------------------------
 * Lambda Abstraction.                                              */

subst(var(X),_,lam(var(X),T),lam(var(X),T)).
subst(var(X),V,lam(var(Y),T),lam(var(Y),NT)) :-
        X \= Y, freeV(V,FV), freeV(T,FT),
        (\+ member(Y,FV) ; \+ member(X,FT)),
        subst(var(X),V,T,NT).
subst(var(X),V,lam(var(Y),T),lam(var(Z),NT)) :-
        X \= Y, freeV(V,FV), freeV(T,FT), member(Y,FV), member(X,FT),
        gensym(Z),
        subst(var(Y),var(Z),T,T1), subst(var(X),V,T1,NT).

/* ------------------------------------------------------------------
 * let expression.
 *
 * We use the equivalence of 'let x = t1 in t2' with '((\x.t2) t1)'.
 * The bound variable x, denoted X, does not occur free in t1,
 * denoted T1. Note that the variable X may be changed to NX.       */

subst(var(Y),V,let(var(X),T1,T2),let(var(NX),NT1,NT2)) :-
        subst(var(Y),V,app(lam(var(X),T2),T1),
        app(lam(var(NX),NT2),NT1)).
/* ------------------------------------------------------------------
 * Function Application.                                            */

subst(var(X),V,app(T1,T2),app(NT1,NT2)) :-
        subst(var(X),V,T1,NT1), subst(var(X),V,T2,NT2).

/* ------------------------------------------------------------------
 * Recursion for the Eager language.                                */

subst(var(Y),V,rec(var(F),lam(var(X),B)),
              rec(var(NF),lam(var(NX),NB))) :-
        subst(var(Y),V,lam(var(F),lam(var(X),B)),
        lam(var(NF),lam(var(NX),NB))).

/* ------------------------------------------------------------------
 * Canonical forms for the Eager language.                          */

canonical(N) :- integer(N).
canonical(pair(C1,C2))   :- canonical(C1), canonical(C2).
canonical(lam(var(X),B)) :- freeV(lam(var(X),B),[]).

/* ------------------------------------------------------------------
 * Evaluation in the Eager language.                                */

eager(plus(T1,T2),C)  :- eager(T1,C1), eager(T2,C2), C is C1 + C2.
eager(minus(T1,T2),C) :- eager(T1,C1), eager(T2,C2), C is C1 - C2.
eager(mult(T1,T2),C)  :- eager(T1,C1), eager(T2,C2), C is C1 * C2.
eager(ite(T0,T1,_),C) :- eager(T0,0),  eager(T1,C).
eager(ite(T0,_,T2),C) :- eager(T0,C0), C0 \= 0, eager(T2,C).
```

```
eager(pair(T1,T2),pair(C1,C2)) :- eager(T1,C1), eager(T2,C2).
eager(fst(T),C1) :- eager(T,pair(C1,_)).
eager(snd(T),C2) :- eager(T,pair(_,C2)).

eager(app(T1,T2),C) :- eager(T1,lam(var(X),B)), eager(T2,C2),
      subst(var(X),C2,B,NB), eager(NB,C).

eager(let(var(X),T1,T2),C) :- eager(T1,C1),
      subst(var(X),C1,T2,T21), eager(T21,C).

eager(rec(var(F),lam(var(X),B)),lam(var(X),NB)) :-
      subst(var(F),rec(var(F),lam(var(X),B)),B,NB).

eager(C,C) :- canonical(C).

/* ---------------------------------------------------------------------
 * The above clause should be last, because otherwise, no computation
 * is done because the variable C unify with any term.
 * =====================================================================
 *                          Various tests
 * ---------------------------------------------------------------------
 */
/* factorial in the Eager language.                                    */

f(N,F) :- eager(app(
            rec(var(f),
               lam(var(x),
                  ite(var(x),1,
                     mult(var(x), app(var(f), minus(var(x),1))))),
            N),
          F).
f1(N,F) :- term(T,"recfact.\\(x).if(x)then(1)else(x*(fact(x-1)))",[]),
           eager(app(T,N),F).
/* ---------------------------------------------------------------------
 * In the language Eager the execution of the term
 * ((rec f. \x.1) ((rec g. (\x.g(x))) 2)) does not terminate.
 * In the language Lazy it returns 1.                                  */

t1e(C) :- eager(app(rec(var(f),lam(var(x),1)),
             app(rec(var(g),lam(var(x), app(var(g),var(x))) ),2))
          ,C).
t11e(C) :- term(T,"((recf.\\(x).1)((recg.\\x.(gx))2))",[]),
          eager(T,C).
/* ------------------------------------------------------------------- */
t2(C) :- term(P,"p1(<#123,#3456>)",[]), eager(P,C).        /* C is 123 */
/* ------------------------------------------------------------------- */
t3(C) :- term(B,"p2(<letx=1inx,letx=1in(x*4)>)",[]),
         eager(B,C).                                       /* C is 4   */
/* ---------------------------------------------------------------------
 * rec.f.\\x.(fx) denotes the minimal fixpoint of the equation f(x)=f(x).
 * In the language Eager the term p2(...) has no normal form.
 * In the language Lazy the value of p2(...) is 32.                    */

t4(C) :- term(B,"p2(<((recf.\\x.(fx))5),#32>)",[]), eager(B,C).
/* ================================================================== */
```

4. Operational Semantics of the Higher Order Language Lazy

In this section we consider the higher order typed functional language Lazy that we have introduced in Section 1.2 on page 200. The operational semantics of the Lazy language has been defined in Table 4 on page 204. The following Prolog program `lazy.pl` implements the deduction rules provided in that table.

Most of the code in the program `lazy.pl` is similar to that of the program `eager.pl` that implements the operational semantics of the Eager language (see page 251). Indeed, most of the rules that define the operational semantics of the Eager language are equal to those that define the operational semantics of the Lazy language (compare Table 3 on page 202 with Table 4 on page 204).

In our program `lazy.pl`, in order to parse and interpret the input string, we use Definite Clause Grammars [**1**]. In that formalism, having defined a syntactic category, say `term(T)`, we need to call a Prolog goal of the form: `term(T,"...",[])`, for binding the Prolog variable `T` to the term obtained by parsing the string `"..."`. Note that the Prolog predicate `term` requires a third argument which is an empty list.

```
/**
 * ========================================================================
 *      LAZY EVALUATION OF A HIGHER ORDER TYPED FUNCTIONAL LANGUAGE
 *
 * filename: lazy.pl
 * Variables are introduced 'without types', but we assume that
 * all terms are well-typed.
 * ------------------------------------------------------------------------
 * Use of a DEFINITE CLAUSE GRAMMAR for the input.
 *
 * Do not insert blank spaces in the input string! For reasons of clarity
 * in the grammar below we have indicated the blank spaces (not to be
 * inserted) as '_'.
 * Note that we can add extra round parentheses '(' and ')' around terms.
 * term     t ::= n  |  x  |  (t+t)  |  (t-t)  |  (t*t)  |
 *                 if_t_then_t_else_t |
 *                 <t,t>  |  p1(t)  |  p2(t)  |
 *                 \\x.t  |  (t_t)  |
 *                 let_x=t_in_t  |
 *                 rec_f.t  |
 *                 (t)
 *
 * variable x ::= a  |  ...  |  z
 * number   n ::= d  |  #dd...d
 * digit    d ::= 0  |  1  |  ...  |  9
 * ========================================================================
 */
term(T) --> "if", term(T0), "then", term(T1), "else", term(T2),
                                                  {T = ite(T0,T1,T2)}.
term(T) --> "<", term(T1), ",", term(T2), ">",  {T = pair(T1,T2)}.
term(T) --> "p1(", term(T1), ")",                 {T = fst(T1)}.
term(T) --> "p2(", term(T1), ")",                 {T = snd(T1)}.

/* the character \ (lambda) is denoted by \\.
   Thus, in the input string, one should use \\.                        */

term(T) --> "\\", termvar(X), ".", term(T1),    {T = lam(var(X),T1)}.
```

```
term(T) --> "(", term(T1), term(T2), ")",       {T = app(T1,T2)}.
term(T) --> "let", termvar(X), "=", term(T1), "in", term(T2),
                                                {T = let(var(X),T1,T2)}.
term(T) --> "rec", termvar(X), ".", term(T1),   {T = rec(var(X),T1)}.
term(T) --> "(", term(T1), ")",                 {T = T1}.

term(T) --> "(", term(T1), "+", term(T2), ")",  {T = plus(T1,T2)}.
term(T) --> "(", term(T1), "-", term(T2), ")",  {T = minus(T1,T2)}.
term(T) --> "(", term(T1), "*", term(T2), ")",  {T = mult(T1,T2)}.
term(T) --> termvar(X),                         {T = var(X)}.

term(N) --> digit(N).
term(T) --> "#", number(T).      /* e.g., #14 represents the number 14 */
                                 /*       2 is represented by 2 or #2  */

/* Since the 2nd clause for number/1 is left recursive the order
 * of the clauses is important: basis case first.                       */

number(N) --> digit(N).
number(N) --> number(N1), digit(D),  {N is (N1*10)+D}.

/* A function identifier is a sequence of one letter followed by 0 or
 * more letters or digits.                                              */

termvar(F) --> "(", termvar(F1), ")",  {F = F1}.
termvar(F) --> letter(F1), termvar1(F2),
           {name(F1,N1), name(F2,N2), append(N1,N2,N3), name(F,N3)}.
termvar(F)  --> letter(F1), {F = F1}.

termvar1(F) --> letterdigit(F1), termvar1(F2),
           {name(F1,N1), name(F2,N2), append(N1,N2,N3), name(F,N3)}.
termvar1(F) --> letterdigit(F).

letterdigit(F) --> letter(F).
letterdigit(F) --> digit(F).

letter(X) --> [C], {"a"=<C, C=<"z", name(X,[C])}.
                                 /* e.g., letter(X,"y",[]) gives X = y */
digit(X)  --> [C], {"0"=<C, C=<"9", X is C - "0"}.
                                 /* e.g., digit(X,"5",[])  gives X = 5 */

/* ---------------------------------------------------------------------
 * Generator of new symbols.                                            */

:- assert(current_index(0)).      /* this is executed at compile time. */

gensym(NewName) :-
   (current_index(I) -> retract(current_index(I)) ; I = 0),
   I1 is I + 1, assert(current_index(I1)),
   name(I1,NameI1), name(NewName,[120, 95 | NameI1]).  /* 120 95 is x_ */

/* ---------------------------------------------------------------------
 * Appending lists.                                                     */

append([],L,L).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

```
/* ------------------------------------------------------------------------
 * Deleting all occurrences of an element from a list.                    */

del(_,[],[]).
del(X,[X|L],T)    :- !, del(X,L,T).
del(X,[Y|Ys],[Y|Zs]) :- del(X,Ys,Zs).


/* ------------------------------------------------------------------------
 * Member of a list.                                                      */

member(X,[X|_]) :- !.
member(X,[_|T]) :- member(X,T).

/* ------------------------------------------------------------------------
 * Union of two sets represented as lists without repetitions.
 * Unfortunately, the order of the elements in a list is significant.
 * Be careful about this!
 * For sets the order of the elements is not significant.                 */

union([],Ys,Ys).
union([X|Xs],Ys,Zs) :- member(X,Ys), !, union(Xs,Ys,Zs).
union([X|Xs],Ys,[X|Zs]) :- union(Xs,Ys,Zs).


/* ------------------------------------------------------------------------
 * Free Variables.                                                        */

freeV(N,[]) :- integer(N).
freeV(var(X),[X]).
freeV(plus(T1,T2),S)  :- freeV(T1,S1),  freeV(T2,S2),
        union(S1,S2,S).
freeV(minus(T1,T2),S) :- freeV(T1,S1),  freeV(T2,S2),
        union(S1,S2,S).
freeV(mult(T1,T2),S)  :- freeV(T1,S1),  freeV(T2,S2),
        union(S1,S2,S).
freeV(ite(T0,T1,T2),S) :- freeV(T0,S0), freeV(T1,S1),
        freeV(T2,S2), union(S0,S1,S01), union(S01,S2,S).
freeV(pair(T1,T2),S) :- freeV(T1,S1), freeV(T2,S2),
        union(S1,S2,S).
freeV(fst(T),S) :- freeV(T,S).
freeV(snd(T),S) :- freeV(T,S).
freeV(lam(var(X),B),S) :- freeV(B,FB), del(X,FB,S).
freeV(app(T1,T2),S) :- freeV(T1,S1), freeV(T2,S2), union(S1,S2,S).
freeV(let(var(X),T1,T2),S) :- freeV(T1,S1), freeV(T2,S2),
        del(X,S2,S21), union(S1,S21,S).
freeV(rec(var(X),B), S) :- freeV(lam(var(X),B),S).


/* ------------------------------------------------------------------------
 * Substitution of 'Value' for Variable in 'Term', thereby deriving
 * 'NewTerm': subst(Variable, Value, Term, NewTerm)                       */

subst(var(_),_,N,N) :- integer(N).
subst(var(X),V,var(X),V).
subst(var(X),_,var(Y),var(Y)) :- X \= Y.
subst(var(X),V,plus(T0,T1),plus(NT0,NT1)) :-
        subst(var(X),V,T0,NT0), subst(var(X),V,T1,NT1).
```

```
subst(var(X),V,minus(T0,T1),minus(NT0,NT1)) :-
         subst(var(X),V,T0,NT0), subst(var(X),V,T1,NT1).
subst(var(X),V,mult(T0,T1),mult(NT0,NT1)) :-
         subst(var(X),V,T0,NT0), subst(var(X),V,T1,NT1).
subst(var(X),V,ite(T0,T1,T2),ite(NT0,NT1,NT2)) :-
         subst(var(X),V,T0,NT0), subst(var(X),V,T1,NT1),
         subst(var(X),V,T2,NT2).
subst(var(X),V,pair(T1,T2),pair(NT1,NT2)) :-
         subst(var(X),V,T1,NT1), subst(var(X),V,T2,NT2).
subst(var(X),V,fst(T),fst(NT)) :- subst(var(X),V,T,NT).
subst(var(X),V,snd(T),snd(NT)) :- subst(var(X),V,T,NT).
subst(var(X),V,app(T1,T2),app(NT1,NT2)) :-
         subst(var(X),V,T1,NT1), subst(var(X),V,T2,NT2).


/* ---------------------------------------------------------------------
 * Lambda Abstraction.                                                 */

subst(var(X),_,lam(var(X),T),lam(var(X),T)).
subst(var(X),V,lam(var(Y),T),lam(var(Y),NT)) :-
         X \= Y, freeV(V,FV), freeV(T,FT),
         (\+ member(Y,FV) ; \+ member(X,FT)),
         subst(var(X),V,T,NT).
subst(var(X),V,lam(var(Y),T),lam(var(Z),NT)) :-
         X \= Y, freeV(V,FV), freeV(T,FT), member(Y,FV), member(X,FT),
         gensym(Z),
         subst(var(Y),var(Z),T,T1), subst(var(X),V,T1,NT).

/* ---------------------------------------------------------------------
 * let expression.
 *
 * We use the equivalence of 'let x = t1 in t2' with '((\x.t2) t1)'.
 * The bound variable x, denoted X, does not occur free in t1,
 * denoted T1. Note that the variable X may be changed to NX.          */

subst(var(Y),V,let(var(X),T1,T2),let(var(NX),NT1,NT2)) :-
         subst(var(Y),V,app(lam(var(X),T2),T1),
         app(lam(var(NX),NT2),NT1)).

/* ---------------------------------------------------------------------
 * Function Application.                                               */

subst(var(X),V,app(T1,T2),app(NT1,NT2)) :-
         subst(var(X),V,T1,NT1), subst(var(X),V,T2,NT2).

/* ---------------------------------------------------------------------
 * Recursion for the Lazy language.                                    */

subst(var(Y),V,rec(var(X),B),rec(var(NX),NB)) :-
         subst(var(Y),V,lam(var(X),B),lam(var(NX),NB)).

/* ---------------------------------------------------------------------
 * Canonical forms for the Lazy language.                              */

canonical(N) :- integer(N).
canonical(pair(T1,T2)) :- freeV(T1,[]), freeV(T2,[]).
canonical(lam(var(X),B)) :- freeV(lam(var(X),B),[]).
```

```
/* ------------------------------------------------------------------
 * Evaluation in the Lazy language.                                 */

lazy(plus(T1,T2),C)  :- lazy(T1,C1), lazy(T2,C2), C is C1 + C2, !.
lazy(minus(T1,T2),C) :- lazy(T1,C1), lazy(T2,C2), C is C1 - C2, !.
lazy(mult(T1,T2),C)  :- lazy(T1,C1), lazy(T2,C2), C is C1 * C2, !.
lazy(ite(T0,T1,_),C) :- lazy(T0,0),  lazy(T1,C).
lazy(ite(T0,_,T2),C) :- lazy(T0,C0), C0 \= 0, lazy(T2,C).

lazy(fst(T),C1) :- lazy(T,pair(T1,_)), lazy(T1,C1).
lazy(snd(T),C2) :- lazy(T,pair(_,T2)), lazy(T2,C2).

lazy(app(T1,T2),C) :- lazy(T1,lam(var(X),B)),
         subst(var(X),T2,B,NB), lazy(NB,C).

lazy(let(var(X),T1,T2),C) :- subst(var(X),T1,T2,NT2),
         lazy(NT2,C).

lazy(rec(var(X),B),C) :- subst(var(X),rec(var(X),B),B,NB),
         lazy(NB,C).

lazy(C,C) :- canonical(C).

/* ------------------------------------------------------------------
 * The above clause lazy(C,C) :- canonical(C) should be last. Otherwise,
 * no computation is done because the variable C unify with any term.
 *
 * ==================================================================
 *                          Various tests
 * ------------------------------------------------------------------
 */
/* factorial in the Lazy language.                                  */

f(N,F) :- lazy(app(
             rec(var(f),
                 lam(var(x),
                     ite(var(x),1,
                         mult(var(x), app(var(f), minus(var(x),1))))),
             N),
          F).

f1(N,F) :- term(T,"recfact.\\(x).if(x)then(1)else(x*(fact(x-1)))",[]),
           lazy(app(T,N),F).
/* ------------------------------------------------------------------
 * C should be pair(120,121).                                       */

t1(C) :- lazy(app(
             rec(var(f),
                lam(var(x),
                    ite(var(x),1,
                        mult(var(x), app(var(f), minus(var(x),1))))),
             5),
         T1), lazy(pair(T1,121),C).

/* ------------------------------------------------------------------
 * t3 should be true because for all terms t1 and t2 we have that:
 * let x=t1 in t2 == (\x.t2) t1.                                    */
```

```
t3 :- lazy(let(var(x),1,plus(var(x),var(x))),C),
      lazy(app(lam(var(x),plus(var(x),var(x))),1),C).

/* ---------------------------------------------------------------------
 * In t4(T) T should be \x_1.y. Indeed: in \y.x the variable y is bound,
 * so y 'does not exist': it is x_1.                                    */

t4(T) :- subst(var(x),var(y),lam(var(y),var(x)),T).

/* ---------------------------------------------------------------------
 * Since let double =\f.\x.f(f(x)) in ((double (\x.x+2)) N)
 * we get C = N+4.           It should be: t5(n,n+4).                  */

t5(N,C) :- lazy(let(var(d),
                    lam(var(f),
                       lam(var(x), app(var(f), app(var(f),var(x))))),
                 app( app(var(d), lam(var(x),plus(var(x),2))),
                 N)),
           C).
/* ---------------------------------------------------------------------
 * The execution of the term ((rec f. \x.1) ((rec g. (\x.g(x))) 2))
 * in the language Eager does not terminate, while
 * in the language Lazy returns 1.                                      */

t6(C) :- lazy(app(rec(var(f),lam(var(x),1)),
             app(rec(var(g),lam(var(x),
                    app(var(g),var(x))) ),2))
         ,C).                                              /* C is 1  */

t61(C) :- term(T,"((recf.\\(x).1)((recg.\\x.(gx))2))",[]),
          lazy(T,C).                                       /* C is 1  */

/* ---------------------------------------------------------------- */
t7(C) :- term(B,"p2(<letx=1inx,letx=1in(x*4)>)",[]),
         lazy(B,C).                                        /* C is 4  */

/* ---------------------------------------------------------------------
 * rec.f.\\x.(fx) denotes the minimal fixpoint of the equation f(x)=f(x).
 * In the language Eager the term p2(...) has no normal form.
 * In the language Lazy the value of p2(...) is 32.                     */

 t8(C) :- term(B,"p2(<((recf.\\x.(fx))5),#32>)",[]),
          lazy(B,C).                                       /* C is 32 */
/* ================================================================== */
```

CHAPTER 9

Parallel Programs and Proof of Their Properties

In this chapter we present: (i) the CCS calculus which is a calculus whose terms can be used to define parallel programs, also called *processes*, (ii) the modal μ-calculus whose formulas can be used to define properties of parallel programs, and (iii) an algorithm, called the *local model checker*, that allows us to prove properties of parallel programs.

1. The Pure CCS Calculus

In this section we present the *pure CCS calculus*, or *CCS calculus*, for short, which is a calculus for defining parallel programs which communicate with each other via handshaking communications. (CCS stands for Calculus for Communicating Systems.) This calculus was introduced by Robin Milner [**10, 11**]. We follow the presentation of [**15**, Section 6].

Here are the syntactic categories of the pure CCS calculus.

(i) The set A of *names*. For every name $\ell \in A$ we assume that there exists a *co-name*, denoted by $\overline{\ell}$. The set of co-names is denoted by $\overline{A}$. We assume that for any $\ell \in A$, $\overline{\overline{\ell}} = \ell$.

(ii) The set of *labels*, which is $A \cup \overline{A}$.

(iii) The set Act of *actions*, which is $A \cup \overline{A} \cup \{\tau\}$, where τ is a distinguished element, called the *invisible action*. The actions in $A \cup \overline{A}$ are said to be *visible actions*. $\alpha, \beta, \ldots$ range over Act.

(iv) A function (not necessarily a bijection) f from Act to Act is said to be a *renaming function* if it preserves co-names, that is, $f(\overline{\ell}) = \overline{f(\ell)}$ and $f(\tau) = \tau$.

In particular, given the set $A = \{\alpha, \beta, \gamma\}$ of actions, the function f defined by the following equations is a renaming function:

$$\begin{array}{ll} f(\alpha) = \gamma & f(\overline{\alpha}) = \overline{\gamma} \\ f(\beta) = \gamma & f(\overline{\beta}) = \overline{\gamma} \\ f(\gamma) = \overline{\alpha} & f(\overline{\gamma}) = \alpha \end{array}$$

(v) The set Id of *identifiers*. P ranges over Id. They are introduced by *definitions* (see below).

(vi) The set $Proc$ of *processes* (also called *terms* or *agents*). p (with possible subscripts) ranges over $Proc$.

$$p ::= 0 \mid \alpha.p \mid \sum_{i \in I} p_i \mid p_1 \,|\, p_2 \mid p \backslash L \mid p[f] \mid P$$

where: 0 (pronounced *zero*) is a distinguished process, α is an action in Act, I is a set of indexes, $L \subseteq A$ is a set of names, f is a renaming function, and P is a process identifier in Id.

When denoting processes we will feel free to use, instead of p, also the identifiers $P, Q, R, \ldots$

(vii) The set of *definitions* of the form: $P =_{def} p$. In definitions we allow ourselves to write $=$, instead of $=_{def}$. (Note that the symbol $=$ is also used below to denote the bisimulation congruence between processes.)

Instead of introducing definitions, we can introduce processes of the form: $\mathbf{rec}\, P.p$. Every process defined by the term $\mathbf{rec}\, P.p$ should be viewed as the process P whose definition is: $P =_{def} p$.

Now we give the informal semantics of the processes.

The process 0 is the process which can do no action. The process $\alpha.p$ is the process which can do the action α and then become the process p. The process $\sum_{i \in I} p_i$ is the process which can be any of the processes p_i for $i \in I$. The process $p_1 \mid p_2$ is the process which behaves as either p_1 or p_2 or establishes a communication between p_1 and p_2. The process $p \backslash L$ behaves like the process p, but it cannot do any action in $L \cup \overline{L}$. The renamed process $p[f]$ behaves like p, except that the actions are renamed. The process whose identifier is P behaves like p, if the definition of P is $P =_{def} p$ (or, equivalently, $\mathbf{rec}\, P.p$).

When writing CCS terms we assume the following syntactical equivalences:
(i) process 0 is syntactically equivalent to $\sum_{i \in I} p_i$, whenever $I = \emptyset$,
(ii) $\sum_{i \in \{0,1\}} p_i$ is syntactically equivalent to $p_0 + p_1$, and
(iii) the operators $+$ and $\mid$ are commutative and associative (thus, for instance, $p_0 + p_1$, is syntactically equivalent to $p_1 + p_0$).

In Table 1 on the facing page we define the operational semantics of CCS processes by introducing the binary relation $\xrightarrow{\alpha} \subseteq Proc \times Proc$, for each $\alpha \in Act$. We will not define the denotational semantics of CCS processes. For the semantics of nondeterministic computations the interested reader may look at [**16**].

DEFINITION 1.1. [α-**derivative**] For all processes P and P', for all actions $\alpha \in Act$, if $P \xrightarrow{\alpha} P'$ we say that P' is an α-*derivative* of P.

We have that, for any action α and any process P and Q,
(i) $\alpha.P \mid \overline{\alpha}.Q \xrightarrow{\alpha} P \mid \overline{\alpha}.Q$, (ii) $\alpha.P \mid \overline{\alpha}.Q \xrightarrow{\overline{\alpha}} \alpha.P \mid Q$, and (iii) $\alpha.P \mid \overline{\alpha}.Q \xrightarrow{\tau} P \mid Q$.
We also have that, for any action α and any process P and Q,
$(\alpha.P \mid \overline{\alpha}.Q)\backslash\{\alpha\} \xrightarrow{\tau} P \mid Q$, and neither α-derivative nor $\overline{\alpha}$-derivative exists for the term $(\alpha.P \mid \overline{\alpha}.Q)\backslash\{\alpha\}$.

Now we will define two relations, subsets of $Proc \times Proc$: (i) the *bisimulation equivalence*, also called *bisimilarity*, denoted $\approx$, and (ii) the *bisimulation congruence*, also called *equality*, denoted $=$.

Note that we can view each of these two relations as the definition of a denotational semantics of CCS terms which associates with any CCS process p its equivalence class.

Prefix:

$$\alpha.p \xrightarrow{\alpha} p$$

Sum:

$$\frac{p_j \xrightarrow{\alpha} q}{\sum_{i\in I} p_i \xrightarrow{\alpha} q} \quad \text{if } j \in I$$

Parallel composition:

$$\frac{p_1 \xrightarrow{\alpha} p_1'}{p_1 \,|\, p_2 \xrightarrow{\alpha} p_1' \,|\, p_2} \qquad \frac{p_2 \xrightarrow{\alpha} p_2'}{p_1 \,|\, p_2 \xrightarrow{\alpha} p_1 \,|\, p_2'} \qquad \frac{p_1 \xrightarrow{\ell} p_1' \qquad p_2 \xrightarrow{\overline{\ell}} p_2'}{p_1 \,|\, p_2 \xrightarrow{\tau} p_1' \,|\, p_2'}$$

Restriction:

$$\frac{p \xrightarrow{\alpha} q}{p\backslash L \xrightarrow{\alpha} q\backslash L} \quad \text{if } \alpha \notin L \cup \overline{L} \text{ for any set } L \subseteq A \text{ of names}$$

Relabelling:

$$\frac{p \xrightarrow{\alpha} q}{p[f] \xrightarrow{f(\alpha)} q[f]} \quad \text{for any renaming function } f$$

Identifier:

$$\frac{p \xrightarrow{\alpha} q}{P \xrightarrow{\alpha} q} \quad \text{where } P =_{def} p$$

TABLE 1. Operational semantics of CCS terms.

Before defining the relations $\approx$ and $=$, we need the following definitions of the relations $\xrightarrow{\alpha}$, $\stackrel{\alpha}{\Longrightarrow}$, and $\stackrel{\widehat{\alpha}}{\Longrightarrow}$, for any $\alpha \in Act$.

Let $t \in Act^*$ be any sequence of elements in Act, and ε be the empty sequence. By $\widehat{t}$ we denote the sequence obtained from t by erasing all τ's.

We have that $\widehat{\tau^n} = \varepsilon$ for any $n \geq 0$. For any $\alpha \in Act$ we have that: if $\alpha = \tau$ then $\widehat{\alpha} = \varepsilon$ else $\widehat{\alpha} = \alpha$.

By $(\xrightarrow{\tau})^*$ we denote the reflexive, transitive closure of $\xrightarrow{\tau}$, that is, for all processes P and Q, $P\,(\xrightarrow{\tau})^*\,Q$ iff there exists $n \geq 0$ such that $P\,(\xrightarrow{\tau})^n\,Q$.

Let us consider any sequence t of actions such that $t =_{def} \alpha_1 \ldots \alpha_n$, where $n \geq 0$ and some of the α_i's may be τ. Then,

(i) $P \xrightarrow{t} Q$ stands for $P \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} Q$,

(ii) $P \stackrel{t}{\Longrightarrow} Q$ stands for $P\,(\xrightarrow{\tau})^*\,\xrightarrow{\alpha_1}\,(\xrightarrow{\tau})^* \ldots (\xrightarrow{\tau})^*\,\xrightarrow{\alpha_n}\,(\xrightarrow{\tau})^*\,Q$, and

(iii) $P \stackrel{\widehat{t}}{\Longrightarrow} Q$ stands for $P\,(\xrightarrow{\tau})^*\,\xrightarrow{\beta_1}\,(\xrightarrow{\tau})^* \ldots (\xrightarrow{\tau})^*\,\xrightarrow{\beta_m}\,(\xrightarrow{\tau})^*\,Q$, where $\widehat{t} = \beta_1 \ldots \beta_m$, for some $0 \leq m \leq n$, is obtained from t by erasing all τ's.

Thus, for all processes P and Q, $P \stackrel{\varepsilon}{\Longrightarrow} Q$ iff $P\,(\stackrel{\tau}{\longrightarrow})^n\,Q$ for some $n \geq 0$. In particular, for every process P, we have that $P \stackrel{\varepsilon}{\Longrightarrow} P$.

If $P \stackrel{t}{\longrightarrow} Q$ we say that there is *a path from* (or *a derivative of*) P *to* Q labeled by *the action sequence* t (or simply, *the sequence* t) [**10**, pages 48–49].

DEFINITION 1.2. [**Bisimilarity** $\approx$] The relation $\approx$ is the *largest* relation satisfying the following property: for all processes P and Q,

$P \approx Q$ iff $\forall \alpha \in Act$. (i) $\forall P'$. if $P \stackrel{\alpha}{\longrightarrow} P'$ then $(\exists Q'.\, Q \stackrel{\widehat{\alpha}}{\Longrightarrow} Q'$ and $P' \approx Q')$ and
(ii) $\forall Q'$. if $Q \stackrel{\alpha}{\longrightarrow} Q'$ then $(\exists P'.\, P \stackrel{\widehat{\alpha}}{\Longrightarrow} P'$ and $P' \approx Q')$.

DEFINITION 1.3. [**Equality** $=$] The relation $=$ is the *largest* relation satisfying the following property: for all processes P and Q,

$P = Q$ iff $\forall \alpha \in Act$. (i) $\forall P'$. if $P \stackrel{\alpha}{\longrightarrow} P'$ then $(\exists Q'.\, Q \stackrel{\alpha}{\Longrightarrow} Q'$ and $P' \approx Q')$ and
(ii) $\forall Q'$. if $Q \stackrel{\alpha}{\longrightarrow} Q'$ then $(\exists P'.\, P \stackrel{\alpha}{\Longrightarrow} P'$ and $P' \approx Q')$.

If for any two processes P and Q we have that $P \approx Q$, we say that P and Q are *bisimilar*.

For any process P, we have that $P \approx \tau.P$ [**10**, page 111].

Let a CCS *context* $C[\_]$ be a CCS term C *without a subterm*. For instance,

(i) $\alpha.\_$,
(ii) $P+\_$,
(iii) $\_ \mid P$, and
(iv) $(\_ \mid P) + Q$

are CCS contexts. If $C[\_]$ is $P+\_$ we have that $C[Q]$ is $P+Q$. Note that $(\_|\_)$ is not a CCS context.

FACT 1.4. [**Bisimilarity $\approx$ is an Equivalence**] (i) The relation $\approx$ is an equivalence relation. (ii) $\approx$ is *not* a congruence, that is, there exist a context $C[\_]$ and two terms t_1 and t_2 such that $t_1 \approx t_2$ and $C[t_1] \not\approx C[t_2]$.

PROOF. For some distinct actions α and β in Act, we have that: (i) $\beta.0 \approx \tau.\beta.0$, and (ii) $\alpha.0 + \tau.\beta.0 \not\approx \alpha.0 + \beta.0$, as we now show. Property (i) holds because $\beta.0 \stackrel{\beta}{\longrightarrow} 0$ and $\tau.\beta.0 \stackrel{\widehat{\beta}}{\Longrightarrow} 0$ (because $\tau.\beta.0 \stackrel{\tau}{\longrightarrow} \stackrel{\beta}{\longrightarrow} 0$). Property (ii) holds because $\alpha.0+\tau.\beta.0 \stackrel{\tau}{\longrightarrow} \beta.0$ and $\alpha.0+\beta.0 \stackrel{\widehat{\tau}}{\Longrightarrow} \alpha.0+\beta.0$ (and obviously, $\beta.0 \not\approx \alpha.0+\beta.0$). □

FACT 1.5. [**Equality $=$ is a Congruence**] The relation $=$ is a congruence and it is the largest congruence contained in the equivalence $\approx$.

We have that $P \mid \tau.Q \approx P \mid Q$ (recall that $\widehat{\tau}$ is ε and for any P, $P \stackrel{\varepsilon}{\Longrightarrow} P$).

We also have that $P \mid \tau.Q \neq P \mid Q$ (indeed, $P \mid \tau.Q \stackrel{\tau}{\longrightarrow} P \mid Q$ and there is no agent R such that $P \mid Q \stackrel{\tau}{\Longrightarrow} R$). (Recall that $\stackrel{\tau}{\Longrightarrow}$ means $(\stackrel{\tau}{\longrightarrow})^* \stackrel{\tau}{\longrightarrow} (\stackrel{\tau}{\longrightarrow})^*$, that is, $(\stackrel{\tau}{\longrightarrow})^k$ with $k \geq 1$.)

The equivalence $\approx$ satisfies the following laws which hold for any process P, Q, and R, any action α and β, any set $L \subseteq A$ of names, and any renaming function f.

Monoid laws:

1. $P + Q \approx Q + P$
2. $(P + Q) + R \approx P + (Q + R)$
3. $P + P \approx P$
4. $P + 0 \approx P$

τ laws:

5. $P + \tau.P \approx \tau.P$
6. $\alpha.\tau.P \approx \alpha.P$
7. $\alpha.(P + \tau.Q) \approx \alpha.(P + \tau.Q) + \alpha.Q$

Restriction laws:

8. $0\backslash L \approx 0$
9. $(P + Q)\backslash L \approx P\backslash L + Q\backslash L$
10. $(\alpha.P)\backslash\{\beta\} \approx$ *if* $\alpha \in \{\beta, \overline{\beta}\}$ *then* 0 *else* $\alpha.(P\backslash\{\beta\})$

Renaming laws:

11. $0[f] \approx 0$
12. $(P + Q)[f] \approx P[f] + Q[f]$
13. $(\alpha.P)[f] \approx f(\alpha).(P[f])$

For any process P we have that: $0 \,|\, P \approx P \,|\, 0 \approx P$.

THEOREM 1.6. [**Expansion Theorem for $\approx$**] Let the process P be $\sum_{i\in I}\ \alpha_i.P_i$ and the process Q be $\sum_{j\in J}\ \beta_j.Q_j$. Then,

$$P \,|\, Q \ \approx\ \sum_{i\in I}\ \alpha_i.(P_i \,|\, Q) + \sum_{j\in J}\ \beta_j.(P \,|\, Q_j) + \sum_{i\in I,\, j\in J,\, \alpha_i=\overline{\beta}_j}\ \tau.(P_i \,|\, Q_j).$$

By this theorem, every $|$ (parallel composition) can be replaced by $+$ (sum). In particular, for every action α, β, α_1, α_2, β_1, and β_2 in *Act*, we have that:

(i) $\alpha.0 \,|\, \beta.0 \approx \alpha.\beta.0 + \beta.\alpha.0$

(ii) $\alpha_1.\alpha_2.0 \,|\, \beta_1.\beta_2.0 \approx \alpha_1.\alpha_2.\beta_1.\beta_2.0 + \alpha_1.\beta_1.\alpha_2.\beta_2.0 + \alpha_1.\beta_1.\beta_2.\alpha_2.0 + \beta_1.\alpha_1.\alpha_2.\beta_2.0 + \beta_1.\alpha_1.\beta_2.\alpha_2.0 + \beta_1.\beta_2.\alpha_1.\alpha_2.0$

(iii) $\alpha.0 \,|\, \overline{\alpha}.0 \,|\, \beta.0 \approx \tau.\beta.0 + \alpha.(\overline{\alpha}.\beta.0 + \beta.\overline{\alpha}.0) + \overline{\alpha}.(\alpha.\beta.0 + \beta.\alpha.0) + \beta.(\tau.0 + \alpha.\overline{\alpha}.0 + \overline{\alpha}.\alpha.0)$

Equivalences (i) and (ii) show that, if two processes cannot perform a τ action together, then their parallel composition behaves as the *interleaving* of their actions in the sense specified by the following definition.

DEFINITION 1.7. [**Interleaving of Sequences**] An interleaving of two sequences $t_1 =_{def} \langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle$ and $t_2 =_{def} \langle \beta_1, \beta_2, \ldots, \beta_m \rangle$ is the sequence $t =_{def} \langle \gamma_1, \gamma_2, \ldots, \gamma_{n+m} \rangle$ such that: (i) for $i = 1, \ldots, n+m$, γ_i is either an element of t_1 or t_2, (ii) if we erase all the α_i's from t we get t_2, and (iii) if we erase all the β_i's from t we get t_1.

EXERCISE 1.8. Show that for every process P and Q,
(i) $0 \,|\, P \approx P$, (ii) $P + \tau.(P + Q) \approx \tau.(P + Q)$, and (iii) $P \,|\, \tau.Q \approx P \,|\, Q$.
Solution of (ii). $P + \tau.(P + Q) \approx P + (P + Q) + \tau.(P + Q) \approx P + Q + \tau.(P + Q) \approx \tau.(P + Q)$.

We say that A is *stable* iff A has no τ-derivatives.

We have the following results [**10**, pages 112 and 156].

THEOREM 1.9. [**Relating Bisimilarity $\approx$ and Equality $=$**] Let us consider any two processes P and Q.
(i) If $P \approx Q$ and P and Q are both stable, then $P = Q$.
(ii) If $P \approx Q$ then for any $\alpha \in Act$, $\alpha.P = \alpha.Q$.
(iii) $P \approx Q$ iff ($P = Q$ or $P = \tau.Q$ or $\tau.P = Q$).

DEFINITION 1.10. [**Finite Process**] A process P is said to be *finite* iff it is of the form: $P ::= 0 \mid \alpha.P \mid \sum_{i \in I} P_i$ and I is a finite set.

In Table 2 we list a system of axioms for $=$. These axioms hold for any process P, Q, and R, and any action α, and are *complete* for finite processes, in the sense that these axioms are sufficient for showing all equalities between finite processes.

By Theorem 1.9 (iii) the axioms of Table 2 are a sound and complete axiom system for establishing, for any two finite processes P and Q, whether or not $P \approx Q$.

The Monoid laws and the τ laws are analogous to those for the bisimulation equivalence we have listed above.

Monoid laws:
1. $P + Q = Q + P$
2. $(P + Q) + R = P + (Q + R)$
3. $P + P = P$
4. $P + 0 = P$

τ laws:
5. $P + \tau.P = \tau.P$
6. $\alpha.\tau.P = \alpha.P$
7. $\alpha.(P + \tau.Q) = \alpha.(P + \tau.Q) + \alpha.Q$

TABLE 2. A sound and complete axiom system for establishing for any two given finite CCS processes P and Q, whether or not $P = Q$. By Theorem 1.9 (iii) on page 268 this is a sound and complete axiom system also for establishing whether or not $P \approx Q$.

We also have the following equalities which hold for any process P and Q, any action α and β, any set $L \subseteq A$ of names, and any renaming function f.

Restriction laws:
8. $0\backslash L = 0$
9. $(P + Q)\backslash L = P\backslash L + Q\backslash L$
10. $(\alpha.P)\backslash\{\beta\} = \mathit{if}\ \alpha \in \{\beta, \overline{\beta}\}\ \mathit{then}\ 0\ \mathit{else}\ \alpha.(P\backslash\{\beta\})$

Renaming laws:
11. $0[f] = 0$
12. $(P + Q)[f] = P[f] + Q[f]$
13. $(\alpha.P)[f] = f(\alpha).(P[f])$

The following theorem is analogous to Theorem 1.6 on the previous page.

THEOREM 1.11. **[Expansion Theorem for $=$]** Let the process P be $\sum_{i\in I}\ \alpha_i.P_i$ and the process Q be $\sum_{j\in J}\ \beta_j.Q_j$. Then,

$P\,|\,Q\ =\ \sum_{i\in I}\ \alpha_i.(P_i\,|\,Q)+\sum_{j\in J}\ \beta_j.(P\,|\,Q_j)+\sum_{i\in I,\,j\in J,\,\alpha_i=\overline{\beta}_j}\ \tau.(P_i\,|\,Q_j)$.

FACT 1.12. For any process P, Q, and R, any action α, any set L of names, and any renaming function f, if $P=Q$ we have that:
(i) $\alpha.P=\alpha.Q$,
(ii) $P+R=Q+R$,
(iii) $P\,|\,R=Q\,|\,R$,
(iv) $P\backslash L=Q\backslash L$, and
(v) $P[f]=Q[f]$.

EXERCISE 1.13. Show that for every process P and Q,
(i) $0\,|\,P=P$,
(ii) $P+\tau.(P+Q)=\tau.(P+Q)$, and
(iii) $P\,|\,\tau.Q\neq P\,|\,Q$

The following two facts establish the uniqueness of solutions of recursively defined processes (see **[10**, page 157] and **[11**, page 59]).

First we need the following definitions.

Let us consider a countable set of variables $Vars=\{X_1,X_2,\ldots\}$.

Let E be any expression constructed as a process (see Point (vi) on page 263) by allowing also variables in $Vars$. Let $E[P_1,\ldots,P_n]$ denote the expression E whose free variables are included in $\{X_1,\ldots,X_n\}$ and substituted by $P_1,\ldots,P_n$, respectively. Thus, $E[P_1,\ldots,P_n]$ stands for $E[P_1/X_1,\ldots,P_n/X_n]$.

DEFINITION 1.14. **[Guarded Process and Sequential Process]** A process X is said to be *guarded in an expression* E iff every occurrence of X is within some subexpressions of E of the form $\ell.F$, with $\ell\in Act-\{\tau\}$.

A process X is said to be *sequential in an expression* E iff every subexpression of E which contains X, apart from X itself, is of the form either $\alpha.F$, with $\alpha\in Act$, or $\sum_{i\in I}\ F_i$, where I is a finite set.

The process X is sequential but not guarded in $\tau.X+\alpha.0$. The process X is guarded but not sequential in $\alpha.X\,|\,\beta.0$. The process X is both sequential and guarded in $\tau(\alpha.(P\,|\,Q)+\beta.X)$.

FACT 1.15. **[System of Processes for $\approx$]** Consider a system of processes of the form:

$$\begin{cases} P_1 \ \approx\ \ell_{11}P_1+\ldots+\ell_{1n}P_n \\ \quad\ \vdots \\ P_n \ \approx\ \ell_{n1}P_1+\ldots+\ell_{nn}P_n \end{cases} \qquad (S\approx)$$

where: (i) some of the sommands may be absent, and (ii) for $i,j\in\{1,\ldots,n\}$, ℓ_{ij} is an action in $Act-\{\tau\}$. We have that there exists a unique solution up to $\approx$ for $P_1,\ldots,P_n$. That is, if processes $P_1,\ldots,P_n$ satisfy the system $(S\approx)$ and also the processes $Q_1,\ldots,Q_n$ satisfy the system $(S\approx)$ then for $i=1,\ldots,n$, we have that $P_i\approx Q_i$.

The system $P \approx \tau.P$ has more than one solution. Actually any process P is a solution of that system. In particular, given any two distinct actions α and β, we have that: (i) $\alpha.0 \approx \tau.\alpha.0$, (ii) $\beta.0 \approx \tau.\beta.0$, and (iii) $\alpha.0 \not\approx \beta.0$.

FACT 1.16. **[System of Processes for $=$]** Consider a system of processes of the form:

$$\begin{cases} P_1 &= E_1[P_1, \ldots, P_n] \\ & \vdots \\ P_n &= E_n[P_1, \ldots, P_n] \end{cases} \qquad (S=)$$

We have that if for $i, j \in \{1, \ldots, n\}$, every P_i is *guarded* and *sequential in every* E_j, then there exists a unique solution up to $=$ for $P_1, \ldots, P_n$. That is, if processes $P_1, \ldots, P_n$ satisfy the system $(S=)$ and also the processes $Q_1, \ldots, Q_n$ satisfy the system $(S=)$ then for $i=1, \ldots, n$, we have that $P_i = Q_i$.

The relations $\approx$ and $=$ can be used for proving properties of parallel programs after encoding them as CCS processes. We will not illustrate in detail this approach here and the interested reader may refer to [**15**, Chapter 6].

The following example will suffice.

In order to show that a given protocol P ensures *mutual exclusion* between two given processes P_1 and P_2 when they want to enter a critical section, it will enough to show that the parallel composition of the protocol P and the processes P_1 and P_2 is bisimilar to the process R defined as follows:

$$R =_{def} in.out.R \qquad (R)$$

where the *in* and *out* actions are performed by a process when entering and exiting, respectively, the critical section. Note that bisimilarity is established by observing the actions *in* and *out* only.

Obviously, we have that $R \xrightarrow{in} \xrightarrow{out} R$ and, thus, the behaviour of R is the infinite sequence $(in\ out)^\omega$ of actions where between any two *in* actions there is always an *out* action.

This guarantees that the processes P_1 and P_2 can never be in the critical section at the same time. Indeed, for those processes to be together in the critical section, it is necessary that, after an *in* action has been performed by a process, say P_1, the other process P_2 performs an *in* action, before P_1 performs the subsequent *out* action, thereby exiting the critical section.

Now we will prove that Peterson algorithm ensures mutual exclusion by applying the technique for proving properties of parallel programs that is described in [**15**, Chapter 6]. As already mentioned, that proof technique is based on the bisimilation equivalence.

Let us consider the following two processes:

process P_1:

while *true* **do**
    non-critical section 1;
    await *test* 1;
    critical section 1;
    od

process P_2:

while *true* **do**
    non-critical section 2;
    await *test* 2;
    critical section 2;
    od

which are assumed to run in parallel. When *test* 1 succeeds process P_1 enters the critical section 1, likewise when *test* 2 succeeds process P_2 enters the critical section 2. Peterson algorithm consists in suitably modifying processes P_1 and P_2 by adding some tests and assignments involving three extra variables: two of those variables are boolean variables and the third one is an integer variable that may assume the values 1 and 2. Let the boolean variables be q_1 and q_2, and the integer variable be s.

The modified processes P_1 and P_2 are as follows.

process P_1:

while *true* **do**
non-critical section 1;
$q_1 := \mathit{true}$; $s := 1$;
await $(\neg q_2) \vee (s = 2)$;
critical section 1;
$q_1 := \mathit{false}$; **od**

process P_2:

while *true* **do**
non-critical section 2;
$q_2 := \mathit{true}$; $s := 2$;
await $(\neg q_1) \vee (s = 1)$;
critical section 2;
$q_2 := \mathit{false}$; **od**

We can encode the two processes and the three variables by using CCS processes as follows [**15**, page 90]. For the variable q_1 we have:

$$\begin{array}{lll} Q1t & =_{def} & \overline{q1rt}.Q1t \; + q1wt.Q1t \; + q1wf.Q1f \qquad (q_1 \; \mathit{true}) \\ Q1f & =_{def} & \overline{q1rf}.Q1f \; + q1wt.Q1t \; + q1wf.Q1f \qquad (q_1 \; \mathit{false}) \end{array}$$

For the variable q_2 we have:

$$\begin{array}{lll} Q2t & =_{def} & \overline{q2rt}.Q2t \; + q2wt.Q2t \; + q2wf.Q2f \qquad (q_2 \; \mathit{true}) \\ Q2f & =_{def} & \overline{q2rf}.Q2f \; + q2wt.Q2t \; + q2wf.Q2f \qquad (q_2 \; \mathit{false}) \end{array}$$

For the variable s we have:

$$\begin{array}{lll} S1 & =_{def} & \overline{r1}.S1 \; + \; w1.S1 \; + \; w2.S2 \qquad (s{=}1) \\ S2 & =_{def} & \overline{r2}.S2 \; + \; w1.S1 \; + \; w2.S2 \qquad (s{=}2) \end{array}$$

For the process P_1 we have the following three equations:

$$\begin{cases} P1 & =_{def} & \overline{q1wt}.\overline{w1}.P11 \\ P11 & =_{def} & q2rt.P11 \; + \; r1.P11 \; + \; q2rf.P12 \; + \; r2.P12 \\ P12 & =_{def} & in.out.\overline{q1wf}.P1 \end{cases} \qquad (P_1)$$

For the process P_2 we have the following three equations:

$$\begin{cases} P2 & =_{def} & \overline{q2wt}.\overline{w2}.P21 \\ P21 & =_{def} & q1rt.P21 \; + \; r2.P21 \; + \; q1rf.P22 \; + \; r1.P22 \\ P22 & =_{def} & in.out.\overline{q2wf}.P2 \end{cases} \qquad (P_2)$$

The initial value of q_1 and q_2 is assumed to be *false* and we take the initial value of s to be 1. (Nothing changes if we take the initial value of s to be 2.)

In the above CCS definitions of the process P_1 and P_2, the critical sections are encoded by the two sequence of actions '$in.out$', while the non-critical sections are not encoded. Indeed, in order to prove the mutual exclusion property of Peterson's algorithm, there is no need to encode the non-critical sections and, as the reader may convince himself, these non-critical sections may be incorporated in the actions $\overline{q1wt}$ and $\overline{q2wt}$, respectively.

The fact that Peterson algorithm satisfies the mutual exclusion property can be established by proving that the compound CCS process

$$(P1 \mid P2 \mid Q1f \mid Q2f \mid S1)\backslash L$$

where $L = \{q1rt, q1rf, q1wt, q1wf, q2rt, q2rf, q2wt, q2wf, r1, r2, w1, w2\}$, is bisimilar to the process R (see Equation (R) on page 270).

This proof can be done through an automatic system, the Edinburgh Concurrency Workbench [**13**]. The following program shows the use of that system. For any two CCS terms `P` and `Q`, the command `eq(P,Q)` returns `true` iff $P \approx Q$.

Lines preceded by the character `*` are comments. Anything between a `*` and the next newline character is also a comment.

```
* ========================================================================
*                    PETERSON ALGORITHM AND BISIMULATION
*
* filename: peterson-bisimulation.cwb
* Use of the Edinburgh Concurrency Workbench.
* ------------------------------------------------------------------------
*** variable q1
agent Q1t  = 'q1rt.Q1t + q1wt.Q1t + q1wf.Q1f;  * q1 true
agent Q1f  = 'q1rf.Q1f + q1wt.Q1t + q1wf.Q1f;  * q1 false
*** variable q2
agent Q2t  = 'q2rt.Q2t + q2wt.Q2t + q2wf.Q2f;  * q2 true
agent Q2f  = 'q2rf.Q2f + q2wt.Q2t + q2wf.Q2f;  * q2 false
*** variable S
agent S1  = 'r1.S1 + w1.S1 + w2.S2;            * s = 1
agent S2  = 'r2.S2 + w1.S1 + w2.S2;            * s = 2
*** process P1
agent P1  = 'q1wt.'w1.P11;
agent P11 = q2rt.P11 + r1.P11 + q2rf.P12 + r2.P12;
agent P12 = in.out.'q1wf.P1;
*** process P2
agent P2  = 'q2wt.'w2.P21;
agent P21 = q1rt.P21 + r2.P21 + q1rf.P22 + r1.P22;
agent P22 = in.out.'q2wf.P2;
* -----------
agent Peterson = (P1|P2|Q1f|Q2f|S1)\L;
set L = {q1rt,q1rf,q1wt,q1wf,q2rt,q2rf,q2wt,q2wf,r1,r2,w1,w2};
* -----------
agent R = in.out.R;
* ------------------------------------------------------------------------
*                           Mutual Exclusion
* ------------------------------------------------------------------------
* cwb
* input "peterson-bisimulation.cwb";
* eq(Peterson,R);   <--- bisimulation between agents Peterson and R: true
* true;
* ========================================================================
```

Since the process R generates the infinite sequence $(in\ out)^\omega$, that proof also shows that Peterson algorithm ensures *absence of deadlock*, that is, it is not the case that both processes cannot proceed. More details on this point may be found in [**15**, Chapter 6].

Actually, one can show that Peterson algorithm ensures (see the proofs using the Edinburgh Concurrency Workbench system [**13**] on page 290):

(1) *mutual exclusion*,
(2) *absence of deadlock*,
(3) *absence of starvation*, and
(4) *bounded overtaking*.

Absence of starvation means that every process is at the **await** statement only a finite amount of time, and bounded overtaking means that if a process is at the **await** statement, the other process can exit the critical section at most once (and this implies that can enter its critical section at most once). Again, more details can be found in [**15**, Chapter 6].

As mentioned at the beginning of this Chapter, the proofs of these properties can also be done by applying the following general three step procedure which we will illustrate in the following Sections 3 and 4.

Step (1). We encode the program under consideration as a CCS process, say p (as we have done above in the case of the Peterson algorithm).

Step (2). We encode the property to be shown as a formula, say φ, of a calculus, called modal μ-calculus (see Section 3 on page 275).

Step (3). We apply an algorithm for showing when the process p satisfies the modal μ-calculus formula φ (see Section 4 on page 283).

Obviously, since in general the problem of deciding whether or not a property holds for a process may be unsolvable, the algorithm of Step (iii) is not guaranteed to terminate in all cases with a yes/no answer.

2. The Hennessy-Milner Logic

In this section we present the Hennessy-Milner logic which characterizes a particular notion of bisimilarity, called *strong bisimilarity*, between CCS terms which we now define.

Let us consider the set Act of actions defined as indicated in Section 1 on page 263 and a set $\mathcal{P}$ of CCS processes.

DEFINITION 2.1. [**Strong Bisimilarity** $\sim$] The relation $\sim$, called *strong bisimilarity*, is the *largest* relation satisfying the following property:
for all CCS processes p and q,

$$p \sim q \text{ iff } \forall a \in Act.\ \text{(i)}\ \ \forall p'.\ \text{if } p \xrightarrow{a} p' \text{ then } (\exists q'.\, q \xrightarrow{a} q' \text{ and } p' \sim q') \text{ and}$$
$$\text{(ii)}\ \forall q'.\ \text{if } q \xrightarrow{a} q' \text{ then } (\exists p'.\, p \xrightarrow{a} p' \text{ and } P' \sim q').$$

If for any two processes p and q we have that $p \sim q$, we say that p and q are *strongly bisimilar*.

Now let us introduce the the Hennessy-Milner logic.

The syntax of an assertion φ of the Hennessy-Milner logic is defined as follows. The symbol a stands for any action in Act.

$$\varphi \ ::= \ \mathbf{true} \ \mid \ \mathbf{false} \ \mid \ \neg\varphi \ \mid \ \bigwedge_{i\in I} \varphi_i \ \mid \ \bigvee_{i\in I} \varphi_i \ \mid$$
$$\langle a\rangle\, \varphi \ \mid \ [a]\, \varphi$$

where I is an index set. Note that the set I may be empty or infinite.

As usual, we have that **true** is $\bigwedge_{i\in\emptyset}\varphi_i$ and **false** is $\bigvee_{i\in\emptyset}\varphi_i$.

The semantics of an assertion φ of the Hennessy-Milner logic is defined as follows via a satisfaction relation $p \models \varphi$, where p is any CCS process:

$p \models \mathbf{true}$

$p \not\models \mathbf{false}$

$p \models \neg\varphi$ if $p \not\models \varphi$

$p \models \bigwedge_{i\in I}\varphi_i$ if for all $i \in I$, $p \models \varphi_i$

$p \models \bigvee_{i\in I}\varphi_i$ if there exists $i \in I$ such that $p \models \varphi_i$

$p \models \langle a\rangle\,\varphi$ if $\exists\, q{\in}\mathcal{P}.\ p \xrightarrow{a} q \wedge q \models \varphi$

$p \models [a]\,\varphi$ if $\forall\, q{\in}\mathcal{P}.\ p \xrightarrow{a} q \Rightarrow q \models \varphi$

Note, in particular, that $p \models [a]\,\varphi$ holds for a process p if there is no process q such that $p \xrightarrow{a} q$ and $q \not\models \varphi$. Thus, $p \models [a]\,\mathbf{false}$ means that process p *cannot* perform an a action.

We state without proof the following theorem which relates the satisfaction relation in the Hennessy-Milner logic to strong bisimilarity.

THEOREM 2.2. [**Strong Bisimilarity and Hennessy-Milner Logic**] Let us consider a set *Act* of actions. Given any two CCS processes p and q constructed by using the set *Act* of actions,

$p \sim q$ *iff* for all Hennessy-Milner assertions φ constructed by using the set *Act* of actions, we have that $p \models \varphi$ iff $q \models \varphi$.

EXAMPLE 2.3. Given the process $p =_{def} a.0 + b.a.0$ we have that:
$p \models \langle a\rangle\,\mathbf{true} \wedge \langle b\rangle\,\langle a\rangle\,\mathbf{true}$. (Note $\wedge$ and not $\vee$.) □

We have the following result [**10**, Proposition 8, page 112]: for all processes p and q,

(i) $p{\sim}q$ implies $p{=}q$, and

(ii) $p{=}q$ implies $p{\approx}q$.

The laws of Table 3 provide a sound and complete axiom system for establishing for any two given *finite* CCS processes p and q, whether or not p and q are strongly bisimilar, that is, whether or not $p{\sim}q$ holds.

Monoid laws:

1. $p + q \sim q + p$
2. $(p + q) + r \sim p + (q + r)$
3. $p + p \sim p$
4. $p + 0 \sim p$

TABLE 3. A sound and complete axiom system for establishing for any two given *finite* CCS processes p and q, whether or not $p{\sim}q$.

3. The Modal μ-Calculus

In this section we present the modal μ-calculus, also called the ν-calculus, when one wants to stress the role of the maximal fixpoints, rather than the minimal fixpoints. This calculus is an extension of the Hennessy-Milner logic we have introduced in the previous section.

Let us consider the set *Act* of actions defined as indicated in Section 1 on page 263 and a set $\mathcal{P}$ of CCS processes. Let us also consider a countable set *Vars* of variables.

The syntax of an *assertion* (or a *property*, or a *formula*) φ of the modal μ-calculus is as follows.

$$\begin{array}{rcl} \varphi & ::= & \mathbf{true} \mid \mathbf{false} \mid S \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \\ & & \langle a \rangle \varphi \mid \langle \mathbf{.} \rangle \varphi \mid [a]\varphi \mid [\mathbf{.}]\varphi \mid \\ & & \nu X.\varphi \mid \mu X.\varphi \end{array}$$

The symbol S stands for any subset of $\mathcal{P}$. The symbol a stands for any action in *Act*. The operators νX and μX, for any variable $X \in \mathit{Vars}$, behave as binders. As usual in the case of binders, we have that every occurrence of a variable is either a *bound occurrence* or a *free occurrence*. (We do not formally define these concepts here and we assume that the reader is already familiar with them.)

Any assertion of the form $\mu X.\varphi$ should be regarded as an abbreviation for the assertion $\neg\nu X.\neg\varphi[\neg X/X]$. (Note that φ has to be replaced by $\varphi[\neg X/X]$.) As usual, by $\varphi\,[\psi/X]$ we denote the assertion φ where the free occurrences of the variable X (that is, the occurrences which are not in the scope of a binder) have been replaced by the assertion ψ.

The relationship between $\nu X.\varphi$ and $\mu X.\varphi$ is based on the semantics of the modal μ-calculus assertions which we will give in Table 4 on the following page (see also Exercise 3.2 on page 279). Thus, according to this definition, the semantics of $\nu X.\varphi$ is equal to the semantics of $\neg\mu X.\neg\varphi[\neg X/X]$.

To be syntactically correct an assertion φ of the modal μ-calculus should satisfy the following condition, assuming that every subassertion of the form $\mu X.\psi$ in φ has been replaced by $\neg\nu X.\neg\psi[\neg X/X]$:

(*Pos*) in every subassertion of the form $\nu X.\psi$, each free occurrence of the variable X in ψ is a *positive occurrence*, that is, it is an occurrence under an even, possibly zero, number of negation symbols.

For instance, given any two subsets S_1 and S_2 of $\mathcal{P}$, we have that the assertion $\nu X.\neg(S_1 \vee (S_2 \wedge \neg[\mathbf{.}]X))$ is syntactically correct, while the assertion $\nu X.(S_1 \vee (S_2 \wedge \neg[\mathbf{.}]X))$ is not, because it does not comply with Condition (*Pos*) above.

Given a variable X and an assertion φ, the assertion $\nu X.\varphi$ is said to be the *maximal fixpoint* (or the *greatest fixpoint*) of φ with respect to the variable X, and the assertion $\mu X.\varphi$ is said to be the *minimal fixpoint* (or the *least fixpoint*) of φ with respect to the variable X. This terminology follows from the fact that the semantics of an assertion is specified as a set of processes and $\nu X.\varphi$ denotes the maximal set X of processes which is a solution of the equation $X = \varphi$. Similarly, the minimal fixpoint $\mu X.\varphi$ denotes the minimal set X of processes which is a solution of the equation $X = \varphi$.

Syntax	Semantics (as a subset of $\mathcal{P}$)
true	$\mathcal{P}$
false	$\emptyset$
S	S for any subset $S \subseteq \mathcal{P}$
$\neg\,\varphi$	$\{p \mid p \notin \varphi\}$, that is, $\{p \mid p \in (\mathcal{P} - \varphi)\}$
$\varphi_1 \wedge \varphi_2$	$\varphi_1 \cap \varphi_2$
$\varphi_1 \vee \varphi_2$	$\varphi_1 \cup \varphi_2$
$\langle a \rangle\,\varphi$	$\{p \mid \exists\, q \in \mathcal{P}.\ p \xrightarrow{a} q \ \wedge\ q \in \varphi\}$ for any $a \in Act$
$\langle \,\mathbf{.}\, \rangle\,\varphi$	$\{p \mid \exists\, a \in Act.\ \exists\, q \in \mathcal{P}.\ p \xrightarrow{a} q \ \wedge\ q \in \varphi\}$
$[a]\,\varphi$	$\{p \mid \forall\, q \in \mathcal{P}.\ p \xrightarrow{a} q \ \Rightarrow\ q \in \varphi\}$ for any $a \in Act$
$[\,\mathbf{.}\,]\,\varphi$	$\{p \mid \forall\, a \in Act.\ \forall\, q \in \mathcal{P}.\ p \xrightarrow{a} q \ \Rightarrow\ q \in \varphi\}$
$\nu X.\varphi$	$\bigcup \{S \subseteq \mathcal{P} \mid S \subseteq \varphi[S/X]\}$
$\mu X.\varphi$	$\bigcap \{S \subseteq \mathcal{P} \mid \varphi[S/X] \subseteq S\}$

TABLE 4. The syntax and the semantics of the assertions of the modal μ-calculus. The set $\mathcal{P}$ is a given set of CCS processes and the set Act is a given set of actions.

Here are some notes on Table 4 which defines the semantics of the assertions of the modal μ-calculus as sets of processes.

(1) First, for any set S of processes, the semantics of the assertion S is identified with S itself.

(2) The assertion $\langle a \rangle\,\varphi$ with the modality $\langle a \rangle$, called '*diamond a*', holds for all processes p in $\mathcal{P}$ such that p can do an a action and become a process which satisfies φ.

(3) The assertion $\langle \,\mathbf{.}\, \rangle\,\varphi$ with the modality $\langle \,\mathbf{.}\, \rangle$, called '*diamond dot*', holds for all processes p in $\mathcal{P}$ which can do any action in Act and become a process which satisfies φ.

(4) Besides the modalities $\langle a \rangle$ and $\langle \,\mathbf{.}\, \rangle$, there are the following two modalities: (i) $[a]$, called '*box a*', and (ii) $[\,\mathbf{.}\,]$, called '*box dot*'. They are dual modalities w.r.t. $\langle a \rangle$ and $\langle \,\mathbf{.}\, \rangle$. Indeed, $[a]\,\varphi$ and $[\,\mathbf{.}\,]\,\varphi$ can be viewed as abbreviations for $\neg\langle a \rangle \neg\varphi$ and $\neg\langle \,\mathbf{.}\, \rangle \neg\varphi$, respectively.

Thus, $[a]\,\varphi$ holds for all processes p in $\mathcal{P}$ such that *either* p cannot do an a action *or* if p can do an a action, then in every way p can do it, p becomes a process satisfying φ. The assertion $[\,\mathbf{.}\,]\,\varphi$ holds for all processes p in $\mathcal{P}$ such that *either* p cannot do any action at all *or* if p can do an action, then in every way p can do an action, p becomes a process satisfying φ. Thus, every action, if any, which a process p satisfying $[\,\mathbf{.}\,]\,\varphi$ can do, leads from p to a process satisfying φ.

The assertion $[a]\,\mathbf{false}$ holds for all processes which cannot do an a action. The assertion $[\,\mathbf{.}\,]\,\mathbf{false}$ holds for all processes which cannot do any action at all.

(5) According to the assumption that the variable X occurs positively in the assertion φ (see Condition (*Pos*) on page 275), the function $\lambda S \in \mathcal{P}.\,\varphi[S/X]$ is monotonic and, by Knaster-Tarski Theorem (see Theorem 3.2 on page 81) it has a maximal and a minimal fixpoint in the lattice of all subsets of $\mathcal{P}$.

The maximal fixpoint of $\lambda S \in \mathcal{P}.\, \varphi[S/X]$ is least upper bound of all its postfixpoints and, thus, we have that the semantics of $\nu X.\varphi$ is $\bigcup\{S \subseteq \mathcal{P} \mid S \subseteq \varphi[S/X]\}$.

The minimal fixpoint of $\lambda S \in \mathcal{P}.\, \varphi[S/X]$ is greatest lower bound of all its prefixpoints and, thus, we have that the semantics of $\mu X.\varphi$ is $\bigcap\{S \subseteq \mathcal{P} \mid \varphi[S/X] \subseteq S\}$.

REMARK 3.1. Since the syntax and the semantics of processes may be identified (see Point (1) on page 276), by abuse of language, we will feel free to write $\nu X.\varphi = \bigcup\{S \subseteq \mathcal{P} \mid S \subseteq \varphi[S/X]\}$. Analogously, for the minimal fixpoint $\mu X.\varphi$, instead of the maximal fixpoint $\nu X.\varphi$.

In general, we will feel free to view assertions as sets of processes and, dually, sets of processes as assertions. Thus, when the operands of the operators $\neg$, $\vee$, and $\wedge$ are sets, rather than assertions, we assume that they denote, respectively, complement (with respect to a given finite set $\mathcal{P}$ of processes to be understood from the context), union, and intersection of sets. □

3.1. Solutions of Language Equations.

In order to better understand to meaning of maximal and minimal fixpoints, in this section we consider equations between sets of words over a given alphabet Σ and we provide their solutions in terms of maximal and minimal fixpoints (see also **[14]**).

We assume that the reader is familiar with the notion of a regular expression over an *alphabet* Σ. The elements of Σ are called *symbols*.

Given a set A of symbols, with $A \subseteq \Sigma$, the set A^*, called the *star closure* of A, is the set of all *finite* words made out of symbols of A. The set A^* is a monoid w.r.t. the concatenation operation. As usual, given two words w_1 and w_2 in A^*, by $w_1\, w_2$ we denote their concatenation, that is, the symbols of w_1 followed by the symbols of w_2. The identity element of the concatenation is the *empty word* ε made out of no symbols.

Let Σ^∞ denote the set of all finite or infinite words made out of symbols in Σ. Let us consider two sets A and B of (finite or infinite) words made out of symbols in Σ.

Given two sets A and B of words in Σ^∞, with $B \neq \emptyset$, we defined their *concatenation*, denoted $A \cdot B$ (or $A\,B$, for short), to be the set

$$A \cdot B =_{def} \{w_A\, w_B \mid w_A \in A \cap \Sigma^* \text{ and } w_B \in B\} \cup (A \cap \Sigma^\omega)$$

where $w_A\, w_B$ denotes the concatenation of the finite words w_A and w_B. Note that the set $A \cap \Sigma^\omega$ is the set of the infinite words of A.

We stipulate that if $B = \emptyset$, then for all sets A, $A \cdot B = \emptyset$. As a consequence of the definition, we have that if $A = \emptyset$ then $A \cdot B = \emptyset$.

Note that if a word w is infinite then for all $v \in \Sigma^\infty$, we have that $w\, v = w$. Thus, if we concatenate a word v to the right of an infinite word w, we get again the infinite word w. Thus, all the infinite words in A belong to $A \cdot B$.

Let A^i denote the set $A \cdot A \cdot \ldots \cdot A$, where A occurs i times. For any set A, let A^0 denote the set $\{\varepsilon\}$ consisting of the empty word ε only. Let $A + B$ denote the union of the sets A and B.

We have that: $A^* = \sum_{i\geq 0} A^i = \{\varepsilon\} \cup A \cup A^2 \cup \ldots \cup A^i \cup \ldots$ Thus, if $A = \emptyset$ then $A^* = \{\varepsilon\}$. We defined the set A^ω as follows:

$$\begin{aligned} A^\omega &=_{def} \Sigma^\infty && \textit{if } \varepsilon \in A \\ &=_{def} \{w_0 w_1 \ldots w_i \ldots \mid i \in \omega \text{ and } w_i \in A \cap \Sigma^*\} \cup (A^* \cap \Sigma^\omega) && \textit{otherwise} \end{aligned}$$

where: (1) $w_0 w_1 \ldots w_i \ldots$, with $i \in \omega$ and $w_i \in A \cap \Sigma^*$, is an infinite concatenation of finite words, each of which is in A, and (2) the set $A^* \cap \Sigma^\omega$ is the set of the infinite words in A^*. Note that any infinite word in A^* is the finite concatenation of $n\,(\geq 1)$ words in A such that the first $n-1$ words are finite words in A and the last one is an infinite word in A. (Obviously, the concatenation of $n\,(\geq 0)$ finite words in A followed by one infinite word in A, is a word in $A^* \cap \Sigma^\omega$.)

If $A = \emptyset$ then $A^\omega = \emptyset$.

By definition, the infinite words of A^* belong to the infinite words of A^ω. Below we will explain why we stipulate that if $\varepsilon \in A$ then A^ω is Σ^∞.

We define A^∞ to be $A^* \cup A^\omega$.

We have the following equalities between sets of (finite or infinite) words:

(i) $A^\omega = A\,A^\omega = A^\omega A$

(ii) if $\varepsilon \in A$ then $A^\omega = A^\infty = \Sigma^\infty$.

The proof of Point (i) is left to the reader. For Point (ii) we have that if $\varepsilon \in A$ then $A^\omega = A^\infty$. Indeed, every word w in A^* also belongs to A^ω because it can be obtained by concatenating a finite number of words in A and then, infinitely many times, the finite word ε (and so, every word in A^* can be viewed as the result of infinite concatenations). The proof that if $\varepsilon \in A$ then $A^\infty = \Sigma^\infty$ will be given below.

Let us consider the following two equations and their solutions. Let A and B be two given sets of (finite or infinite) words made out of symbols in Σ.

(1) Equation $E1$: $X = A\,X + B$.

The minimal solution for X, denoted $\mu X.A\,X + B$, of Equation $E1$ is the set A^*B (recall the Arden rule for the equations between regular expressions). The proof of this fact can be done by applying the Kleene Theorem by iterated applications of the function $\lambda X.A\,X + B$, starting from the empty set. The set A^*B is the unique solution of Equation $E1$ if $\varepsilon \notin A$.

The maximal solution for X, denoted $\nu X.A\,X + B$, of Equation $E1$ is the set $A^*B + A^\omega$.

Here is the proof that $\nu X.A\,X + B = A^*B + A^\omega$. Starting from Σ^∞, by iterated application of the function $\lambda X.A\,X + B$, we get the following sequence of sets of words:

$\Sigma^\infty,\ A\,\Sigma^\infty + B,\ A^2\,\Sigma^\infty + A\,B + B,\ A^3\,\Sigma^\infty + A^2\,B + A\,B + B,\ \ldots$

whose limit point is $A^*B + A^\omega$ because, as the reader may show, $A^*B + A^\omega = A\,(A^*B + A^\omega) + B$ and for any other solution Z of Equation $E1$, we have that $Z \subseteq A^*B + A^\omega$.

If we assume that $B = \{\varepsilon\}$ then

(1.1) $\mu X.A\,X + B = A^*$, and

(1.2) $\nu X.A\,X + B = A^* + A^\omega = A^\infty$.

(2) Equation $E2$: $X = A\,X$.

The maximal solution for X , denoted $\nu X.A\,X$, of Equation $E2$ is the set A^ω. If we assume that $\varepsilon \in A$ then $\nu X.A\,X = \Sigma^\infty$. Actually, as already indicated, if $\varepsilon \in A$ then $\Sigma^\infty = A^\omega = A^\infty$.

We have that if $\varepsilon \in A$ then $A^\omega = \Sigma^\infty$, because if $\varepsilon \in A$ the maximal fixpoint A^ω of the function $\lambda X.A\,X$ is equal to Σ^∞. Indeed, if $\varepsilon \in A$ we have that $(\lambda X.A\,X)(\Sigma^\infty) = \Sigma^\infty$, and thus, for every ordinal α, $(\lambda X.A\,X)^\alpha(\Sigma^\infty) = \Sigma^\infty$.

As a consequence of the fact that if $\varepsilon \in A$ then $A^\omega = \Sigma^\infty$, we also get that if $\varepsilon \in A$ then $A^\infty = \Sigma^\infty$, because as already shown, if $\varepsilon \in A$ then $A^\omega = A^\infty$.

Note that the set of the infinite words of A^* may be different from the set of the infinite words of A. Indeed, take $A = \{a, b^\omega\}$. We have that:
(i) $A^* = \{\varepsilon\} \cup \{a, b^\omega\} \cup \{aa, b^\omega, ab^\omega\} \cup \{aaa, b^\omega, ab^\omega, aab^\omega\} \cup \ldots = a^* + a^*b^\omega$, and
(ii) $A \cap \Sigma^\omega = \{b^\omega\}$, while $A^* \cap \Sigma^\omega = a^*b^\omega$.

Obviously, the set of the infinite words of A is included in the set of the infinite words of A^*.

Exercise 3.2. Given a set $\mathcal{P}$ of CCS processes and a modal μ-calculus formula φ, show that the semantics of $\mu X.\varphi$ is equal to the semantics of $\neg\nu X.\neg\varphi[\neg X/X]$, which is $\neg\bigcup\{R \mid R \subseteq \neg\varphi[\neg R/X]\}$.

Solution. We have to show that $\bigcap\{S \subseteq \mathcal{P} \mid \varphi[S/X] \subseteq S\} = \neg\bigcup\{S \mid S \subseteq \neg\varphi[\neg S/X]\}$.
We have that: $\bigcap\{S \mid \varphi[S/X] \subseteq S\} =$ {by negating φ} $=$
$= \bigcap\{S \mid \neg\varphi[S/X] \supseteq \mathcal{P}-S\} =$ {by taking $S=\mathcal{P}-R$} $=$
$= \bigcap\{\mathcal{P}-R \mid \neg\varphi[\neg R/X] \supseteq R\} =$ {De Morgan} $=$
$= \neg\bigcup\neg\{\mathcal{P}-R \mid \neg\varphi[\neg R/X] \supseteq R\} =$ {by complementing $\mathcal{P}-R$} $=$
$= \neg\bigcup\{R \mid R \subseteq \neg\varphi[\neg R/X]\}$. □

Exercise 3.3. Show that the function $\lambda S \in \mathcal{P}.\,[a]S$, for some action $a \in Act$, is monotonic (with respect to set inclusion), but not continuous.

Solution. Monotonicity is obvious. For showing that continuity does *not* hold, let us consider the following ω-sequence of sets of processes:

$\{p_0\},\ \{p_0, p_1\},\ \ldots,\ \{p_0, p_1, \ldots, p_i\},\ \ldots$

where every process p_i is distinct from p_j, for $i \neq j$. (Recall that any finite set of processes can be viewed as an assertion.)

We have that:

(i) $[a]\{p_0\} = \{p \mid \forall q.\ p \xrightarrow{a} q \Rightarrow q=p_0\}$,

(ii) $[a]\{p_0, p_1\} = \{p \mid \forall q.\ p \xrightarrow{a} q \Rightarrow (q=p_0 \vee q=p_1)\}, \ldots,$

(iii) $[a]\{p_0, p_1, \ldots, p_i\} = \{p \mid \forall q.\ p \xrightarrow{a} q \Rightarrow q=p_j$ for some j with $0 \leq j \leq i\}$.

For instance, $\{a.p_0\} \subseteq [a]\{p_0\}$, $\{a.p_0, a.p_1, a.p_0 + a.p_1\} \subseteq [a]\{p_0, p_1\}$.

Now, the least upper bound of the given ω-sequence of set of processes is the set $\mathcal{A} =_{def} \{p_i \mid i \in \omega\}$. We have that:

$[a]\mathcal{A} = \big\{p \mid \forall q.\ \big(p \xrightarrow{a} q \Rightarrow (\exists i \geq 0.\, q=p_i)\big)\big\}$.

Thus, $[a]\mathcal{A}$ includes the process $\sum_{i\in\omega} a.p_i$, while this process does not belong to any set of the ω-chain:

$[a]\{p_0\},\ [a]\{p_0, p_1\},\ \ldots,\ [a]\{p_0, p_1, \ldots, p_i\},\ \ldots$

because every element in every set of this ω-chain is a CCS term which is a finite sum. The reader should recall here that an element is a member of the least upper bound of an ω-chain of sets (with respect to set inclusion) iff it is a member of one of the sets in the ω-chain. □

3.2. Some Useful Modal μ-Calculus Assertions.

Let us introduce the following assertions:

$$\begin{array}{lll}
(\alpha) \quad \mathit{possibly}(B) & =_{def}\ \mu X.(B \vee \langle \mathbf{.} \rangle X) & \equiv \langle \mathbf{.} \rangle^* B \\
(\beta) \quad \mathit{eventually}(B) & =_{def}\ \mu X.(B \vee (\langle \mathbf{.} \rangle \mathbf{true} \wedge [\mathbf{.}]X)) & \equiv (\langle \mathbf{.} \rangle \mathbf{true} \wedge [\mathbf{.}])^* B \\
(\varepsilon) \quad \mathit{inevitably}(A) & =_{def}\ \nu X.(A \wedge [\mathbf{.}]X) & \equiv (A \wedge [\mathbf{.}])^\omega \\
(\zeta) \quad A\ \mathit{until}\ B & =_{def}\ \nu X.(B \vee (A \wedge [\mathbf{.}]X)) & \equiv (A \wedge [\mathbf{.}])^* B \ \vee\ (A \wedge [\mathbf{.}])^\omega
\end{array}$$

The expressions in the rightmost column specify the values of the minimal fixpoints and maximal fixpoints in the middle column as indicated in Section 3.1 on page 277. For instance,

(i) $\langle \mathbf{.} \rangle^* B$ stands for $B \vee \langle \mathbf{.} \rangle B \vee \ldots \vee \langle \mathbf{.} \rangle^n B \vee \ldots$, for all $n \geq 0$ (thus, it may be the case that no infinite sequence of actions is possible), and

(ii) $(A \wedge [\mathbf{.}])^\omega$ stands for $A \wedge [\mathbf{.}]A \wedge \ldots \wedge [\mathbf{.}]^n A \wedge \ldots \wedge [\mathbf{.}]^\omega A$ (thus, an infinite sequence of actions is possible).

We can represent the meaning of those assertions by considering *trees of paths* (or *derivations trees* [**10**, page 49]), as indicated in Figure 2 on the facing page. In that figure we assume that after performing an action, a process 'moves to the right' and, since in general more than one action is possible, we get a tree rooted on the leftmost node. In particular (see also Figure 1),

(1) the term $a.t$ of CCS can be viewed as a tree such that: (1.1) the root has a single outgoing arc, (1.2) that arc has label a, and (1.3) that arc connects the root itself to the root of the subtree t, and

(2) the term $t_1 + t_2$ of CCS can be viewed as a tree whose root has the two subtrees t_1 and t_2.

By the Expansion Theorems (see Theorem 1.6 on page 267 and Theorem 1.11 on page 269), we have that also CCS terms which involve the parallel composition operator $|$, can be viewed as trees.

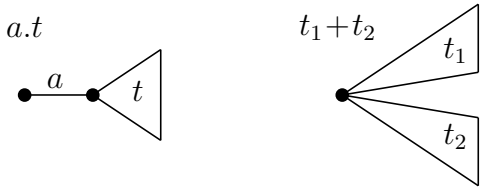


FIGURE 1. Trees of paths corresponding to the CCS terms $a.t$ and $t_1 + t_2$.

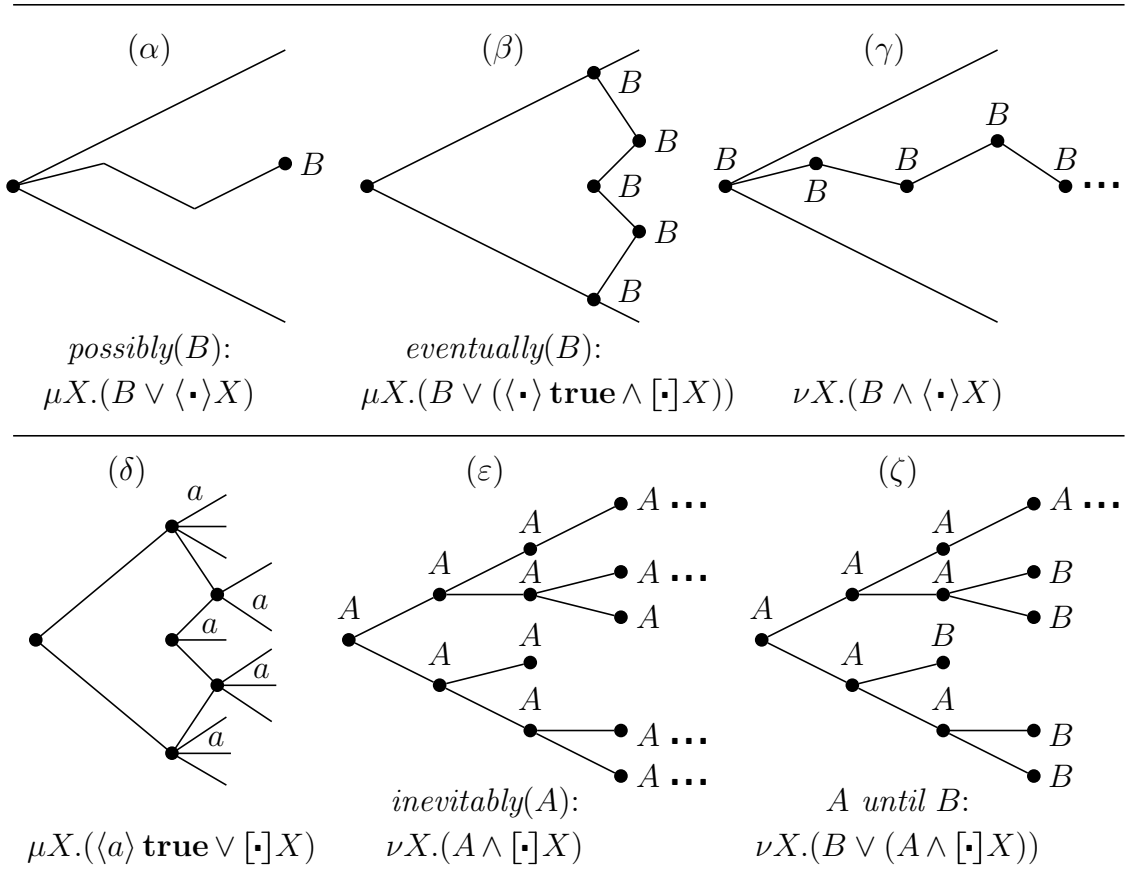


FIGURE 2. Trees which illustrate the meaning of some modal μ-calculus assertions. In (γ) the dots mean that there exists an *infinite* path on whose nodes B always holds. In (ε) the dots mean that: (i) A holds on all nodes of all (finite or infinite) paths. In (ζ) the dots mean that: (i) all paths starting from the process next to the dots, are infinite, and (ii) $A \wedge \neg B$ holds on all nodes of those infinite paths.

By recalling the meaning of the minimal and maximal fixpoints of equations as indicated in Section 3.1 on page 277, we have that:

(α) *possibly*(B) means that there is a process on a path starting from the root, on which B holds (see Figure 2(α) on the current page),

(β) *eventually*(B) means that on all paths starting from the root, there is a process on which B holds (see Figure 2(β)),

(γ) the assertion $\nu X.(B \wedge \langle \blacksquare \rangle X)$ means that there is an *infinite* path starting from the root on whose nodes B always holds (see Figure 2(γ)),

(δ) the assertion $\mu X.(\langle a \rangle \mathbf{true} \vee [\blacksquare] X)$ means that on all paths starting from the root there is a process which can do an a action (see Figure 2(δ)),

(ε) *inevitably*(A) means that A holds on every processes of all (finite or infinite) paths starting from the root (see Figure 2(ε)), and

(ζ) *A until B* means that for every path π starting from the root, *either* (ζ1) on all processes of the path π the assertion $A \wedge \neg B$ holds and π is infinite, *or* (ζ2) on the path π there is a finite prefix π_k such that:

(ζ2.1) π_k ends by a process on which B holds, and

(ζ2.2) in all other processes of π_k the assertion $A \wedge \neg B$ holds.

This notion of *until* is said to be the *weak until* because it may be the case that B never holds on an infinite path on whose nodes A always holds (see Figure 2(ζ)).

There is also a notion, called the *strong until*, where Case (ζ1) does not occur and, thus, on all paths π there is a process on which the assertion B holds. We have that the assertion *A strong until B* is $\mu X.(B \vee (A \wedge \langle \mathbf{.} \rangle \mathbf{true} \wedge [\mathbf{.}]X))$, that is, $(A \wedge \langle \mathbf{.} \rangle \mathbf{true} \wedge [\mathbf{.}])^* B$, if we were to use the notation of Section 3.1 (see page 277).

The following two assertions which are often of interest in practice.

(*Infinitely Often*)
The assertion $\nu Z.\mu Y.\langle a \rangle((\langle b \rangle \mathbf{true} \wedge Z) \vee Y)$ holds at a process p iff there exists an infinite path π beginning at p and whose sequence of labels is a^ω, on which *infinitely often* it is possible to perform a b action.

(*Almost Always*)
The assertion $\mu Y.\nu Z.\langle a \rangle((\langle b \rangle \mathbf{true} \vee Y) \wedge Z)$ holds at a process p iff there exists an infinite path π beginning at p and whose sequence of labels is a^ω, on which *almost always* it is possible to perform a b action, that is, only in a finite number of processes on π it is not possible to perform a b action.

Given a finite set $\mathcal{P}$ of processes and an action a, we have that:

(1.1) $\mu X.\langle a \rangle X$ is $\emptyset$ (that is, **false**),

(1.2) $\nu X.\langle a \rangle X$ is the set of processes in $\mathcal{P}$ which *can* do a^ω (that is, the set of processes in $\mathcal{P}$ which can do an infinite sequence of a actions),

(1.3) $\mu X.[a]X$ is the set of processes in $\mathcal{P}$ which *cannot* do a^ω (that is, the set of processes in $\mathcal{P}$ which cannot do an infinite sequence of a actions),

(1.4) $\nu X.[a]X$ is $\mathcal{P}$ (that is, **true**),

(2.1) $\mu X.\langle \mathbf{.} \rangle X$ is $\emptyset$ (that is, **false**),

(2.2) $\nu X.\langle \mathbf{.} \rangle X$ is the set of processes in $\mathcal{P}$ which *can* do an infinite sequence of actions,

(2.3) $\mu X.[\mathbf{.}]X$ is the set of processes in $\mathcal{P}$ which *cannot* do an infinite sequence of actions, and

(2.4) $\nu X.[\mathbf{.}]X$ is $\mathcal{P}$ (that is, **true**).

Note that for $k = 1, 2$, the expression $k.1$ is the complement (w.r.t. $\mathcal{P}$) of the expression $k.4$ and, likewise, the expression $k.2$ is the complement (w.r.t. $\mathcal{P}$) of the expression $k.3$. For instance, (1.2) is the complement of (1.3). Indeed,

$$\mu X.[a]X = \neg \nu X.\neg [a] \neg X = \neg \nu X.\langle a \rangle X.$$

4. The Local Model Checker for Finite Processes

In this section we present the *local model checker* which is an algorithm for testing whether or not, given a CCS process p and an assertion φ, we have that φ holds for p. In order for this algorithm to be a decision procedure, the given process p should be *a finite-state process*, that is, the set $\mathcal{P}_p$ of the *processes reachable from* p which we now define, should be finite. We define $\mathcal{P}_p$ as follows:

$\mathcal{P}_p =_{def} \{q \mid q \in \mathcal{P} \wedge p \longrightarrow^* q\}$

where: (i) for all processes $p, q \in \mathcal{P}$, $p \longrightarrow q$ holds iff $\exists\, a \in Act.\ p \xrightarrow{a} q$, and (ii) $\longrightarrow^*$ is the reflexive, transitive closure of $\longrightarrow$. When p is understood from the context, we will simply write $\mathcal{P}$, instead of $\mathcal{P}_p$.

Table 5 which resembles Table 4 on page 276, defines the satisfaction relation $\mathcal{P}, p \models \varphi$ which expresses the fact that the assertion φ holds for the finite-state process p whose set of reachable processes is $\mathcal{P}$.

With reference to Rules 10 and 11 of Table 5, note that in order to test whether or not $p \in \varphi$, the assertion φ is viewed as a set of processes, as indicated in Table 4, and $\vee$ denotes union of sets as well as disjunction of assertions. Dually, sets are viewed as assertions when we act on them using the operators $\neg$, $\vee$, and $\wedge$.

1.	$\mathcal{P}, p \models \mathbf{true}$		
2.	$\mathcal{P}, p \not\models \mathbf{false}$		
3.	$\mathcal{P}, p \models S$	if $p \in S$	for any subset $S \subseteq \mathcal{P}$
4.	$\mathcal{P}, p \not\models S$	if $p \notin S$	for any subset $S \subseteq \mathcal{P}$
5.	$\mathcal{P}, p \models \neg\varphi$	if $\mathcal{P}, p \not\models \varphi$	
6.	$\mathcal{P}, p \models \varphi_1 \wedge \varphi_2$	if $\mathcal{P}, p \models \varphi_1$ and $\mathcal{P}, p \models \varphi_2$	
7.	$\mathcal{P}, p \models \varphi_1 \vee \varphi_2$	if $\mathcal{P}, p \models \varphi_1$ or $\mathcal{P}, p \models \varphi_2$	
8.	$\mathcal{P}, p \models \langle a \rangle \varphi$	if there exists $q \in \{q \mid p \xrightarrow{a} q\}$ such that $\mathcal{P}, q \models \varphi$	
9.	$\mathcal{P}, p \models \langle \cdot \rangle \varphi$	if there exists $q \in \{q \mid \exists\, a \in Act.\ p \xrightarrow{a} q\}$ such that $\mathcal{P}, q \models \varphi$	
10.	$\mathcal{P}, p \models \nu X.\varphi$	if $p \in \varphi$	
11.	$\mathcal{P}, p \models \nu X.\varphi$	if $\mathcal{P}, p \models \varphi[\nu X.(\{p\} \vee \varphi) \,/\, X]$ and $p \notin \varphi$	

TABLE 5. The satisfaction relation $\mathcal{P}, p \models \varphi$ for a finite-state CCS process p and a modal μ-calculus assertion φ. The set $\mathcal{P}$ is the set of processes reachable from p. The action a is any action in Act.

The rewriting rules listed in Table 6 on the following page give us a sound and complete algorithm to decide whether or not $\mathcal{P}, p \models \varphi$ for a finite-state CCS process p and a modal μ-calculus assertion φ, and the set $\mathcal{P}$ of processes reachable from p, in the sense established by the following Theorem 4.1 on the next page (see [**19**, page 331]).

Note that according to the following Theorem 4.1 on the following page, when checking whether or not $\mathcal{P}, p \models \varphi$ holds, the initial expression to which the rewriting rules of Table 6 are applied, should be $\mathcal{P}, p \vdash \overline{\varphi}$, where $\overline{\varphi}$ is the assertion φ, where every subassertion $\nu X.\psi$ has been transformed into the equivalent assertion $\nu X.(\emptyset \vee \psi)$, where $\emptyset$ denotes the empty set of processes. In assertions such as $\nu X.(\emptyset \vee \psi)$,

1.	$\mathcal{P}, p \vdash \mathbf{true}$	$\mapsto$	$\mathbf{true}$	
2.	$\mathcal{P}, p \vdash \mathbf{false}$	$\mapsto$	$\mathbf{false}$	
3.	$\mathcal{P}, p \vdash S$	$\mapsto$	$\mathbf{true}$ if $p \in S$	for any subset $S \subseteq \mathcal{P}$
4.	$\mathcal{P}, p \vdash S$	$\mapsto$	$\mathbf{false}$ if $p \notin S$	for any subset $S \subseteq \mathcal{P}$
5.	$\mathcal{P}, p \vdash \neg\varphi$	$\mapsto$	$\neg(\mathcal{P}, p \vdash \varphi)$	
6.	$\mathcal{P}, p \vdash \varphi_1 \wedge \varphi_2$	$\mapsto$	$\mathcal{P}, p \vdash \varphi_1 \wedge \mathcal{P}, p \vdash \varphi_2$	
7.	$\mathcal{P}, p \vdash \varphi_1 \vee \varphi_2$	$\mapsto$	$\mathcal{P}, p \vdash \varphi_1 \vee \mathcal{P}, p \vdash \varphi_2$	
8.	$\mathcal{P}, p \vdash \langle a \rangle \varphi$	$\mapsto$	$\mathcal{P}, q_1 \vdash \varphi \vee \ldots \vee \mathcal{P}, q_n \vdash \varphi$ where $\{q_1, \ldots, q_n\} = \{q \mid p \xrightarrow{a} q\}$	
9.	$\mathcal{P}, p \vdash \langle \mathbf{.} \rangle \varphi$	$\mapsto$	$\mathcal{P}, q_1 \vdash \varphi \vee \ldots \vee \mathcal{P}, q_n \vdash \varphi$ where $\{q_1, \ldots, q_n\} = \{q \mid \exists\, a.\, p \xrightarrow{a} q\}$	
10.	$\mathcal{P}, p \vdash \nu X.(S \vee \varphi)$	$\mapsto$	$\mathbf{true}$ if $p \in S$	for any subset $S \subseteq \mathcal{P}$
11.	$\mathcal{P}, p \vdash \nu X.(S \vee \varphi)$	$\mapsto$	$\mathcal{P}, p \vdash \varphi[\nu X.(\{p\} \vee S \vee \varphi) \,/\, X]$ if $p \notin S$	for any subset $S \subseteq \mathcal{P}$

TABLE 6. Rewriting rules for establishing whether or not $\mathcal{P}, p \vdash \varphi$ holds for a finite-state CCS process p, a modal μ-calculus assertion φ, and the set $\mathcal{P}$ of processes reachable from p. Every empty disjunction of the form $b_1 \vee \ldots \vee b_n$, for $n = 0$, is rewritten to **false**. The action a is any action in Act.

the round parentheses will be omitted when understood from the context and, thus, for any set S of processes and any assertion ψ, we will feel free to write $\nu X.S \vee \psi$, instead of $\nu X.(S \vee \psi)$.

Note also that in Table 6 the operators $\neg$, $\vee$, and $\wedge$ are overloaded: sometimes they act on triples of the form $\mathcal{P}, p \vdash \varphi$ and some other times they act on assertions. The reader should distinguish between these two different uses.

THEOREM 4.1. [**Soundness and Completeness of the Local Model Checker**] Let p be a finite-state CCS process, φ be a modal μ-calculus assertion, and $\mathcal{P}$ be the finite set of processes reachable from p. We have that: $\mathcal{P}, p \models \varphi$ holds iff $(\mathcal{P}, p \vdash \overline{\varphi}) \mapsto^* \mathbf{true}$, where: (i) $\overline{\varphi}$ is the assertion φ where every subassertion $\nu X.\psi$ has been transformed into $\nu X.(\emptyset \vee \psi)$, and (ii) $\mapsto^*$ denotes the reflexive, transitive closure of the relation $\mapsto$ which is defined in Table 6.

The proof of this theorem is based on the following lemma [**19**, page 327].

LEMMA 4.2. [**Reduction Lemma**] Consider a set $\mathcal{P}$ and a monotonic function φ from $2^{\mathcal{P}}$ to $2^{\mathcal{P}}$, where $2^{\mathcal{P}}$ denotes the powerset of $\mathcal{P}$. For any $S \subseteq \mathcal{P}$ we have that:

$S \subseteq \nu X.\varphi(X) \;\Leftrightarrow\; S \subseteq \varphi(\nu X.(S \cup \varphi(X)))$.

PROOF. ($\Rightarrow$) First we show that $S \cup \varphi(\nu X.\varphi(X)) = \nu X.\varphi(X)$. Indeed,

$S \cup \varphi(\nu X.\varphi(X)) = \{\nu X.\varphi(X)$ is a fixpoint of $\varphi\} =$

$= S \cup \nu X.\varphi(X) = \{$by hypothesis, $S \subseteq \nu X.\varphi(X)\} =$

$= \nu X.\varphi(X)$.

Thus, $\nu X.\varphi(X)$ is a fixpoint of the function $\lambda X.S \cup \varphi(X)$. Since, by definition, $\nu X.(S \cup \varphi(X))$ is the maximal fixpoint of that function, we have that:

$$\nu X.\varphi(X) \subseteq \nu X.(S \cup \varphi(X)) \qquad (\dagger 1)$$

By monotonicity of φ, from (†1) we get:

$$\varphi(\nu X.\varphi(X)) \subseteq \varphi(\nu X.(S \cup \varphi(X))) \qquad (\dagger 2)$$

Since, by hypothesis, we have that $S \subseteq \nu X.\varphi(X)$, and by definition, $\nu X.\varphi(X) = \varphi(\nu X.\varphi(X))$, we get that $S \subseteq \varphi(\nu X.\varphi(X))$. Hence, by (†2) we get:

$$S \subseteq \varphi(\nu X.(S \cup \varphi(X))).$$

($\Leftarrow$) We assume that $S \subseteq \varphi(\nu X.(S \cup \varphi(X)))$.

$$\nu X.(S \cup \varphi(X)) = \{\text{by definition of } \nu\} = S \cup \varphi(\nu X.(S \cup \varphi(X))). \qquad (\dagger 3)$$

Then, by the assumption $S \subseteq \varphi(\nu X.(S \cup \varphi(X)))$, from (†3) we get:

$$\nu X.(S \cup \varphi(X)) = \varphi(\nu X.(S \cup \varphi(X))). \qquad (\dagger 4)$$

Thus, $\nu X.(S \cup \varphi(X))$ is a fixpoint of φ. Since $\nu X.\varphi(X)$ is the maximal fixpoint of φ, we get:

$$\nu X.(S \cup \varphi(X)) \subseteq \nu X.\varphi(X). \qquad (\dagger 5)$$

By the assumption $S \subseteq \varphi(\nu X.(S \cup \varphi(X)))$, from (†4) we get: $S \subseteq \nu X.(S \cup \varphi(X))$. By (†5) we get:

$$S \subseteq \nu X.\varphi(X). \qquad \square$$

Note that, given a set $\mathcal{P}$ and a monotonic function φ from $2^{\mathcal{P}}$ to $2^{\mathcal{P}}$, for any $S \subseteq \mathcal{P}$, we have that:

(1) $S \subseteq \nu X.\varphi(X) \;\Rightarrow\; S \subseteq \nu X.(S \cup \varphi(X))$

(2) $S \subseteq \nu X.\varphi(X) \;\not\Leftarrow\; S \subseteq \nu X.(S \cup \varphi(X))$

Indeed, for (1) we have that: (1.i) $\nu X.\varphi(X) \subseteq \nu X.(S \cup \varphi(X))$ (as shown in (†1) of Lemma 4.2), and (1.ii) by hypothesis, $S \subseteq \nu X.\varphi(X)$.
For (2) we consider the function $\nu X.B \cup AX$ for some sets of words A and B. We have that: (2.i) $\nu X.B \cup AX$ is $A^\omega \cup A^*B$, (2.ii) $\nu X.AX$ is A^ω, and (2.iii) in general, $A^*B \not\subseteq A^\omega$.

In the following examples we show the evaluation of the satisfaction relation $\mathcal{P}, p \models \varphi$ by using the rewriting rules of Table 6 on the facing page.

Example 4.3. Let us consider the following four CCS finite-state processes:

$$P_1 =_{def} a.P_2 + b.P_4 \qquad P_2 =_{def} a.P_3 + a.P_4 \qquad P_3 =_{def} a.P_4 \qquad P_4 =_{def} 0$$

They can be depicted as in Figure 3. Let $\mathcal{P}$ be the set $\{P_1, P_2, P_3, P_4\}$.
We want to check whether or not $\mathcal{P}, P_1 \models \nu X.\langle a \rangle X$ holds, that is, whether or not from P_1 one can do an infinite sequence of a actions (recall that $\nu X.\langle a \rangle X$ is $\langle a \rangle^\omega$).

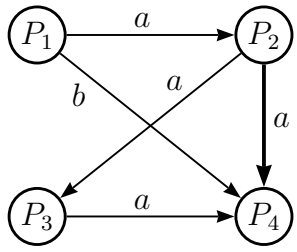


FIGURE 3. Some CCS processes. From P_1 it is *not* possible to do an infinite sequence of a actions.

We have that (for reasons of brevity, we have not indicated all the rewriting steps and thus, sometimes $\mapsto$ actually stands for $\mapsto^*$):

$\mathcal{P}, P_1 \vdash \nu X.(\emptyset \vee \langle a\rangle X)$

$\mapsto \mathcal{P}, P_1 \vdash \langle a\rangle X\,[\nu X.(\{P_1\} \vee \langle a\rangle X)\,/\,X]$, that is, $\mathcal{P}, P_1 \vdash \langle a\rangle \nu X.(\{P_1\} \vee \langle a\rangle X)$

$\mapsto \mathcal{P}, P_2 \vdash \nu X.(\{P_1\} \vee \langle a\rangle X)$

$\mapsto \mathcal{P}, P_2 \vdash (\{P_1\} \vee \langle a\rangle X)\,[\nu X.(\{P_1, P_2\} \vee \langle a\rangle X)/X]$, that is,
$\mathcal{P}, P_2 \vdash \{P_1\} \vee \langle a\rangle \nu X.(\{P_1, P_2\} \vee \langle a\rangle X)$

$\mapsto \mathcal{P}, P_2 \vdash \langle a\rangle \nu X.(\{P_1, P_2\} \vee \langle a\rangle X)$

$\mapsto \mathcal{P}, P_3 \vdash \nu X.(\{P_1, P_2\} \vee \langle a\rangle X) \;\vee\; \mathcal{P}, p_4 \vdash \nu X.(\{P_1, P_2\} \vee \langle a\rangle X)$

$\mapsto \mathcal{P}, P_3 \vdash \{P_1, P_2\} \vee \langle a\rangle \nu X.(\{P_1, P_2, P_3\} \vee \langle a\rangle X) \;\vee$
$\mathcal{P}, P_4 \vdash \{P_1, P_2\} \vee \langle a\rangle \nu X.(\{P_1, P_2, P_4\} \vee \langle a\rangle X)$

$\mapsto \mathcal{P}, P_3 \vdash \langle a\rangle \nu X.(\{P_1, P_2, P_3\} \vee \langle a\rangle X) \;\vee\; \mathcal{P}, P_4 \vdash \langle a\rangle \nu X.(\{P_1, P_2, P_4\} \vee \langle a\rangle X)$

$\mapsto \mathcal{P}, P_4 \vdash \nu X.(\{P_1, P_2, P_3\} \vee \langle a\rangle X) \;\vee\; \mathbf{false}$

$\mapsto \mathcal{P}, P_4 \vdash \{P_1, P_2, P_3\} \vee \langle a\rangle \nu X.(\{P_1, P_2, P_3, P_4\} \vee \langle a\rangle X)$

$\mapsto \mathcal{P}, P_4 \vdash \langle a\rangle \nu X.(\{P_1, P_2, P_3, P_4\} \vee \langle a\rangle X)$

$\mapsto \mathbf{false}$

We may also establish that $\mathcal{P}, P_1 \vdash \nu X.\langle a\rangle X$ does not hold by computing the maximal fixpoint of the function $\lambda X.\langle a\rangle X$ starting from $\mathcal{P}$ and iterating the application of $\lambda X.\langle a\rangle X$ until a fixpoint is reached. Recall that, given a set $X \subseteq \mathcal{P}$, $\langle a\rangle X$ is the subset of $\mathcal{P}$ such that every process in $\langle a\rangle X$ can do an a action and become a process in X. We have the following sequence of sets of processes, where in line 1 we have indicated the set $\mathcal{P}$ of all processes, and for $i = 2, 3, 4, 5$, in line i we have indicated the set of processes which can do an a action and become a process in the set indicated in line $i-1$:

1. $\mathcal{P} = \{P_1, P_2, P_3, P_4\}$
2. $\langle a\rangle \mathcal{P} = \{P_1, P_2, P_3\}$
3. $\langle a\rangle(\langle a\rangle \mathcal{P}) = \{P_1, P_2\}$
4. $\langle a\rangle(\langle a\rangle(\langle a\rangle \mathcal{P})) = \{P_1\}$
5. $\langle a\rangle(\langle a\rangle(\langle a\rangle(\langle a\rangle \mathcal{P}))) = \emptyset$

At line 5 we have reached a fixpoint, because $\langle a \rangle \emptyset = \emptyset$. Thus, since the maximal fixpoint is $\emptyset$, we have that for no process in $\mathcal{P}$ we can do an infinite sequence of a actions.

EXAMPLE 4.4. If we consider the following four processes:

$Q_1 =_{def} a.Q_2 + b.Q_4 \qquad Q_2 =_{def} a.Q_3 \qquad Q_3 =_{def} a.Q_4 \qquad Q_4 =_{def} a.Q_2$

which can be depicted as in Figure 4. Let $\mathcal{Q}$ be the set $\{Q_1, Q_2, Q_3, Q_4\}$.

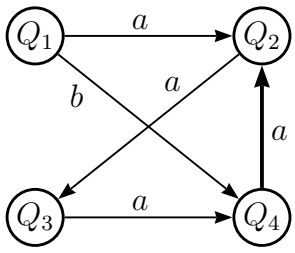


FIGURE 4. Some CCS processes. From Q_1 it is possible to do an infinite sequence of a actions. Note the edge labeled by a from Q_4 to Q_2, instead the edge labeled by a from P_2 to P_4 of Figure 3 on the preceding page.

We have that $\mathcal{Q}, Q_1 \models \nu X.\langle a \rangle X$ holds. Indeed:

$\mathcal{Q}, Q_1 \vdash \nu X.(\emptyset \vee \langle a \rangle X)$
$\quad \mapsto \mathcal{Q}, Q_1 \vdash \langle a \rangle \nu X.(\{Q_1\} \vee \langle a \rangle X)$
$\quad \mapsto \mathcal{Q}, Q_2 \vdash \nu X.(\{Q_1\} \vee \langle a \rangle X)$
$\quad \mapsto \mathcal{Q}, Q_2 \vdash \{Q_1\} \vee \langle a \rangle (\nu X.(\{Q_1, Q_2\} \vee \langle a \rangle X))$
$\quad \mapsto \mathcal{Q}, Q_2 \vdash \langle a \rangle (\nu X.(\{Q_1, Q_2\} \vee \langle a \rangle X))$
$\quad \mapsto \mathcal{Q}, Q_3 \vdash \nu X.(\{Q_1, Q_2\} \vee \langle a \rangle X)$
$\quad \mapsto \mathcal{Q}, Q_3 \vdash \{Q_1, Q_2\} \vee \langle a \rangle \nu X.(\{Q_1, Q_2, Q_3\} \vee \langle a \rangle X)$
$\quad \mapsto \mathcal{Q}, Q_3 \vdash \langle a \rangle \nu X.(\{q_1, Q_2, Q_3\} \vee \langle a \rangle X)$
$\quad \mapsto \mathcal{Q}, Q_4 \vdash \nu X.(\{Q_1, Q_2, Q_3\} \vee \langle a \rangle X)$
$\quad \mapsto \mathcal{Q}, Q_4 \vdash \{Q_1, Q_2, Q_3\} \vee \langle a \rangle \nu X.(\{Q_1, Q_2, Q_3, Q_4\} \vee \langle a \rangle X)$
$\quad \mapsto \mathcal{Q}, Q_4 \vdash \langle a \rangle \nu X.(\{Q_1, Q_2, Q_3, Q_4\} \vee \langle a \rangle X)$
$\quad \mapsto \mathcal{Q}, Q_2 \vdash \nu X.(\{Q_1, Q_2, Q_3, Q_4\} \vee \langle a \rangle X)$
$\quad \mapsto \mathcal{Q}, Q_2 \vdash \{Q_1, Q_2, Q_3, Q_4\} \vee \langle a \rangle \nu X.(\{Q_1, Q_2, Q_3, Q_4\} \vee \langle a \rangle X)$
$\quad \mapsto \mathbf{true}$

As indicated at the end of Example 4.3 on page 285, we can establish that $\mathcal{Q}, Q_1 \vdash \nu X.\langle a \rangle X$ holds by computing the maximal fixpoint of the function $\lambda X.\langle a \rangle X$ starting from $\mathcal{Q}$. By applying the function $\lambda X.\langle a \rangle X$ to the set of processes $\mathcal{Q}$ we get the set $\langle a \rangle \mathcal{Q} = \{Q_1, Q_2, Q_3, Q_4\} = \mathcal{Q}$ (recall that $\langle a \rangle \mathcal{Q}$ by definition is the set of processes which can do an a action and become a process in $\mathcal{Q}$). Thus, a fixpoint is reached and we have that from any process in $\mathcal{Q}$ we can do an infinite sequence of a actions.

EXAMPLE 4.5. Let us consider the set $\mathcal{Q}$ of processes of Example 4.4 on the previous page. We have that $\mathcal{Q}, Q_1 \models \mu X.\langle a\rangle X$ does *not* hold, that is, $\mathcal{Q}, Q_1 \vdash \mu X.\langle a\rangle X \ \mapsto^* \ \mathbf{false}$. Indeed, we have that: for all $\mathbf{b} \in \{\mathbf{true}, \mathbf{false}\}$,

$\mathcal{Q}, Q_1 \vdash \mu X.\langle a\rangle X \ \mapsto^* \ \mathbf{b}$ {by the definition of $\mu X.\varphi$}
iff $\mathcal{Q}, Q_1 \vdash \neg\nu X.\neg\langle a\rangle\neg X \ \mapsto^* \ \mathbf{b}$
{by the initialization indicated in Theorem 4.1 on page 284}
iff $\mathcal{Q}, Q_1 \vdash \nu X.(\emptyset \vee \neg\langle a\rangle\neg X) \ \mapsto^* \ \neg\mathbf{b}$
iff $\mathcal{Q}, Q_1 \vdash \neg\langle a\rangle\neg\nu X.(\{Q_1\} \vee \neg\langle a\rangle\neg X) \ \mapsto^* \ \neg\mathbf{b}$
iff $\mathcal{Q}, Q_1 \vdash \langle a\rangle\neg\nu X.(\{Q_1\} \vee \neg\langle a\rangle\neg X) \ \mapsto^* \ \mathbf{b}$
iff $\mathcal{Q}, Q_2 \vdash \neg\nu X.(\{Q_1\} \vee \neg\langle a\rangle\neg X) \ \mapsto^* \ \mathbf{b}$
iff $\mathcal{Q}, Q_2 \vdash \nu X.(\{Q_1\} \vee \neg\langle a\rangle\neg X) \ \mapsto^* \ \neg\mathbf{b}$
iff $\mathcal{Q}, Q_2 \vdash \{Q_1\} \vee \neg\langle a\rangle\neg\nu X.(\{Q_1, Q_2\} \vee \neg\langle a\rangle\neg X) \ \mapsto^* \ \neg\mathbf{b}$
iff $\mathcal{Q}, Q_2 \vdash \langle a\rangle\neg\nu X.(\{Q_1, Q_2\} \vee \neg\langle a\rangle\neg X) \ \mapsto^* \ \mathbf{b}$
iff $\mathcal{Q}, Q_3 \vdash \nu X.(\{Q_1, Q_2\} \vee \neg\langle a\rangle\neg X) \ \mapsto^* \ \neg\mathbf{b}$
iff $\mathcal{Q}, Q_3 \vdash \{Q_1, Q_2\} \vee \neg\langle a\rangle\neg\nu X.(\{Q_1, Q_2, Q_3\} \vee \neg\langle a\rangle\neg X) \ \mapsto^* \ \neg\mathbf{b}$
iff $\mathcal{Q}, Q_3 \vdash \langle a\rangle\neg\nu X.(\{Q_1, Q_2, Q_3\} \vee \neg\langle a\rangle\neg X) \ \mapsto^* \ \mathbf{b}$
iff $\mathcal{Q}, Q_4 \vdash \neg\nu X.(\{Q_1, Q_2, Q_3\} \vee \neg\langle a\rangle\neg X) \ \mapsto^* \ \mathbf{b}$
iff $\mathcal{Q}, Q_4 \vdash \nu X.(\{Q_1, Q_2, Q_3\} \vee \neg\langle a\rangle\neg X) \ \mapsto^* \ \neg\mathbf{b}$
iff $\mathcal{Q}, Q_4 \vdash \{Q_1, Q_2, Q_3\} \vee \neg\langle a\rangle\neg\nu X.(\{Q_1, Q_2, Q_3, Q_4\} \vee \neg\langle a\rangle\neg X) \ \mapsto^* \ \neg\mathbf{b}$
iff $\mathcal{Q}, Q_4 \vdash \neg\langle a\rangle\neg\nu X.(\{Q_1, Q_2, Q_3, Q_4\} \vee \neg\langle a\rangle\neg X) \ \mapsto^* \ \neg\mathbf{b}$
iff $\mathcal{Q}, Q_4 \vdash \langle a\rangle\neg\nu X.(\{Q_1, Q_2, Q_3, Q_4\} \vee \neg\langle a\rangle\neg X) \ \mapsto^* \ \mathbf{b}$
iff $\mathcal{Q}, Q_2 \vdash \neg\nu X.(\{Q_1, Q_2, Q_3, Q_4\} \vee \neg\langle a\rangle\neg X) \ \mapsto^* \ \mathbf{b}$
iff $\mathcal{Q}, Q_2 \vdash \nu X.(\{Q_1, Q_2, Q_3, Q_4\} \vee \neg\langle a\rangle\neg X) \ \mapsto^* \ \neg\mathbf{b}$
iff $\mathcal{Q}, Q_2 \vdash \{Q_1, Q_2, Q_3, Q_4\} \vee \neg\langle a\rangle\neg\nu X.(\{Q_1, Q_2, Q_3, Q_4\} \vee \neg\langle a\rangle\neg X) \ \mapsto^* \ \neg\mathbf{b}$

Now, in this last expression we have that $\neg\mathbf{b}$ is **true** because $Q_2 \in \{Q_1, Q_2, Q_3, Q_4\}$. Thus, we get that $\mathcal{Q}, Q_1 \vdash \mu X.\langle a\rangle X \ \mapsto^* \ \mathbf{false}$.

As we have indicated at the end of Example 4.3 on page 285, we can check that $\mathcal{Q}, Q_1 \vdash \mu X.\langle a\rangle X$ does not hold also by computing the minimal fixpoint of the function $\lambda X.\langle a\rangle X$, starting from the empty set and iteratively applying the function $\lambda X.\langle a\rangle X$ until we get a set $\mathcal{M} \subseteq \mathcal{Q}$ such that $\langle a\rangle\,\mathcal{M} = \mathcal{M}$. Then we have that $\mathcal{Q}, Q_1 \vdash \mu X.\langle a\rangle X$ iff $Q_1 \in \mathcal{M}$. Since $\langle a\rangle\,\emptyset = \emptyset$, we have that $\langle a\rangle\,\mathcal{M} = \mathcal{M}$, for $\mathcal{M} = \emptyset$. Thus, since $Q_1 \notin \emptyset$, we have that $\mathcal{Q}, Q_1 \vdash \mu X.\langle a\rangle X$ does not hold, as expected.

Here is an alternative way of deriving the same result. The value of $\mu X.\langle a\rangle X$ which is equivalent to $\mu X.(\langle a\rangle X \vee \mathbf{false})$, is $\langle a\rangle^* \mathbf{false}$ (see Equation $E1$ on page 278). Thus, $\mu X.\langle a\rangle X$ holds for a process P such that $\mathbf{false} \vee \langle a\rangle\,\mathbf{false} \vee \langle a\rangle\langle a\rangle\,\mathbf{false} \vee \ldots$ holds, that is, $\mu X.\langle a\rangle X$ holds for a process P if there exists $n \geq 0$ such that P can do an a action n times, thereby reaching a state where **false** holds. Since **false** holds in no state, we have that for all processes in $\mathcal{Q}$, $\mu X.\langle a\rangle X$ does not hold.

In order to test whether or not a given CCS term satisfies a given modal μ-calculus formula one can use an automatic verification system, the Edinburgh Concurrency Workbench [**13**].

On page 290 we will present a program, called `peterson-mucalculus.cwb`, which by using the Edinburgh Concurrency Workbench, demonstrates that: (i) some CCS agents satisfy some modal μ-calculus formulas, and (ii) Peterson algorithm guarantees mutual exclusion, absence of deadlock, absence of starvation, and bounded overtaking.

In that program we use the following notations. The operator `<->` stands for $\langle \texttt{.} \rangle$ and `[-]` stands for $[\texttt{.}]$. The operator `~` (tilde) stands for $\neg$ (not), `|` stands for $\vee$ (or), and `&` stands for $\wedge$ (and). The operator `max`$(X.\varphi)$ stands for $\nu X.\varphi$ and, similarly, `min`$(X.\varphi)$ stands for $\mu X.\varphi$.

For any action `a`$\in Act$, for any μ-calculus formula φ, an agent P satisfies `<<a>>` φ iff there exists an agent P' such that $P \stackrel{\texttt{a}}{\Longrightarrow} P'$ (see the definition of $\stackrel{\texttt{a}}{\Longrightarrow}$ on page 265) and P' satisfies φ. Similarly, an agent P satisfies `[[a]]` φ iff for all agents P', if $P \stackrel{\texttt{a}}{\Longrightarrow} P'$ then P' satisfies φ.

For any μ-calculus formula φ, an agent P satisfies `<<->>` φ iff there exist a visible action `a`, (that is, an action `a` $\in Act - \{\tau\}$) and an agent P' such that $P \stackrel{\texttt{a}}{\Longrightarrow} P'$ and P' satisfies φ. Similarly, an agent P satisfies `[[-]]` φ iff for all visible actions `a` and agents P', if $P \stackrel{\texttt{a}}{\Longrightarrow} P'$ then P' satisfies φ. In particular, `[[-]]` φ holds if P cannot perform any `a` action.

In any line whatever follows a semicolon '`;`' is a comment, and when we write a line of the form:

```
prop P = ...;            A: true        B: false
```

we mean that the formula `P` holds for the process `A` and it does not hold for the process `B`. The following pair of commands:

```
echo "for P1: F1 true =";
checkprop(P1,F1);
```

produces as output the line:

```
for P1: F1 true =
```

followed by the line:

```
true
```

(or `false`), if the agent `P1` satisfies (or does not satisfy) property `F1`. The `echo` command has been inserted for anticipating the expected answer so that, at run time, we may easily check whether or not the computed answer is the correct one.

The agent `Peterson` and all other agents which are required in the definition of the agent `Peterson`, are the same as in the Peterson algorithm listed on page 272, except that: (i) now we have different actions `in` and `out` for the different agents (we have the actions `in1` and `out1` for agent `P12` and the actions `in2` and `out2` for agent `P22`), and (ii) we have added to the agent `P1` a visible action `b1` and to the agent `P2` a visible action `b2`, for easily identifying the point in time when an agent has completed a request to enter its critical section.

The fact that Peterson algorithm guarantees mutual exclusion has been expressed by the validity of the following modal μ-calculus formula:

$$\psi =_{def} \nu X.\big([in_1]\,([in_2]\,\mathbf{false}) \wedge [in_2]\,([in_1]\,\mathbf{false}) \wedge [\,\mathbf{.}\,]\,X\big)$$

which is equivalent to: $\neg\mu X.\big(\langle in_1\rangle\,(\langle in_2\rangle\,\mathbf{true}) \vee \langle in_2\rangle\,(\langle in_1\rangle\,\mathbf{true}) \vee \langle\,\mathbf{.}\,\rangle\,X\big)$.

The formula ψ expresses the fact that for an agent: (i) it is impossible to perform an in_1 action immediately before an in_2 action, and (ii) it is impossible to perform an in_2 action immediately before an in_1 action, and (iii) whatever action the agent performs, it becomes an agent that satisfies the same formula ψ.

In order to express the mutual exclusion property, instead of the formula ψ, we can also use the following formula:

$$\psi' =_{def} \nu X.\big([out_1]\,([out_2]\,\mathbf{false}) \wedge [out_2]\,([out_1]\,\mathbf{false}) \wedge [\,\mathbf{.}\,]\,X\big)$$

because it is possible to perform the sequence of actions $out_1\,out_2$ or the sequence $out_2\,out_1$, iff two distinct agents have been in their critical section at the same time.

```
* ========================================================================
*               PETERSON ALGORITHM AND MU-CALCULUS ASSERTIONS
*
* filename: peterson-mucalculus.cwb
* Use of the Edinburgh Concurrency Workbench.
* ------------------------------------------------------------------------

agent A = a.0 + b.a.0;
prop  P = <a>T & <b><a>T;                  A: true
* ------------------------------------------------------------------------
*                    Minimal and maximal fixpoints
* ------------------------------------------------------------------------
agent Z = 0;
agent T0 = a.T0;

prop  Ma  = min(X.<a>T | [-]X);            Z: true     T0: true
prop  Na  = max(X.<a>T | [-]X);            Z: true     T0: true
prop  Ma1 = min(X.<a>T | (<->T & [-]X));   Z: false    T0: true
prop  Na1 = max(X.<a>T | (<->T & [-]X));   Z: false    T0: true

* ========================================================================
*** variable q1
agent Q1t  = 'q1rt.Q1t + q1wt.Q1t + q1wf.Q1f;  * q1 true
agent Q1f  = 'q1rf.Q1f + q1wt.Q1t + q1wf.Q1f;  * q1 false
*** variable q2
agent Q2t  = 'q2rt.Q2t + q2wt.Q2t + q2wf.Q2f;  * q2 true
agent Q2f  = 'q2rf.Q2f + q2wt.Q2t + q2wf.Q2f;  * q2 false
*** variable S
agent S1  = 'r1.S1 + w1.S1 + w2.S2;            * s = 1
agent S2  = 'r2.S2 + w1.S1 + w2.S2;            * s = 2

* ------------------------------------------------------------------------
* Note the visible actions b1 and b2. Performing the action b1 (or b2)
* means that the request to enter the critical section is completed.
* ------------------------------------------------------------------------
*** process Peterson
agent P1  = 'q1wt.'w1.b1.P11;                * <-----------------------+
agent P11 = q2rt.P11 + r1.P11 + q2rf.P12 + r2.P12;                      |
agent P12 = in1.out1.'q1wf.P1;               * =-----------------------+
```

```
agent P2  = 'q2wt.'w2.b2.P21;                * <------------------------+
agent P21 = q1rt.P21 + r2.P21 + q1rf.P22 + r1.P22;                       |
agent P22 = in2.out2.'q2wf.P2;               * =------------------------+

agent Peterson = (P1|P2|Q1f|Q2f|S1)\L;
set   L = {q1rt,q1rf,q1wt,q1wf,q2rt,q2rf,q2wt,q2wf,r1,r2,w1,w2};

* ------------------------------------------------------------------------
* process Reduced_Peterson avoids infinitely many reading operations at
* P11 or P21, i.e., staying in P11 or P21 forever.
* ------------------------------------------------------------------------
*** process Reduced_Peterson
agent R1  = 'q1wt.'w1.b1.R11;                * <------------------------+
agent R11 = q2rf.R12 + r2.R12;          (only two summands)              |
agent R12 = in1.out1.'q1wf.R1;               * =------------------------+

agent R2  = 'q2wt.'w2.b2.R21;                * <------------------------+
agent R21 = q1rf.R22 + r1.R22;          (only two summands)              |
agent R22 = in2.out2.'q2wf.R2;               * =------------------------+

agent Reduced_Peterson = (R1|R2|Q1f|Q2f|S1)\L;
* ------------------------------------------------------------------------
prop Inv(Phi) = max(Y.(Phi & [-]Y));
prop Ev(Phi)  = min(X.(Phi | (<->T & [-]X)));
*
* ------------------------------------------------------------------------
*                       Mutual Exclusion Property
* ------------------------------------------------------------------------
prop Mutex = max(X.[in1] [in2] F & [in2] [in1] F & [-] X);

* The following two sequences: (i) ... in2 in1 ...,  (ii) ... in1 in2 ...
* are impossible.
* Peterson: true; Reduced_Peterson: true;
* The following stronger version of the Mutual Exclusion property, where
* tau actions are not considered, holds for Peterson and also for
* Reduced_Peterson:
* prop Mutex1 = max(X.[[in1]] [[in2]] F & [[in2]] [[in1]] F & [[-]] X);
*
*** In what follows, after: echo "for Peterson: Mutex true =";        ***
*** and checkprop(Peterson,Mutex), we should get: true.               ***
*** Similarly, for the other echo-checkprop pairs, both in the case   ***
*** of "true" and "false".                                            ***

echo "====== Mutual exclusion";
echo "for Peterson: Mutex true =";
checkprop(Peterson,Mutex);
echo "for Reduced_Peterson: Mutex true =";
checkprop(Reduced_Peterson,Mutex);
*
* ------------------------------------------------------------------------
*                           Absence of Deadlock
* ------------------------------------------------------------------------
prop AD = (Inv([[out1]] Ev(((<<in1>>T)|(<<in2>>T))))) &
          (Inv([[out2]] Ev(((<<in1>>T)|(<<in2>>T)))));
* Peterson: false (because infinitely many read operations at P11 or P21)
* Reduced_Peterson: true
```

```
echo "====== Absence of deadlock";
echo "for Peterson: AD false = ";
checkprop(Peterson,AD);
echo "for Reduced_Peterson: AD true = ";
checkprop(Reduced_Peterson,AD);
*
* ------------------------------------------------------------------
*                          Absence of Starvation
* ------------------------------------------------------------------
* After the actions in1, out1, and 'q1wf by process P22 (or R22), all
* actions of the system Peterson (or Reduced_Peterson) may be performed
* by the other process. This case occurs if P1 (or R1) after exiting its
* critical section, does not make any request to re-enter in it.

prop AS1 = Inv([[b1]] Ev(<<in1>> T));
* Peterson: false (because infinitely many read operations at P11 or P21)
* Reduced_Peterson: true  (because only one read operation at R11 or R21)

prop AS2 = Inv([[in1]] Ev(<<in1>> T));
* Peterson: false; Reduced_Peterson: false;
* false because no request is made to re-enter the critical section.

echo "====== Absence of starvation";
echo "for Peterson: AS1 false = ";
checkprop(Peterson,AS1);
echo "for Reduced_Peterson: AS1 true = ";
checkprop(Reduced_Peterson,AS1);
* ---------
echo "for Peterson: AS2 false = ";
checkprop(Peterson,AS2);
echo "for Reduced_Peterson: AS2 false = ";
checkprop(Reduced_Peterson,AS2);
*
* ------------------------------------------------------------------
*                           Bounded Overtaking
* ------------------------------------------------------------------
prop B1 =  max(X.([in2] [b1] [out2] [b2] [in2] F) & [-]X);
* The following sequence (forgetting tau actions):
* in2 b1 out2 b2 in2 ... is impossible,
* because after b1, P2 may exit twice from the critical section.
* Peterson: true; Reduced_Peterson: true;
prop B2 = ~max(X.([[b2]][[in2]][[out2]][[b1]][[b2]][[in2]] F) & [[-]]X);
* The following sequence (forgetting tau actions):
* b2 in2 out2 b1 b2 in2 ... is possible,
* because after b1, P2 may exit once from the critical section.
* Peterson: true; Reduced_Peterson: true;

echo "====== Bounded overtaking";
echo "for Peterson: B1 true =";
checkprop(Peterson, B1);
echo "for Peterson: B2 true =";
checkprop(Peterson, B2);
* ---------
echo "for Reduced_Peterson: B1 true =";
checkprop(Reduced_Peterson,B1);
echo "for Reduced_Peterson: B2 true =";
checkprop(Reduced_Peterson,B2);
* ==================================================================
```

In the following program, called `examples.cwb`, we show some more examples of use the Edinburgh Concurrency Workbench [**13**] and we show, in particular, that some CCS agents satisfy some modal μ-calculus formulas which use minimal and maximal fixpoints operators.

In that program we have defined the sets $\{\texttt{P1}, \texttt{P2}, \texttt{P3}, \texttt{P4}\}$ and $\{\texttt{Q1}, \texttt{Q2}, \texttt{Q3}, \texttt{Q4}\}$ of agents (see the lines marked with `(*)`). For those sets we have that: (i) agent `P1` cannot do an infinite sequence of a actions because $\nu X.\langle a\rangle X$ does not hold for `P1` (see `P1 : false` in line `(!)` of program `examples.cwb` and also Example 4.3 on page 285), and (ii) agent `Q1` can do an infinite sequence of a actions because $\nu X.\langle a\rangle X$ holds for `Q1` (see `Q1 : true` in line `(!)` of program `examples.cwb` and also Example 4.4 on page 287).

Note that, given a finite set $\mathcal{P}$ of agents, we have that: (i) $\mu X.\langle a\rangle X$ holds for no agent in $\mathcal{P}$, (ii) $\nu X.\langle a\rangle X$ holds for the agents in $\mathcal{P}$ which can do an infinite sequence of a actions, also denoted a^ω, (iii) $\mu X.[a]X$ holds for all agents in $\mathcal{P}$, except those which can do a^ω, and (iv) $\nu X.[a]X$ holds for all agents in $\mathcal{P}$.

Exercise 4.6. Show that, given a finite set $\mathcal{P}$ of agents, we have that:
(i) $\mu X.\langle \texttt{.}\rangle X$ holds for no agent, (ii) $\nu X.\langle \texttt{.}\rangle X$ holds for the agents which can do an infinite sequence of actions, (iii) $\mu X.[\texttt{.}]X$ holds for all agents in $\mathcal{P}$ which cannot do an infinite sequence of actions, and (iv) $\nu X.[\texttt{.}]X$ holds for all agents in $\mathcal{P}$.

```
* ========================================================================
*     PROVING PROPERTIES OF CCS PROCESSES USING THE MODAL MU-CALCULUS
*
* filename: examples.cwb
* Use of the Edinburgh Concurrency Workbench.
* ------------------------------------------------------------------------
*                     More on minimal and maximal fixpoints
* ------------------------------------------------------------------------
agent B = a.C;                <---a---
agent C = a.B + b.0;        B ---a--> C ---b--> 0
agent H = b.0;
prop  Maa    = min(X.<a><a>X);              B: false    C: false
prop  MabT   = min(X.<a><b>T);              B: true     C: false
prop  Mabor  = min(X.<a><a>X | <a><b>T);    B: true     C: false
prop  Maband = min(X.<a><a>X & <a><b>T);    B: false    C: false
prop  Naband = max(X.<a><a>X & <a><b>T);    B: true     C: false
prop  Delta  = max(X.(<a>T | [-]X));        H: true

* ------------------------------------------------------------------------
agent P1 = a.P2 + b.P4;   In P1, P2, P3, and P4 there are no a-cycles.(*)
agent P2 = a.P3 + a.P4;   (see also line (!) below)                    (*)
agent P3 = a.P4;                                                       (*)
agent P4 = 0;                                                          (*)
* ---------
agent Q1 = a.Q2 + b.Q4;   In Q1, Q2, Q3, and Q4 there is an a-cycle:   (*)
agent Q2 = a.Q3;          (see also line (!) below)                    (*)
agent Q3 = a.Q4;              <-------a--------                        (*)
agent Q4 = a.Q2;            Q2 --a--> Q3 --a--> Q4                     (*)

* ---------
agent R1 = a.R2;              <---a---
agent R2 = a.R1 + a.0;      R1 ---a--> R2 ---a--> 0
```

```
* ---------
agent T1 = a.T2;              <---a---
agent T2 = a.T1;            T1 ---a--> T2
* ---------
agent T0 = a.T0;
* ------------------------------------------------------------------------
agent A = a.0;              A ---a--> 0

agent D = a.E;              a--- G <--a
agent E = b.F + a.G;        |         |
agent G = a.D;              V         |          <--b
agent F = b.F;              D ---a--> E ---b--> F ---
*
* ------------------------------------------------------------------------
prop  Ba = [a]F;          P1: false       Q1: false
prop  Bb = [b]F;          P2: true        Q3: true
*
* ------------------------------------------------------------------------
* Key to the names of properties:    M:min,  N:max,  d:diamond,  b:box.
* Thus, for instance,  Mda:min,diamond,a     Nbb:max,box,b
* ---------
*
prop Mda = min(X.<a>X);   A,B,C: false   Q1,R1,T1,T0,D,E,F,G: false
prop Nda = max(X.<a>X);   A,P1,F: false  B,C,Q1,R1,T1,T0,D,E,G: true  (!)
prop Mba = min(X.[a]X);   A,P1,F: true   B,C,Q1,R1,T1,T0,D,E,G: false
prop Nba = max(X.[a]X);   A,B,C: true    D,E,F,G: true
* ---------

prop Mdb = min(X.<b>X);   A,B,C: false   D,E,F,G: false
prop Ndb = max(X.<b>X);   A,B,C: false   D,G: false          E,F: true
prop Mbb = min(X.[b]X);   A,B,C: true    D,G: true           E,F: false
prop Nbb = max(X.[b]X);   A,B,C: true    D,E,F,G: true
* ------------------------------------------------------------------------

prop MM = min(X.<a>(min(Y.<b>Y)) & <a><a><a>X);   D: false   E,G: false
prop MN = min(X.<a>(max(Y.<b>Y)) & <a><a><a>X);   D: false   E,G: false
prop NM = max(X.<a>(min(Y.<b>Y)) & <a><a><a>X);   D: false   E,G: false
prop NN = max(X.<a>(max(Y.<b>Y)) & <a><a><a>X);   D: true    E,G: false
* ------------------------------------------------------------------------

prop MM1  = min(X.<a>(min(Y.<b>Y)) | <a><a><a>X); D: false   E,G: false
prop MN1a = min(X.<a>(max(Y.<b>Y)) | <a><a><a>X); D: true    E,G: false
prop MN1b = min(X.<a>(max(Y.<b>Y)  | <a><a>X));   D: true    E,G: false
prop MN1c = min(X.<a>(max(Y.<b>Y)  | <a>X));      D: true    E,G: true
prop NM1  = max(X.<a>(min(Y.<b>Y)  | <a><a>X));   D: true    E,G: true
prop NN1  = max(X.<a>(max(Y.<b>Y)  | <a><a>X));   D: true    E,G: true
* ------------------------------------------------------------------------

prop Mor   = min(X.<a>X | <a>T);         Q1,R1,T1,T0: true
prop Mand  = min(X.<a>X & <a>T);         Q1,R1,T1,T0: false
* ---------

* prop Maa = min(X.<a><a>X);              Q1,R1,T1,T0: false
prop M2or  = min(X.<a><a>X | <a>T);      Q1,R1,T1,T0: true
prop M2and = min(X.<a><a>X & <a>T);      Q1,R1,T1,T0: false
prop M3or  = min(X.[a]F | <a>X);         R1,R2: true         T1:false
* ------------------------------------------------------------------------
```

```
prop Ev(Phi)  = min(X.(Phi | (<->T & [-]X)));                    Eventually
prop Inv(Phi) = max(Y.(Phi & [-]Y));                             Inevitably

prop Fa = Inv(<a>T);              T1,T2: true           A,B,C,R1: false
prop Ft = Inv(T);                 T1,T2,A,B,C,R1: true
prop Fb = Ev(<b>T);               D,P1,Q1: true         P2: false
prop Fc = Inv(Ev(<b>T));          D: true               P1,Q1: false
prop Fd = max(X.T & <->X);        R1: true              A: false
* ========================================================================
* cwb
* input "examples.cwb";
* checkprop(P1,Nda);                                  (see line (!) above)
* false;
* checkprop(Q1,Nda);                                  (see line (!) above)
* true;
* ========================================================================
```

5. Implementation of a Local Model Checker

In this section we consider a Prolog program, called `lmc.pl`, which implements a local model checker. This `lmc.pl` program checks whether or not a given modal μ-calculus assertion holds in a process (or a state) of a given finite-state process (see [**19**, Chapter 14]).

In our program `lmc.pl`, in order to parse and interpret the input string, we use Definite Clause Grammars [**1**]. In that formalism, having defined a syntactic category, say `binding(B)`, we need to call a Prolog goal of the form: `binding(B,"...",[])`, for binding the Prolog variable `B` to the term obtained by parsing the string `"..."`. Note that the Prolog predicate `binding` requires a third argument which is an empty list.

For any process '`Process`' and any list '`Definitions`' of process definitions, we have that the modal μ-calculus assertion '`Assertion`' holds if and only if the atom `sat(Process, Assertion, Definitions, true)` holds.

We also have that the modal μ-calculus assertion '`Assertion`' does not hold if and only if the atom `sat(Process, Assertion, Definitions, false)` holds.

In the program `lmc.pl` below, the reader will find several comments that will help him to understand the Prolog code. The characters `/*` indicate the beginning of a comment and the characters `*/` indicate the end of a comment.

At the end of the program, we have added, among other comments, the proof that Peterson algorithm ensures mutual exclusion. The mutual exclusion property has been expressed by the modal μ-calculus formula:

$$\neg\mu X.\big(\langle in\rangle\,(\langle in\rangle\,\mathbf{true})\ \vee\ \langle\,\mathbf{.}\,\rangle\,X\big)$$

that is, by pushing $\neg$ inside and substituting $\neg X$ for X,

$$\nu X.\big([in]\,([in]\,\mathbf{false})\ \wedge\ [\,\mathbf{.}\,]\,X\big).$$

Note that these formulas are different from those on page 290, because here we assume, as in the program on page 271, that in Peterson algorithm the two processes perform the same *in* action to enter their critical section, and perform the same *out*

action to exit their critical section. Analogously to what we indicated on page 290, mutual exclusion of Peterson algorithm can also be expressed by the formula:

$$\neg\mu X.\big(\langle out\rangle\,(\langle out\rangle\,\mathbf{true}) \vee \langle \mathbf{.}\rangle\,X\big)$$

that is,

$$\nu X.\big([out]\,([out]\,\mathbf{false}) \wedge [\mathbf{.}]\,X\big)$$

because we can have two consecutive *out* actions iff two agents have been in their critical section at the same time.

Note also that in program `lmc.pl` the negation has been denoted by - (minus), not by ~ (tilde), as in the program `peterson-mucalculus.cwb` on page 290.

```
/**
 * ========================================================================
 *                          LOCAL MODEL CHECKER
 * filename: lmc.pl
 *
 * ======================================================================== */

/* Generator of new symbols.                                               */

:- assert(current_index(0)).      /* this is executed at compile time. */

gensym(NewName) :-
   (current_index(I) -> retract(current_index(I)) ; I = 0),
   I1 is I + 1, assert(current_index(I1)),
   name(I1,NameI1), name(NewName,[120, 95 | NameI1]).  /* 120 95 is x_ */

/* ------------------------------------------------------------------------
 * Appending lists.                                                        */

append([],L,L).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).

/* ------------------------------------------------------------------------
 * Deleting all occurrences of an element from a list.                     */

del(_,[],[]).
del(X,[X|L],T)    :- !, del(X,L,T).
del(X,[Y|Ys],[Y|Zs]) :- del(X,Ys,Zs).

/* ------------------------------------------------------------------------
 * Member of a list.                                                       */

member(X,[X|_]) :- !.
member(X,[_|T]) :- member(X,T).

/* ------------------------------------------------------------------------
 * Union of two sets:
 * the sets are represented as lists without repetitions.
 *
 * Note that the order of the elements in a list is significant,
 * while in a set the order of the elements is not significant.            */

union([],Ys,Ys).
union([X|Xs],Ys,Zs) :- member(X,Ys), !, union(Xs,Ys,Zs).
union([X|Xs],Ys,[X|Zs]) :- \+ member(X,Ys), union(Xs,Ys,Zs).
```

```
/**
 * ------------------------------------------------------------------------
 * sat(Process,Assertion,Definitions,T)
 *
 * We use the if-then-else construct: P -> Q ; R.
 * Recall that the construct: P -> Q ; R  only explores the first
 * solution to the goal P.
 * sat(P,B,Defs,T) never fails and after solving sat(P,B,Defs,T),
 * we have that the variable T is bound either to tt or ff.
 * ------------------------------------------------------------------------
 */
sat(_P,ff,_,T) :- T = ff, !.
sat(_P,tt,_,T) :- T = tt, !.

/* ------------------------------------------------------------------------
/* setting the atomic assertions 'a' and 'not(b)', denoted at(a) and
 * not(at(b)), respectively, for process ide(p).
 * For other atomic assertions and/or processes do likewise.          */

sat(ide(p),at(a),_,T) :- T = tt, !. % <---------------------- clause (c1)
sat(ide(p),at(b),_,T) :- T = ff, !. % <---------------------- clause (c2)

/* ------------------------------------------------------------------- */
sat(P,not(B),Defs,NT)    :- sat(P,B,Defs,T), !, (T = tt -> NT = ff ;
                            (T = ff -> NT = tt   )), !.

sat(P,and(B1,B2),Defs,T) :- sat(P,B1,Defs,T1), !, (T1 = ff -> T = ff ;
                            (sat(P,B2,Defs,T2) -> T = T2 )), !.
sat(P,or(B1,B2),Defs,T)  :- sat(P,B1,Defs,T1), !, (T1 = tt -> T = tt ;
                            (sat(P,B2,Defs,T2) -> T = T2 )), !.

sat(P,diam(Act,B),Defs,T)     :- trans(P,Act,Defs,Q), sat(Q,B,Defs,T1),
                                 T1 = tt, !, T = tt.
sat(_P,diam(_Act,_B),_Defs,T) :- T = ff, !.
sat(P,ex_diam(B),Defs,T)      :- trans(P,_,Defs,Q), sat(Q,B,Defs,T1),
                                 T1 = tt, !, T = tt.
sat(_P,ex_diam(_B),_Defs,T)   :- T = ff, !.

sat(P,box(Act,B),Defs,T) :- sat(P,not(diam(Act,not(B))),Defs,T), !.
sat(P,ex_box(B),Defs,T)  :- sat(P,not(ex_diam(not(B))),Defs,T), !.

sat(P,nu(var(X),Set,B),Defs,T) :-
             member(P,Set) -> (T = tt) ;
                              (union([P],Set,Set1),
                               subst(var(X),nu(var(X),Set1,B),B,NB),
                               sat(P,NB,Defs,T)), !.

/* Set is a list of bindings. For instance, for processes q1t and p1
 * Set is the following list of pairs:
 * [ [ide(q1t), sum([dot(out(a),ide(q1t)), dot(in(c),ide(q1t)),
 *                  dot(in(d),ide(q1f))])],
 *   [ide(p1),  dot(out(c),dot(out(u),ide(p11)))] ]                   */

sat(P,mu(var(X),Set,B),Defs,T) :-
     subst(var(X),not(var(X)),B,NB),
     sat(P,not(nu(var(X),Set,not(NB))),Defs,T), !.
```

```
/* The following clause makes 'failure to prove sat(...)' equivalent to
   'sat(...) does not hold', that is, we assume negation-as-failure. This
   clause should be the LAST clause for sat/4. To see when it is called,
   use, instead, for instance: sat(_P,_S,_D,T) :- print('.'), T=ff, !.
   This clause should never be called.                                  */

sat(_P,_S,_D,T) :- T = ff, !.  % <--------------------------------------
/* ---------------------------------------------------------------------
 * lookup(ProcessIdentifier, ProcessDefinition, ProcessDefinitionList) */

lookup(ide(P),Def,[[ide(Q), Def1]|_T]) :- P = Q, Def = Def1, !.
lookup(ide(P),Def,[[ide(Q), _Def]| T]) :- \+ (P = Q), lookup(ide(P),Def,T).

/* ---------------------------------------------------------------------
 * Action names                                                         */
act(in(_)).
act(out(_)).
act(tau).
/* ---------------------------------------------------------------------
 * compl(Action, ComplementedAction)
 * compl_list(Action, ComplementedAction)                               */

compl(in(P),out(P)).
compl(out(P),in(P)).
compl_list([],[]).
compl_list([A|As],[B|Bs]) :- compl(A,B), compl_list(As,Bs).
/* ---------------------------------------------------------------------
 * Transitions of CCS terms.
 * transition(Process,Action,ProcessDefinitionList,NewProcess)
 *
 * During transitions, the definitions are kept in the argument Defs,
 * that is the ProcessDefinitionList.
 * trans(P,Act,Def,Q) may fail.                                         */

trans(dot(Act,P),Act,_Defs,P) :- act(Act).

trans(sum([P|_]),Act,Defs,Q)  :- trans(P,Act,Defs,Q).
trans(sum([_|Ps]),Act,Defs,Q) :- trans(sum(Ps),Act,Defs,Q).

trans(par(P,Q),Act,Defs,par(P1,Q))  :-
        trans(P,Act,Defs,P1).
trans(par(P,Q),Act,Defs,par(P,Q1))  :-
        trans(Q,Act,Defs,Q1).
trans(par(P,Q),tau,Defs,par(P1,Q1)) :-
        trans(P,ActP,Defs,P1), trans(Q,ActQ,Defs,Q1),
        compl(ActP,ActQ).

trans(restr(P,L),Act,Defs,restr(P1,L)) :- compl_list(L,CL),
        union(L,CL,LCL), trans(P,Act,Defs,P1),
        \+ member(Act,LCL).

/* unfolding of an identifier.
   ide(P) should be of the form: ide(c), for some constant c.           */

trans(ide(P),Act,Defs,Q) :- P \= 0, lookup(ide(P),Def,Defs),
        trans(Def,Act,Defs,Q).

/* No other clauses for trans/4!  <---------------------------------- */
```

```
/* ------------------------------------------------------------------
 * Free Variables.
 * freeV(F,S) should never fail.
 * ------------------------------------------------------------------ */

freeV(tt,[]) :- !.
freeV(ff,[]) :- !.
freeV(at(_),[]) :- !.
freeV(var(X),[X]) :- !.

freeV(not(B),S)     :- freeV(B,S), !.
freeV(and(B1,B2),S) :- freeV(B1,S1), freeV(B2,S2), union(S1,S2,S), !.
freeV(or(B1,B2),S)  :- freeV(B1,S1), freeV(B2,S2), union(S1,S2,S), !.

freeV(diam(_,B),S)  :- freeV(B,S), !.
freeV(ex_diam(B),S) :- freeV(B,S), !.
freeV(box(_,B),S)  :- freeV(B,S), !.
freeV(ex_box(B),S) :- freeV(B,S), !.

freeV(nu(var(X),_,B),S) :- freeV(B,S1), del(X,S1,S), !.
freeV(mu(var(X),_,B),S) :- freeV(B,S1), del(X,S1,S), !.

/* ------------------------------------------------------------------
 * Substitution of 'Value' for 'Variable' in 'Term',
 * thereby deriving
 * 'NewTerm': subst(Variable, Value, Term, NewTerm).
 * subst(var(X),V,T,NT) should never fail.
 * ------------------------------------------------------------------ */

subst(_,_,tt,NT) :- !, NT = tt.
subst(_,_,ff,NT) :- !, NT = ff.
subst(_,_,at(P),NT) :- !, NT = at(P).

subst(var(X),V,var(X),V) :- !.
subst(var(X),_,var(Y),var(Y)) :- X \= Y, !.

subst(var(X),V,not(B),not(NB)) :- subst(var(X),V,B,NB), !.
subst(var(X),V,and(B1,B2),and(NB1,NB2)) :-
        subst(var(X),V,B1,NB1), subst(var(X),V,B2,NB2), !.
subst(var(X),V,or(B1,B2),or(NB1,NB2)) :-
        subst(var(X),V,B1,NB1), subst(var(X),V,B2,NB2), !.

subst(var(X),V,diam(Act,B),diam(Act,NB)) :-
        subst(var(X),V,B,NB), !.
subst(var(X),V,ex_diam(B),ex_diam(NB)) :-
        subst(var(X),V,B,NB), !.

subst(var(X),V,box(Act,B),box(Act,NB)) :-
        subst(var(X),V,B,NB), !.
subst(var(X),V,ex_box(B),ex_box(NB)) :-
        subst(var(X),V,B,NB), !.

/* ------------------------------------------------------------------
 * Substitution for Maximal and Minimal fixpoints.                  */

subst(var(X),_T,nu(var(X),Set,B),nu(var(X),Set,NB)) :- NB = B, !.
```

```
subst(var(X),V,nu(var(Y),Set,B),nu(var(Z),Set,NB)) :-
         X \= Y, freeV(V,FV), freeV(B,FB),
         member(Y,FV), member(X,FB), gensym(Z),
         subst(var(Y),var(Z),B,B1), subst(var(X),V,B1,NB), !.

subst(var(X),V,nu(var(Y),Set,B),nu(var(Y),Set,NB)) :-
         X \= Y, freeV(V,FV), freeV(B,FB),
         (\+ member(Y,FV) ; \+ member(X,FB)),
         subst(var(X),V,B,NB), !.

subst(var(X),V,mu(var(Y),Set,B),mu(var(Z),Set,NB)) :-
         subst(var(X),V,nu(var(Y),Set,B),nu(var(Z),Set,NB)), !.
/**
 * ========================================================================
 * Use of DEFINITE CLAUSE GRAMMAR for the input.
 *
 * Correctness of the local model checker requires that
 * the given assertion should have DISTINCT bound variables. <<==========
 *
 * Do not insert blank spaces in the input string!
 * Note that we can add extra round parentheses `(' and `)' around
 * assertions and processes.
 * ------------------------------------------------------------------------
 *
 * assertion   f ::= tt    |   ff    | <----------- truth  values
 *                   {d}   |           <----------- atomic assertion
 *                   -f    |  (f*f)  |  (f+f) |
 *                   <a>f  |   <>f   |  [a]f  |  []f  <--- a is an input-
 *                   nx.f  |  mx.f   |                      output action
 *                   (f)
 *
 * atomic assertion    d    ::= a   |  ...  |  z
 *
 * input-output action ioact ::= act |  !act
 *                      act   ::= a   |  ...  |  z
 *
 * variable             x     ::= a   |  ...  |  z
 * ------------------------------------------------------------------------
 * process     p ::= a.p  |  sum([p,p,...])  |  p|p  |
 *                   (p[a,b,...]) |   <------------------ for restriction
 *                   i    |           (a, b,... are input-output actions)
 *                   (p)
 *
 * process-identifier i ::= 0  |  letter(letter+digit)*
 *
 * binding:           i=p
 * definitions:       [binding, binding, ...]
 * ------------------------------------------------------------------------ */
/* ------------------------------------------------------------------------
 * parsing an assertion:                                                   */

assn(B) --> "{", atomic_assn(B1), "}",         {B = at(B1)}.

assn(B) --> "(", assn(B1), "*", assn(B2), ")", {B = and(B1,B2)}.
assn(B) --> "(", assn(B1), "+", assn(B2), ")", {B = or(B1,B2)}.
assn(B) --> "-", assn(B1),                     {B = not(B1)}.

assn(B) --> "<>", assn(B1),                    {B = ex_diam(B1)}.
```

```
assn(B) --> "<", ioact(A), ">", assn(B1),      {B = diam(A,B1)}.

assn(B) --> "[]", assn(B1),                    {B = ex_box(B1)}.
assn(B) --> "[", ioact(A), "]", assn(B1),      {B = box(A,B1)}.

assn(B) --> "n", boolvar(X), ".", assn(B1),    {B = nu(var(X),[],B1)}.
assn(B) --> "m", boolvar(X), ".", assn(B1),    {B = mu(var(X),[],B1)}.

assn(B) --> "(", assn(B1), ")",                {B = B1}.

assn(B) --> truthval(B).
assn(B) --> boolvar(B1),                       {B = var(B1)}.

/* parsing an atomic assertion                                          */
atomic_assn(X) --> [C],      {"a"=<C, C=<"z", name(X,[C])}.

/* parsing an truth value: it is either "tt" or "ff"                    */

truthval(X) --> [C,C], {C = 116, name(X,[C,C])}.% this is "tt". 116 is t.
truthval(X) --> [C,C], {C = 102, name(X,[C,C])}.% this is "ff". 102 is f.

/* parsing a boolean variable:                                          */
boolvar(X) -->  [C],        {"a"=<C, C=<"z", name(X,[C])}.

/* parsing an input-output action:                                      */
ioact(X) -->  act(X1),      {X = in(X1)}.
ioact(X) -->  "!", act(X1), {X = out(X1)}.
act(X)   -->  [C],          {"a"=<C, C=<"z", name(X,[C])}.

/* ---------------------------------------------------------------------
 * parsing a process:                                                   */

binding(B)  --> procname(F), "=", proc(T),            {B = [F,T]}.

proc(P)     --> ioact(X), ".", proc(P1),              {P = dot(X,P1)}.
proc(P)     --> "sum([",  proclist(L), "])",          {P = sum(L)}.
proc(P)     --> "(",  proc(P1), "|", proc(P2), ")",   {P = par(P1,P2)}.
proc(P)     --> "(",  proc(P1), "[", actlist(L), "])", {P = restr(P1,L)}.
proc(P)     --> "(",  proc(P1), ")",                  {P = P1}.
proc(P)     --> procname(P1),                         {P = P1}.

proclist(L) --> proc(P), ",", proclist(Ps),           {L = [P|Ps]}.
proclist(L) --> proc(P),                              {L = [P]}.

actlist(L) --> ioact(X), ",", actlist(Xs),            {L = [X|Xs]}.
actlist(L) --> ioact(X),                              {L = [X]}.

/* ---------------------------------------------------------------------
 * A process identifier is either 0 or a sequence of one letter followed
 * by zero or more letters or digits.                                   */

procname(F) --> letter(F1), procname1(F2),
                {name(F1,N1), name(F2,N2), append(N1,N2,N3), name(F0,N3),
                              F = ide(F0)}.   %   <------- here is ide(_)

procname(F)  --> letter(F1), {F = ide(F1)}.
procname(F)  --> "0",        {F = ide(0)}.
```

```
procname1(F) --> letterdigit(F1), procname1(F2),
           {name(F1,N1), name(F2,N2), append(N1,N2,N3), name(F,N3)}.
procname1(F) --> letterdigit(F).

letterdigit(F) --> letter(F).
letterdigit(F) --> digit(F).
letter(X) --> [C], {"a"=<C, C=<"z", name(X,[C])}.
                                /* e.g., letter(X,"y",[]) gives X = y */
digit(X)  --> [C], {"0"=<C, C=<"9", X is C - "0"}.
                                /* e.g., digit(X,"5",[])  gives X = 5 */
/**
 * ========================================================================
 *                          Various tests
 * tt stands for true.  If T is tt then the property holds.
 * ff stands for false. If T is ff then the property does not hold.
 * ======================================================================== */
/* P = a.P
 * To show: P |= nu X. {} (<a> tt  /\ [a]X).     T is tt          (1)
 * To show: P |= mu X. {} (<a> tt  /\ [a]X).     T is ff          (2)  */

w1(T) :- sat(ide(p),
           nu(var(x),[], and(diam(in(a), tt), box(in(a),var(x))) ),
           [[ide(p), dot(in(a),ide(p))]],
         T).                                  /* T is tt          (1)  */
w2(T) :- sat(ide(p),
           mu(var(x),[], and(diam(in(a), tt), box(in(a),var(x))) ),
           [[ide(p), dot(in(a),ide(p))]],
         T).                                  /* T is ff          (2)  */

/* ------------------------------------------------------------------------
 * P = a.Q;   Q = a.P;
 * To show: P |= nu X. {} <a>X                   T is tt          (3)
 * To show: P |= mu X. {} <a>X                   T is ff          (4)  */

w3(T) :- binding(P,"p=a.q",[]), binding(Q,"q=a.p",[]),
         sat(ide(p),
           nu(var(x),[],diam(in(a),var(x))),
           [P,Q],
         T).                                  /* T is tt          (3)  */

w4(T) :- binding(P,"p=a.q",[]), binding(Q,"q=a.p",[]),
         sat(ide(p),
           mu(var(x),[],diam(in(a),var(x))),
           [P,Q],
         T).                                  /* T is ff          (4)  */

/* inevitably <a> tt, that is, nu X. {} (<a> tt /\ [-]X)), holds for P */
w5(T) :- binding(P,"p=a.q",[]), binding(Q,"q=a.p",[]),
         sat(ide(p),
           nu(var(x),[],(and(diam(in(a),tt),ex_box(var(x))))),
           [P,Q],
         T).                                 /* T is tt           (5)  */

/* ------------------------------------------------------------------------
 * P = a.Q + a.0;   Q = a.P;
 * To show: P |= nu X. {} (<a> tt /\ <a> <a> X)  T is tt          (6)
 * To show: P |= mu X. {} (<a> tt /\ <a> <a> X)  T is ff          (7)  */
```

```
w6(T) :- binding(P,"p=sum([a.q,a.0])",[]), binding(Q,"q=a.p",[]),
         sat(ide(p),
            nu(var(x),[], and(diam(in(a),tt),
                              diam(in(a),diam(in(a),var(x))) )),
            [P,Q],
         T).                                  /* T is tt          (6)  */

w7(T) :- binding(P,"p=sum([a.q,a.0])",[]), binding(Q,"q=a.p",[]),
         sat(ide(p),
            mu(var(x),[], and(diam(in(a),tt),
                              diam(in(a),diam(in(a),var(x))) )),
            [P,Q],
         T).                                  /* T is ff          (7)  */
/* ------------------------------------------------------------------------
 * P = a.P + a.0;
 * To show: P |= nu X. {} (<a> tt /\ <a>  X)     T is tt          (8)
 * To show: P |= mu X. {} (<a> tt /\ <a>  X)     T is ff          (9)  */

w8(T) :- binding(P,"p=sum([a.p,a.0])",[]),
         sat(ide(p),
            nu(var(x),[], and(diam(in(a),tt),
                              diam(in(a),var(x)) )),
            [P],
         T).                                  /* T is tt          (8)  */

w9(T) :- binding(P,"p=sum([a.p,a.0])",[]),
         sat(ide(p),
            mu(var(x),[], and(diam(in(a),tt),
                              diam(in(a),var(x)) )),
            [P],
         T).                                  /* T is ff          (9)  */

/* inevitably <a> tt, that is, nu X. {} (<a> tt /\ [-]X)),
   does NOT hold for P                                                  */
w10(T) :- binding(P,"p=sum([a.p,a.0])",[]),
          sat(ide(p),
             nu(var(x),[],(and(diam(in(a),tt),ex_box(var(x)))),
              [P],
          T).                                 /* T is ff          (10)  */

/* ------------------------------------------------------------------------
 * P = a.Q;   Q = a.P + a.0;
 * To show: P |= nu X. {} ([a] ff \/ <a> X)          T is tt     (11)
 * To show: P |= mu X. {} ([a] ff \/ <a> X)          T is tt     (12)
 * To show: P |= nu X. {} (<a><a> tt /\ <a><a> X)    T is tt     (13)
 * To show: P |= mu X. {} (<a><a> tt /\ <a><a> X)    T is ff     (14)  */

w11(T) :- binding(P,"p=a.q",[]), binding(Q,"q=sum([a.p,a.0])",[]),
          sat(ide(p),
            nu(var(x),[], or(box(in(a),ff), diam(in(a),var(x)))),
            [P,Q],
          T).                                 /* T is tt          (11)  */

w12(T) :- binding(P,"p=a.q",[]), binding(Q,"q=sum([a.p,a.0])",[]),
          sat(ide(p),
            mu(var(x),[], or(box(in(a),ff), diam(in(a),var(x)))),
            [P,Q],
          T).                                 /* T is tt          (12)  */
```

```
w13(T) :- binding(P,"p=a.q",[]), binding(Q,"q=sum([a.p,a.0])",[]),
          sat(ide(p),
            nu(var(x),[], and(diam(in(a),diam(in(a),tt)),
                              diam(in(a),diam(in(a),var(x))))),
            [P,Q],
          T).                                  /* T is tt         (13)  */

w14(T) :- binding(P,"p=a.q",[]), binding(Q,"q=sum([a.p,a.0])",[]),
          sat(ide(p),
            mu(var(x),[], and(diam(in(a),diam(in(a),tt)),
                              diam(in(a),diam(in(a),var(x))))),
            [P,Q],
          T).                                  /* T is ff         (14)  */

/* inevitably <a> tt, that is, nu X. {} (<a> tt /\ [-]X)),
   does NOT hold for P                                                   */

w15(T) :- binding(P,"p=a.q",[]), binding(Q,"q=sum([a.p,a.0])",[]),
          sat(ide(p),
             nu(var(x),[],(and(diam(in(a),tt),ex_box(var(x))))),
              [P,Q],
          T).                                 /* T is ff          (15)  */
/**
 * ------------------------------------------------------------------------
 * Sender   =  a.Sender1
 * Sender1  = !b.( d.Sender +  c.Sender1)
 * Medium   =  b.(!c.Medium + !e.Medium)
 * Receiver =  e.f.!d.Receiver
 * System   = (Sender | Medium | Receiver)\{b,c,d,e}
 *
 * To show: System |= inv ([f] ev (<a> tt))
 *          where: inv(A) = nu X. {} (A /\ [.]X)
 *            and  ev(A)  = mu Y. {} (A \/ (<.> tt /\ [.]Y)),
 * that is, to show:
 * System |= nu X.{}(([f] mu Y.{}(<a>tt \/ (<.>tt /\ [.]Y))) /\ [.]X)
 *                                                                  (16)
 * w16(T) holds: T is tt. (16) holds.                                */

w16(T) :- assn(F, "nx.(([f](my.((<a>tt)+((<>tt)*[]y))))*[]x)",       []),
     binding(Sender,  "sender=a.sender1",                            []),
     binding(Sender1, "sender1=!b.sum([d.sender,c.sender1])",        []),
     binding(Medium,  "medium=b.sum([!c.medium,!e.medium])",         []),
     binding(Receiver,"receiver=e.f.!d.receiver",                    []),
     binding(System,  "system=(((sender|medium)|receiver)[b,c,d,e])",[]),
     sat(ide(system),F,
            [Sender, Sender1, Medium, Receiver, System],
     T).                                  /* T is tt         (16)  */
/* ------------------------------------------------------------------------
 * To show:
 * System |= nu X.{}(([a] mu Y.{}(<f>tt \/ (<.>tt /\ [.]Y))) /\ [.]X)
 *                                                                  (17)
 * Note: (17) is (16) with 'a' and 'f' interchanged.
 * w17(T) does NOT hold: T is ff. (17) does not hold.
 *
 * This is due to the fact that Sender1 and Medium may 'talk to each
 * other' forever: Sender1 always chooses 'b. c.Sender1   and
 *                 Medium  always chooses  b.'c.Medium               */
```

```
w17(T) :- assn(F, "nx.(([a](my.((<f>tt)+((<>tt)*[]y))))*[]x)",       []),
     binding(Sender,  "sender=a.sender1",                            []),
     binding(Sender1, "sender1=!b.sum([d.sender,c.sender1])",        []),
     binding(Medium,  "medium=b.sum([!c.medium,!e.medium])",         []),
     binding(Receiver,"receiver=e.f.!d.receiver",                    []),
     binding(System,  "system=(((sender|medium)|receiver)[b,c,d,e])",[]),
     sat(ide(system),F,
        [Sender, Sender1, Medium, Receiver, System],
     T).                                    /* T is ff         (17)  */

/* ------------------------------------------------------------------------
 * P1 = a.P2 + b.P4;  P2 = a.P3 + a.P4;  P3 = a.P4;  P4 = 0;
 * To show: P1 |= nu X. {} <a>X                  T is ff         (18)  */


w18(T) :- sat(ide(p1),
             nu(var(x),[],diam(in(a), var(x))),
             [[ide(p1), sum([dot(in(a),ide(p2)), dot(in(b),ide(p4))])],
              [ide(p2), sum([dot(in(a),ide(p3)), dot(in(a),ide(p4))])],
              [ide(p3), dot(in(a),ide(p4))],
              [ide(p4), ide(0)]],
             T).                            /* T is ff         (18)  */

/* ------------------------------------------------------------------------
 * Q1 = a.Q2 + b.Q4;  Q2 = a.Q3;  Q3 = a.Q4;  Q4 = a.Q2;
 * To show: Q1 |= nu X. {} <a>X                  T is tt         (19)  */

w19(T) :- sat(ide(p1),
         nu(var(x),[],diam(in(a), var(x))),
         [[ide(p1), sum([dot(in(a),ide(p2)), dot(in(b),ide(p4))])],
          [ide(p2), dot(in(a),ide(p3))],
          [ide(p3), dot(in(a),ide(p4))],
          [ide(p4), dot(in(a),ide(p2))]],
         T).                                /* T is tt         (19)  */


/* ------------------------------------------------------------------------
 * S = a.R;  R = a.S + b.0;
 * To show: S |= mu X. {} <a>(<b> tt \/ <a>X)         T is tt    (20)
 * To show: S |= mu X. {} <a>(<b> tt /\ <a>X)         T is ff    (21)
 * To show: S |= nu X. {} <a>(<b> tt \/ <a>X)         T is tt    (22)
 * To show: S |= nu X. {} <a>(<c> tt \/ <a>X)         T is tt    (23)
 * To show: S |= nu X. {} <a>(<b> tt /\ <a>X)         T is tt    (24)
 * To show: S |= nu X. {} <a>(<c> tt /\ <a>X)         T is ff    (25)  */

w20(T) :- sat(ide(s),
        mu(var(x),[],diam(in(a), or(diam(in(b),tt),diam(in(a),var(x))))),
        [[ide(s), dot(in(a),ide(r))],
         [ide(r), sum([dot(in(a),ide(s)), dot(in(b),ide(0))])]],
        T).                                 /* T is tt         (20)  */

w21(T) :- sat(ide(s),
        mu(var(x),[],diam(in(a), and(diam(in(b),tt),diam(in(a),var(x))))),
        [[ide(s), dot(in(a),ide(r))],
         [ide(r), sum([dot(in(a),ide(s)), dot(in(b),ide(0))])]],
        T).                                 /* T is ff         (21)  */
```

```
w22(T) :- sat(ide(s),
       nu(var(x),[],diam(in(a), or(diam(in(b),tt),diam(in(a),var(x)))),
       [[ide(s), dot(in(a),ide(r))],
        [ide(r), sum([dot(in(a),ide(s)), dot(in(b),ide(0))])]],
       T).                                  /* T is tt          (22)  */

w23(T) :- sat(ide(s),
       nu(var(x),[],diam(in(a), or(diam(in(c),tt),diam(in(a),var(x)))),
       [[ide(s), dot(in(a),ide(r))],
        [ide(r), sum([dot(in(a),ide(s)), dot(in(b),ide(0))])]],
       T).                  /* Notice: in(c).  T is tt          (23)  */

w24(T) :- sat(ide(s),
       nu(var(x),[],diam(in(a), and(diam(in(b),tt),diam(in(a),var(x)))),
       [[ide(s), dot(in(a),ide(r))],
        [ide(r), sum([dot(in(a),ide(s)), dot(in(b),ide(0))])]],
       T).                                  /* T is tt          (24)  */

w25(T) :- sat(ide(s),
       nu(var(x),[],diam(in(a), and(diam(in(c),tt),diam(in(a),var(x)))),
       [[ide(s), dot(in(a),ide(r))],
        [ide(r), sum([dot(in(a),ide(s)), dot(in(b),ide(0))])]],
       T).                  /* Notice: in(c).  T is ff          (25)  */

/* ------------------------------------------------------------------------
 * S = a.R;  R = a.S + b.W;  W = b.W;
 * To show: S |= mu X. {} <a>((nu Y. {} <b>Y) \/ <a>X)  T is tt  (26)
 * To show: R |= mu X. {} <a>((nu Y. {} <b>Y) \/ <a>X)  T is ff  (27)
 * To show: S |= nu X. {} <a>((nu Y. {} <b>Y) \/ <a>X)  T is tt  (28)
 * To show: R |= nu X. {} <a>((nu Y. {} <b>Y) \/ <a>X)  T is tt  (29)
 *
 * See also file "examples.cwb".  Processes D, E, and F of "examples.cwb"
 * are similar (not equal) to our processes S, R, and W, respectively.
 * prop MN1b = min(X.<a>(max(Y.<b>Y) | <a><a>X));  D: true E: false (26')
 * prop NN1  = max(X.<a>(max(Y.<b>Y) | <a><a>X));  D: true E: true  (29')
 *                                                                      */
w26(T) :- sat(ide(s),
       mu(var(x),[],diam(in(a), or(nu(var(y),[],diam(in(b),var(y))),
                                   diam(in(a),var(x))))),
       [[ide(s), dot(in(a),ide(r))],
        [ide(r), sum([dot(in(a),ide(s)), dot(in(b),ide(w))])],
        [ide(w), dot(in(b),ide(w))]],
       T).                                  /* T is tt          (26)  */

w27(T) :- sat(ide(r),
       mu(var(x),[],diam(in(a), or(nu(var(y),[],diam(in(b),var(y))),
                                   diam(in(a),var(x))))),
       [[ide(s), dot(in(a),ide(r))],
        [ide(r), sum([dot(in(a),ide(s)), dot(in(b),ide(w))])],
        [ide(w), dot(in(b),ide(w))]],
       T).                                  /* T is ff          (27)  */
w28(T) :- sat(ide(s),
       nu(var(x),[],diam(in(a), or(nu(var(y),[],diam(in(b),var(y))),
                                   diam(in(a),var(x))))),
       [[ide(s), dot(in(a),ide(r))],
        [ide(r), sum([dot(in(a),ide(s)), dot(in(b),ide(w))])],
        [ide(w), dot(in(b),ide(w))]],
       T).                                  /* T is tt          (28)  */
```

```
w29(T) :- sat(ide(r),
        nu(var(x),[],diam(in(a), or(nu(var(y),[],diam(in(b),var(y))),
                                    diam(in(a),var(x))))),
        [[ide(s), dot(in(a),ide(r))],
         [ide(r), sum([dot(in(a),ide(s)), dot(in(b),ide(w))])],
         [ide(w), dot(in(b),ide(w))]],
        T).                                    /* T is tt         (29)  */
/* ========================================================================
 *                          Testing the transitions
 * ---------------------------------------------------------------------- */
/* a.0 | !a.0                                                             */
t1(Act,T) :- trans(par(ide(p),ide(q)), Act,
          [[ide(p), dot(in(a), ide(0))], [ide(q), dot(out(a), ide(0))]],
          T).
/* ?- t1(A,T).
 * A = in(a),
 * T = par(ide(0),ide(q)) ? ;
 * A = out(a),
 * T = par(ide(p),ide(0)) ? ;
 * A = tau,
 * T = par(ide(0),ide(0)) ? ;
 * no
                                                                          */
/* (a.0 | !a.0)\[!a]  ------------------------------------------------- */
t2(Act,T) :- trans(restr(par(ide(p),ide(q)),[out(a)]), Act,
          [[ide(p), dot(in(a), ide(0))], [ide(q), dot(out(a), ide(0))]],
          T).
/* ?- t2(A,T).
 * A = tau,
 * T = restr(par(ide(0),ide(0)),[out(a)]) ? ;
 * no                                                                     */

/* (a.0 | !a.0)\[b]   ------------------------------------------------- */
t3(Act,T) :- trans(restr(par(ide(p),ide(q)),[in(b)]), Act,
          [[ide(p), dot(in(a), ide(0))], [ide(q), dot(out(a), ide(0))]],
          T).
/* ?- t3(A,T).
 * A = in(a),
 * T = restr(par(ide(0),ide(q)),[in(b)]) ? ;
 * A = out(a),
 * T = restr(par(ide(p),ide(0)),[in(b)]) ? ;
 * A = tau,
 * T = restr(par(ide(0),ide(0)),[in(b)]) ? ;
 * no                                                                     */

/* (a.0 + a.0 + b.0)     ---------------------------------------------- */
t4(Act,T) :- trans(sum([dot(in(a),ide(0)), dot(in(a),ide(0)),
                        dot(in(b),ide(0))]),Act,[],T).
/* ?- t4(A,T).
 * A = in(a),
 * T = ide(0) ? ;
 * A = in(a),
 * T = ide(0) ? ;
 * A = in(b),
 * T = ide(0) ? ;
 * no                                                                     */
```

```
/* p = a.0  and  q = !a.0.  Thus, p -> 0 with action in(a). ---------- */
t5(Act,T) :- trans(ide(p), Act,
          [[ide(p), dot(in(a), ide(0))], [ide(q), dot(out(a), ide(0))]],
          T).

/* ?- t5(A,T).
 * A = in(a),
 * T = ide(0) ? ;
 * no                                                                  */
/* ====================================================================
 *              Testing satisfiability of atomic assertions
 * p = a.0    (a is an action here)
 * For process p the atomic assertion a holds and the atomic assertion b
 * does not hold (see the above clauses (c1) and (c2)).
 * -------------------------------------------------------------------
 */
w30(T) :- sat(ide(p),
          and(at(a),diam(in(a),tt)),
          [[ide(p), dot(in(a),ide(0))]],
          T).                                  /* T is tt         (30)  */

w31(T) :- assn(F, "({b}*<a>tt)",    []),
         binding(P, "p=a.0", []),
         sat(ide(p), F, [P], T).              /* T is ff         (31)  */

/* ====================================================================
 *             Peterson Algorithm: Mutual Exclusion Property
 *
 * mutex(T) holds: T is tt.
 *
 * We can replace the assertion:
 *      "nx.(([i]([i]ff))*([]x))"
 * by the assertion (see mutex1(T) below):
 *      "-(mx.((<i>(<i>tt))+(<>x)))"
 * ------------------------------------------------------------------- */

 mutex(T) :- assn(F, "nx.(([i]([i]ff))*([]x))",             []),

         binding(Q1t, "q1t=sum([!a.q1t,c.q1t,d.q1f])",      []),
         binding(Q1f, "q1f=sum([!b.q1f,c.q1t,d.q1f])",      []),
         binding(Q2t, "q2t=sum([!e.q2t,g.q2t,h.q2f])",      []),
         binding(Q2f, "q2f=sum([!f.q2f,g.q2t,h.q2f])",      []),
         binding(S1,  "s1=sum([!r.s1,u.s1,w.s2])",          []),
         binding(S2,  "s2=sum([!s.s2,u.s1,w.s2])",          []),

         binding(P1,  "p1=!c.!u.p11",                       []),
         binding(P11, "p11=sum([e.p11,r.p11,f.p12,s.p12])", []),
         binding(P12, "p12=i.o.!d.p1",                      []),
         binding(P2,  "p2=!g.!w.p21",                       []),
         binding(P21, "p21=sum([a.p21,s.p21,b.p22,r.p22])", []),
         binding(P22, "p22=i.o.!h.p2",                      []),

         binding(Peterson,
           "pet=((p1|(p2|(q1f|(q2f|s1))))[a,b,c,d,e,f,g,h,r,s,u,w])",[]),
         sat(ide(pet), F,
                [Q1t,Q1f,Q2t,Q2f,S1,S2,P1,P11,P12,P2,P21,P22,Peterson],
             T).                                  /* T is tt             */
```

```
/* ------------------------------------------------------------------ */

mutex1(T) :- assn(F, "-(mx.((<i>(<i>tt))+(<>x)))",          []),

         binding(Q1t, "q1t=sum([!a.q1t,c.q1t,d.q1f])",      []),
         binding(Q1f, "q1f=sum([!b.q1f,c.q1t,d.q1f])",      []),
         binding(Q2t, "q2t=sum([!e.q2t,g.q2t,h.q2f])",      []),
         binding(Q2f, "q2f=sum([!f.q2f,g.q2t,h.q2f])",      []),
         binding(S1,  "s1=sum([!r.s1,u.s1,w.s2])",          []),
         binding(S2,  "s2=sum([!s.s2,u.s1,w.s2])",          []),

         binding(P1,  "p1=!c.!u.p11",                       []),
         binding(P11, "p11=sum([e.p11,r.p11,f.p12,s.p12])", []),
         binding(P12, "p12=i.o.!d.p1",                      []),
         binding(P2,  "p2=!g.!w.p21",                       []),
         binding(P21, "p21=sum([a.p21,s.p21,b.p22,r.p22])", []),
         binding(P22, "p22=i.o.!h.p2",                      []),

         binding(Peterson,
           "pet=((p1|(p2|(q1f|(q2f|s1))))[a,b,c,d,e,f,g,h,r,s,u,w])",[]),
         sat(ide(pet), F,
                [Q1t,Q1f,Q2t,Q2f,S1,S2,P1,P11,P12,P2,P21,P22,Peterson],
            T).                                  /* T is tt              */

/* ====================================================================
 *               Testing the order of possible actions
 * p = a.0 + a.(b.0)         p can do an 'a' action and then a 'b' action
 * --------------------------------------------------------------------
 */
w32(T) :- binding(P,"p=sum([a.0,a.(b.0)])",[]),
          sat(ide(p),
             diam(in(a),diam(in(b),tt)),
             [P],
          T).                                    /* T is tt          (32)  */
/* ==================================================================== */
go1 :-
w1(T1), print(' 01tt:'), print(T1), w2(T2), print(' 02ff:'), print(T2),
w3(T3), print(' 03tt:'), print(T3), w4(T4), print(' 04ff:'), print(T4),
w5(T5), print(' 05tt:'), print(T5), w6(T6), print(' 06tt:'), print(T6),
w7(T7), print(' 07ff:'), print(T7), w8(T8), print(' 08tt:'), print(T8),
w9(T9), print(' 09ff:'), print(T9),
nl, w10(T10), print(' 10ff:'), print(T10),
w11(T11),print(' 11tt:'),print(T11), w12(T12),print(' 12tt:'),print(T12),
w13(T13),print(' 13tt:'),print(T13), w14(T14),print(' 14ff:'),print(T14),
w15(T15),print(' 15ff:'),print(T15), w16(T16),print(' 16tt:'),print(T16),
w17(T17),print(' 17ff:'),print(T17), w18(T18),print(' 18ff:'),print(T18),
nl, w19(T19), print(' 19tt:'), print(T19),
w20(T20),print(' 20tt:'),print(T20), w21(T21),print(' 21ff:'),print(T21),
w22(T22),print(' 22tt:'),print(T22), w23(T23),print(' 23tt:'),print(T23),
w24(T24),print(' 24tt:'),print(T24), w25(T25),print(' 25ff:'),print(T25),
w26(T26),print(' 26tt:'),print(T26), w27(T27),print(' 27ff:'),print(T27),
nl, w28(T28),print(' 28tt:'),print(T28),
w29(T29),print(' 29tt:'),print(T29), w30(T30),print(' 30tt:'),print(T30),
w31(T31),print(' 31ff:'),print(T31), w32(T32),print(' 32tt:'),print(T32),
(T1=tt, T2=ff, T3=tt, T4=ff, T5=tt, T6=tt, T7=ff, T8=tt, T9=ff, T10=ff,
T11=tt, T12=tt, T13=tt, T14=ff, T15=ff, T16=tt, T17=ff, T18=ff, T19=tt,
T20=tt, T21=ff, T22=tt, T23=tt, T24=tt, T25=ff, T26=tt, T27=ff, T28=tt,
T29=tt, T30=tt, T31=ff, T32=tt) -> (print(' *yes')); true.
```

```
/* -------------------- */

ww(Form,T) :- assn(F, Form,      []),
   binding(Sender,  "sender=a.sender1",                              []),
   binding(Sender1, "sender1=!b.sum([c.sender1,d.sender])",          []),
   binding(Medium,  "medium=b.sum([!e.medium,!c.medium])",           []),
   binding(Receiver,"receiver=e.f.!d.receiver",                      []),
   binding(System,  "system=((sender|(medium|receiver))[b,c,d,e])",[]),
   sat(ide(system),F,
          [Sender, Sender1, Medium, Receiver, System],
   T).

go2 :-  print(' Communication Channel:'),
   F1  = "nx.(([a](my.((<a>tt)+((<>tt)*[]y))))*[]x)",
   F16 = "nx.(([f](my.((<a>tt)+((<>tt)*[]y))))*[]x)",   % as in w16(T)
   F17 = "nx.(([a](my.((<f>tt)+((<>tt)*[]y))))*[]x)",   % as in w17(T)
   F4  = "nx.(([f](my.((<f>tt)+((<>tt)*[]y))))*[]x)",
   ww(F1,T1), print(' ff:'), print(T1),
   ww(F16,T16), print(' 16tt:'), print(T16),
   ww(F17,T17), print(' 17ff:'), print(T17),
   ww(F4,T4), print(' ff:'), print(T4),
   (T1=ff, T16=tt, T17=ff, T4=ff) -> print(' *yes ') ; true.
/* -------------------- */
go3 :- print(' Peterson Protocol:'),
   mutex(T), print(' tt:'), print(T),
   mutex1(T1), print(' tt:'), print(T1),
   (T=tt, T1=tt) -> print(' *yes') ; true.

go :- statistics(runtime,[T11,_T12]), go1, nl, go2, nl, go3,
   statistics(runtime,[T21,_T22]), nl,
   print(' time: '), T is T21-T11, print(T), print(' ms').
/* ========================================================================= */
/* In order to run all the tests, type:  go.
   You should get (the value of 3080 ms for the elapsed time may
   be different):
| ?- go.
 01tt:tt 02ff:ff 03tt:tt 04ff:ff 05tt:tt 06tt:tt 07ff:ff 08tt:tt 09ff:ff
 10ff:ff 11tt:tt 12tt:tt 13tt:tt 14ff:ff 15ff:ff 16tt:tt 17ff:ff 18ff:ff
 19tt:tt 20tt:tt 21ff:ff 22tt:tt 23tt:tt 24tt:tt 25ff:ff 26tt:tt 27ff:ff
 28tt:tt 29tt:tt 30tt:tt 31ff:ff 32tt:tt *yes
 Communication Channel: ff:ff 16tt:tt 17ff:ff ff:ff *yes
 Peterson Protocol: tt:tt tt:tt *yes
 time: 3080 ms
yes
========================================================================= */
```

Appendix A. Complete Lattices and Complete Partial Orders

In this section we show that to choose complete lattices, rather than complete partial orders, for providing the denotational semantics of conditionals, is problematic.

Indeed, let us consider the following four terms:

($a1$) **if** t **then** (**if** t **then** e_0 **else** e_1) **else** (**if** t **then** e_2 **else** e_3)

($a2$) **if** t **then** e_0 **else** e_3

($b1$) **if** t_0 **then** (**if** t_1 **then** e_0 **else** e_1) **else** (**if** t_1 **then** e_2 **else** e_3)

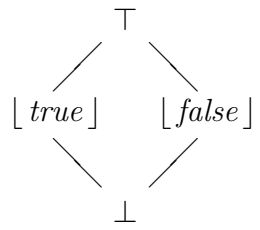
($b2$) **if** t_1 **then** (**if** t_0 **then** e_0 **else** e_2) **else** (**if** t_0 **then** e_1 **else** e_3)

We expect that the denotational semantics of the **if-then-else** construct should satisfy the following two equations:

(α) $[\![a1]\!]\rho = [\![a2]\!]\rho$

(β) $[\![b1]\!]\rho = [\![b2]\!]\rho$

Let us consider the complete lattice B of the form:



and let us assume that the semantics of the **if-then-else** construct is given by a function *Cond* (the reader should not confuse this function with the function with the same name we have introduced on page 94). Now *Cond* should denote a function from $B \times B \times B$ to B such that for any triple $\langle t_0, t_1, t_2 \rangle$ of boolean terms we have:

$[\![\, \textbf{if}\, t_0\, \textbf{then}\, t_1\, \textbf{else}\, t_2 \,]\!]\, \rho \;=\; \mathit{Cond}([\![\, t_0 \,]\!]\, \rho, [\![\, t_1 \,]\!]\, \rho, [\![\, t_2 \,]\!]\, \rho)$

We have that: $\mathit{Cond}(\lfloor \mathit{true} \rfloor, x, y) = x$, $\mathit{Cond}(\lfloor \mathit{false} \rfloor, x, y) = y$. Since *Cond* should be strict on the first argument, we also have that: $\mathit{Cond}(\bot, x, y) = \bot$. By monotonicity on the first argument we have the following two options for the definition of $\mathit{Cond}(\top, x, y)$:

Option ($\sqcup$) : $\quad \mathit{Cond}(\top, x, y) = x \sqcup y$

Option ($\top$) : $\quad \mathit{Cond}(\top, x, y) = \top$

Now let us consider the pair of terms $a1$ and $a2$. In order to establish (α) we cannot take Option ($\sqcup$), because for $[\![t]\!]\rho = \top$ we have that $[\![\, a1 \,]\!]\rho = [\![\, a2 \,]\!]\rho$ holds iff

$([\![\, e_0 \,]\!]\rho \sqcup [\![\, e_1 \,]\!]\rho) \sqcup ([\![\, e_2 \,]\!]\rho \sqcup [\![\, e_3 \,]\!]\rho) = ([\![\, e_0 \,]\!]\rho \sqcup [\![\, e_3 \,]\!]\rho)$ holds,

and, in general, this does not hold. Indeed, if $[\![\, e_0 \,]\!]\rho = [\![\, e_3 \,]\!]\rho = \bot$ then $[\![\, e_1 \,]\!]\rho \sqcup [\![\, e_2 \,]\!]\rho$ may be different from $\bot$.

In order to establish (β) we cannot take Option ($\top$), because for $[\![t_1]\!]\rho = \top$ we have that $[\![\, b1 \,]\!]\rho = [\![\, b2 \,]\!]\rho$ holds iff $Cond([\![\, t_0 \,]\!]\, \rho, \top, \top) = \top$ holds and, in general, this does not hold. Indeed, if $[\![\, t_0 \,]\!]\, \rho = \bot$ then $Cond([\![\, t_0 \,]\!]\, \rho, \top, \top) = \bot$.

Thus, since we can take neither Option ($\sqcup$) nor Option ($\top$), we conclude that for the semantics of the **if-then-else** construct it is problematic to use complete lattices, instead of cpo's.

Appendix B. Scott Topology

Given a cpo $(D, \sqsubseteq)$ we define a topology, called *Scott topology*, as follows (see also [**19**]).

DEFINITION 1.1. [**Scott-open Set**] We say that a subset V of D is a *Scott-open set* (or an *open set*, for short) iff
(i) (*upward closure*) for all d in D, for all e in D, *if* $(d \sqsubseteq e \wedge d \in V)$ *then* $e \in V$, and
(ii) (*finite approximant of limit points*) for all ω-chains $d_0 \sqsubseteq d_1 \sqsubseteq \ldots$ in D,
if $\bigsqcup_{i\in\omega} d_i \in V$ *then* there exists $n \in \omega$ such that $d_n \in V$.

PROPOSITION 1.2. [**Empty Subset**] Given a cpo $(D, \sqsubseteq)$, the empty subset $\emptyset$ of D is an open set.

PROOF. Points (i) and (ii) of Definition 1.1 hold because $\forall x \in \emptyset.\, P(x)$ holds for every unary predicate P. □

PROPOSITION 1.3. [**Whole Subset**] Given a cpo $(D, \sqsubseteq)$, the subset of D which is D itself, is an open set.

PROOF. Points (i) and (ii) of Definition 1.1 hold because the conclusions of the implications hold. □

PROPOSITION 1.4. [**Finite or Infinite Union of Sets**] Let us consider a collection $\{D_i \mid i \in I\}$ of open sets in the cpo $(D, \sqsubseteq)$. Then, $\bigcup_{i\in I} D_i$ is an open set in the cpo $(D, \sqsubseteq)$.

PROOF. (i) Take any $d \in \bigcup_{i\in I} D_i$. Let $d \in D_k$, for some $k \in I$. Take any e such that $d \sqsubseteq e$. Since D_k is open $e \in D_k$. Thus, $e \in \bigcup_{i\in I} D_i$.

(ii) Take an ω-chain $d_0 \sqsubseteq d_1 \sqsubseteq \ldots$ in D. Assume $\bigsqcup_{m\in\omega} d_m \in \bigcup_{i\in I} D_i$. Now we have to show that there exists $n \in \omega$ such that $d_n \in \bigcup_{i\in I} D_i$.

Indeed, since $\bigsqcup_{m\in\omega} d_m \in \bigcup_{i\in I} D_i$ we have that there exists $k \in I$ such that $\bigsqcup_{m\in\omega} d_m \in D_k$. Since D_k is open, there exists $n \in \omega$ such that $d_n \in D_k$. Thus, there exists $n \in \omega$ such that $d_n \in \bigcup_{i\in I} D_i$. □

REMARK 1.5. Since the index set I can be finite or infinite, we have that the *finite* or *infinite union* of open sets is an open set.

PROPOSITION 1.6. [**Finite Intersection of Sets**] Let us consider a *finite* collection $\{D_i \mid i \in F\}$ of open sets in the cpo $(D, \sqsubseteq)$. Then, $\bigcap_{i\in F} D_i$ is an open set in the cpo $(D, \sqsubseteq)$.

PROOF. (i) Take any $d \in \bigcap_{i\in F} D_i$. Thus, $d \in D_k$, for all $k \in F$. Take any e such that $d \sqsubseteq e$. We have that $e \in D_k$, for all $k \in F$, because D_k is an open set. Thus, $e \in \bigcap_{i\in F} D_i$.

(ii) Take an ω-chain $d_0 \sqsubseteq d_1 \sqsubseteq \ldots$ in D. Assume $\bigsqcup_{m\in\omega} d_m \in \bigcap_{i\in F} D_i$. Now we have to show that there exists $n \in \omega$ such that $d_n \in \bigcap_{i\in F} D_i$. Indeed, since $\bigsqcup_{m\in\omega} d_m \in \bigcap_{i\in F} D_i$ we have that for all $k \in F$, $\bigsqcup_{m\in\omega} d_m \in D_k$. Now, by definition of an open set, we have that for all $k \in F$, there exists $n_k \in \omega$ such that $d_{n_k} \in D_k$. Let us consider the maximum value, call it n_{max}, in the set $\{n_k \,|\, k \in F\}$. We have that $d_{n_{max}} \in \bigcap_{i\in F} D_i$, because for all $k \in F$, D_k is an open set (and thus, upward closed). Therefore, Condition (ii) of Definition 1.1 holds and the proof is completed. □

REMARK 1.7. The intersection of an *infinite* number of open sets need not be an open set. In particular, the maximum value in the set $\{n_k \,|\, k \in F\}$ may not exist because $\{n_k \,|\, k \in F\}$ may be an infinite, unbounded set.

PROPOSITION 1.8. [**Open Set Not Below a Given Element**] For any element $d \in D$, the set $\{x \,|\, x \not\sqsubseteq d\}$ is open.

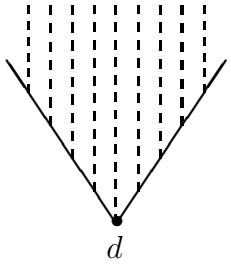
PROOF. (i) Take any $d_1 \in \{x \,|\, x \not\sqsubseteq d\}$. Take any e_1 such that $d_1 \sqsubseteq e_1$ we have to show that $e_1 \not\sqsubseteq d$. Let us assume the contrary, that is, $e_1 \sqsubseteq d$. Since $d_1 \sqsubseteq e_1$, by transitivity, we have that $d_1 \sqsubseteq d$, contrary to our assumption that $d_1 \in \{x \,|\, x \not\sqsubseteq d\}$.

(ii) Take an ω-chain $d_0 \sqsubseteq d_1 \sqsubseteq \ldots$ in D such that $\bigsqcup_{i\in\omega} d_i \in \{x \,|\, x \not\sqsubseteq d\}$. We have to show that there exists $n \in \omega$ such that $d_n \not\sqsubseteq d$. Let us assume the contrary, that is, for all $k \in \omega$, $d_k \sqsubseteq d$. By definition of the least-upper-bound $\bigsqcup$, we get that $\bigsqcup_{i\in\omega} d_i \sqsubseteq d$, which contradicts the hypothesis that $\bigsqcup_{i\in\omega} d_i \in \{x \,|\, x \not\sqsubseteq d\}$. □

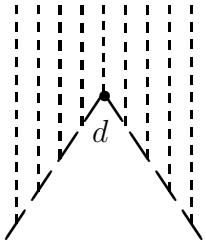
DEFINITION 1.9. [**Limit Point and Finite Point**] An element d of a cpo is said to be a *limit point* iff there exists an ω-chain $d_0 \sqsubseteq d_1 \sqsubseteq \ldots$ such that: (i) $d = \bigsqcup_{i\in\omega} d_i$, and (ii) $\forall i \in \omega.\, \exists j \in \omega.\, i < j \wedge d_i \neq d_j$.

An element of a cpo is said to be a *finite point* iff it is not a limit point.

REMARK 1.10. We have that $d \notin \{x \,|\, x \not\sqsubseteq d\}$, because $d \sqsubseteq d$ (see also Figure 5).



$V_1 = \{x \mid d \sqsubseteq x \wedge x \text{ is a finite point}\}$



$V_2 = \{x \mid x \not\sqsubseteq d\}$

FIGURE 5. A pictorial view of the open sets: $V_1 = \{x \mid d \sqsubseteq x \wedge x$ is a finite point$\}$ and $V_2 = \{x \mid x \not\sqsubseteq d\}$. $d \in V_1$ and $d \notin V_2$.

DEFINITION 1.11. [**Topological Continuity**] A function is said to be *topologically continuous* iff for any open subset V of E its inverse image $f^{-1}(V)$ is an open set of D.

PROPOSITION 1.12. [**Continuity and Topological Continuity**] Let us consider the two cpo's D and E. A function f from D to E is *continuous* iff it is topologically-continuous.

PROOF. (*only-if part*) Take any f continuous from D to E. (i) Take any $d \in f^{-1}(V)$ (see also Figure 6). Take any $e \in D$ such that $d \sqsubseteq e$. We have to show that $e \in f^{-1}(V)$.

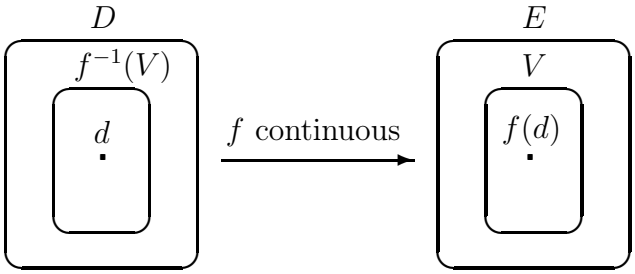


FIGURE 6. The inverse image $f^{-1}(V)$ of the open set V in E is an open set in D iff f is topologically continuous function from D to E.

We have that $f(d) \in V$ because $d \in f^{-1}(V)$. Since f is monotonic $f(d) \sqsubseteq f(e)$. Since V is open, $f(e) \in V$. Thus, $e \in f^{-1}(V)$.

(ii) Take an ω-chain $d_0 \sqsubseteq d_1 \sqsubseteq \ldots$ in D. Assume that $\bigsqcup_{i\in\omega} d_i \in f^{-1}(V)$, that is, $f(\bigsqcup_{i\in\omega} d_i) \in V$. We have to show that there exists $n \in \omega$ such that $d_n \in f^{-1}(V)$.

By monotonicity we have that $f(d_0) \sqsubseteq f(d_1) \sqsubseteq \ldots$ is an ω-chain in E. Since $f(\bigsqcup_{i\in\omega} d_i) \in V$ we have that $\bigsqcup_{i\in\omega} f(d_i) \in V$ (indeed, by continuity, $f(\bigsqcup_{i\in\omega} d_i) = \bigsqcup_{i\in\omega} f(d_i)$). Since V is open, there exists $k \in \omega$ such that $f(d_n) \in V$. Thus, there exists $k \in \omega$ such that $d_n \in f^{-1}(V)$.

(*if part*) Take any f topologically continuous. We have to show: (1) f is monotonic, and (2) f preserves limits.

(1) Take any $d \sqsubseteq e$ in D. We have to show that $f(d) \sqsubseteq f(e)$. We will assume the contrary and we will get a contradiction.

Let us assume that $f(d) \not\sqsubseteq f(e)$. We have that $V = \{x \mid x \not\sqsubseteq f(e)\}$ is an open set in E. Thus, $f(d) \in \{x \mid x \not\sqsubseteq f(e)\}$ and $f(e) \notin V$. Hence, $d \in f^{-1}(V)$ and $e \notin f^{-1}(V)$ Since f is topologically continuous, $f^{-1}(V)$ is an open set and $d \sqsubseteq e$, we have that $e \in f^{-1}(V)$. This contradicts that $e \notin f^{-1}(V)$.

(2) Take an ω-chain $d_0 \sqsubseteq d_1 \sqsubseteq \ldots$ in D. We have to show that $\bigsqcup_{i\in\omega} f(d_i) = f(\bigsqcup_{i\in\omega} d_i)$. Let us first show that

(2.1) $\bigsqcup_{i\in\omega} f(d_i) \sqsubseteq f(\bigsqcup_{i\in\omega} d_i)$ and then

(2.2) $f(\bigsqcup_{i\in\omega} d_i) \sqsubseteq \bigsqcup_{i\in\omega} f(d_i)$.

Proof of (2.1). Since for all $i \in \omega$, $d_i \sqsubseteq \bigsqcup_{i\in\omega} d_i$, by monotonicity, we get: for all $i \in \omega$, $f(d_i) \sqsubseteq f(\bigsqcup_{i\in\omega} d_i)$. Thus, $\bigsqcup_{i\in\omega} f(d_i) \sqsubseteq f(\bigsqcup_{i\in\omega} d_i)$.

Proof of (2.2). Assume the contrary, that is,

$$f(\textstyle\bigsqcup_{i\in\omega} d_i) \not\sqsubseteq \bigsqcup_{i\in\omega} f(d_i). \qquad (\dagger)$$

Now, consider the subset $V = \{x \mid x \not\sqsubseteq \bigsqcup_{i\in\omega} f(d_i)\}$ in E. V is an open set in E because: (a) $\bigsqcup_{i\in\omega} f(d_i) \in E$ (indeed, $f(d_0) \sqsubseteq f(d_1) \sqsubseteq \ldots$ is an ω-chain in E, and since E is a cpo, $\bigsqcup_{i\in\omega} f(d_i) \in E$), and (b) Proposition 1.8 on the facing page holds. In particular, we have that:

$$\textstyle\bigsqcup_{i\in\omega} f(d_i) \notin V. \qquad (\dagger\dagger)$$

By (†) we also have that $f(\bigsqcup_{i\in\omega} d_i) \in V$. Thus, $\bigsqcup_{i\in\omega} d_i \in f^{-1}(V)$. Since $f^{-1}(V)$ is an open set in D, there exists $k \in \omega$ such that $d_k \in f^{-1}(V)$. Thus, $f(d_k) \in V$. Now we have that $f(d_k) \sqsubseteq \bigsqcup_{i\in\omega} f(d_i)$ by definition of $\bigsqcup_{i\in\omega} f(d_i)$. Since V is an open set in E and $f(d_k) \in V$, we have that $\bigsqcup_{i\in\omega} f(d_i) \in V$, which contradicts (††). □

PROPOSITION 1.13. [**Characterization of Open Sets**] The open subsets of a cpo D are precisely those subsets which are equal to $f^{-1}(\top)$ for some continuous function $f : D \to \mathbf{O}$, where $\mathbf{O}$ is the cpo $\{\bot, \top\}$ with $\bot \sqsubseteq \top$.

PROOF. (i) Since a continuous function f is topologically continuous and for any topologically continuous function the inverse images of open sets are open sets, it should be the case that the subset $\{\top\}$ of $\mathbf{O}$ is an open set in $\mathbf{O}$. This is immediate.

(ii) We have to show that any open A of D can be mapped onto $\{\top\}$ by a continuous function f which maps every element of A to $\top$.

(ii.1) Take any $d \in A$. Take any e such that $d \sqsubseteq e$. Since A is open, $e \in A$. Since $f(d) = f(e) = \top$ we have that f is monotonic (indeed, $f(d) \sqsubseteq f(e)$, because both are $\top$).

(ii.2) Take any ω-chain $d_0 \sqsubseteq d_1 \sqsubseteq \ldots$ in D and assume that $\bigsqcup_{i\in\omega} d_i \in A$. Since A is an open set, there exists $k \in \omega$ such that $d_k \in A$. We have to show that $f(\bigsqcup_{i\in\omega} d_i) = \bigsqcup_{i\in\omega} f(d_i)$. Indeed, since $\bigsqcup_{i\in\omega} d_i \in A$ we have that $f(\bigsqcup_{i\in\omega} d_i) = \top$. We also have that there exists $k \in \omega$ such that $d_k \in A$ because $\bigsqcup_{i\in\omega} d_i \in A$ and A is an open set. Thus, there exists $k \in \omega$ such that $f(d_k) = \top$. Thus, $\bigsqcup_{i\in\omega} f(d_i) = \top$ because $f(d_k) \sqsubseteq \bigsqcup_{i\in\omega} f(d_i)$. □

As a consequence of Proposition 1.13 we can choose an open subset of the cpo D by choosing a continuous function from D to $\mathbf{O}$.

DEFINITION 1.14. [**Scott-closed Set**] A subset of a cpo is said to be a *Scott-closed set* (or a *closed set*, for short) iff it is the complement of a Scott-open subset.

THEOREM 1.15. [**Scott-closed Sets are Inclusive**] Every Scott-closed subset of a cpo is inclusive.

PROOF. Recall that a subset P of a cpo D is said to be *inclusive* iff for all ω-chains $d_0 \sqsubseteq d_1 \sqsubseteq \ldots$ in D, if $\forall n \in \omega.\, d_n \in P$ then $\bigsqcup_{n\in\omega} d_n \in P$ (see Definition 6.1 on page 103). Assume that P is a Scott-closed set and that for all ω-chains $d_0 \sqsubseteq d_1 \sqsubseteq \ldots$ we have that for all $n \in \omega$, $d_n \in P$. We will prove that $\bigsqcup_{n\in\omega} d_n \in P$ by contradiction.

Assume to the contrary that $\bigsqcup_{n\in\omega} d_n \in \overline{P}$, where $\overline{P}$ is the complement of P. Since, by definition, $\overline{P}$ is a Scott-open set, we also have that for the ω-chain $d_0 \sqsubseteq d_1 \sqsubseteq \ldots$ if $\bigsqcup_{n\in\omega} d_n \in \overline{P}$ then there exists $n \in \omega$ such that $d_n \in \overline{P}$. Thus, we have that there exists $n \in \omega$ such that $d_n \in \overline{P}$. But this contradicts the hypothesis that for all $n \in \omega$, $d_n \in P$. □

We have the following two Facts 1.17 and 1.18. First we need this definition.

DEFINITION 1.16. [**The set** Ω] Let ω be the set $\{0, 1, 2, \ldots\}$ of the natural numbers. Let Ω be the cpo consisting of the set $\omega \cup \{\infty\}$ with the partial order relation induced by the following chain: $0 \sqsubseteq 1 \sqsubseteq 2 \sqsubseteq \ldots \sqsubseteq \infty$.

FACT 1.17. [**Inclusive Subsets of** Ω] The inclusive subsets of Ω are either the finite subsets of Ω or the infinite subsets A of Ω which satisfy the following property:

if $(\forall n \in \omega.\, \exists m \in \omega.\, n < m \ \wedge\ m \in A)$ then $\infty \in A$. □

FACT 1.18. Let us consider the cpo Ω (see Definition 1.16 on the preceding page). We have that: (i) $\{\infty\}$ is an inclusive subset of Ω, and (ii) $\{\infty\}$ is not a Scott-closed subset of Ω.

PROOF. (i). Immediate. (ii) The fact that $\{\infty\}$ is not Scott-closed follows from the fact that its complement (w.r.t. Ω) is the subset $\{0, 1, 2, \ldots\}$ which is *not* Scott-open. Indeed, $\{0, 1, 2, \ldots\}$ does not satisfy Condition (i) of Definition 1.1, because $0 \sqsubseteq \infty$ and $\infty \notin \{0, 1, 2, \ldots\}$. □

Index

Bibliography

[1] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, New York, Second Edition, 1984.

[2] N. Dershowitz. Termination of Rewriting. *Journal of Symbolic Computation*, 3(1-2):69–116, 1987.

[3] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewod Cliffs, N.J., 1976.

[4] J. H. Gallier. On the Existence of Optimal Fixpoints. *Mathematical System Theory*, 13:207–215, 1979.

[5] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus*. London Mathematical Society. Cambridge University Press, 1986.

[6] J. Hsiang. Refutational Theorem Proving Using Term-Rewriting Systems. *Artificial Intelligence*, 25:255–300, 1985.

[7] G. Huet and D. C. Oppen. Equations and Rewrite Rules: A Survey. Tech. Report CSL-111, SRI International, Menlo Park, California, USA, 1980.

[8] P. J. Landin. The Mechanical Evaluation of Expressions. *Computer Journal*, 6(4):308–320, 1964.

[9] Z. Manna. *Mathematical Theory of Computation*. MacGraw-Hill, 1974.

[10] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[11] R. Milner. *Communicating and Mobile Systems*: *the π-calculus*. Cambridge University Press, 1999.

[12] E. Mendelson. *Introduction to Mathematical Logic*. Wadsworth & Brooks/Cole Advanced Books & Software, Monterey, California, USA. Third Edition. 1987.

[13] F. Moller and P. Stevens. *The Edinburgh Concurrency Workbench*. User Manual Version 7.1, Laboratory for Foundations of Computer Science, University of Edinburgh, Scotland, U.K., 1999. http://homepages.inf.ed.ac.uk/perdita/cwb//.

[14] D. Park. Concurrency and Automata on Infinite Sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183, London, UK. Springer-Verlag. 1981.

[15] A. Pettorossi. *Elements of Concurrent Programming*. Aracne Editrice. Third Edition. 2008.

[16] G. D. Plotkin. A Powerdomain Construction. *SIAM Journal on Computing*, 5(3):452–487, 1976.

[17] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[18] J. E. Stoy. *The Scott-Strachey Approach to Programming Languages*. The MIT Press, Cambridge, MA, 1977.

[19] G. Winskel. *The Formal Semantics of Programming Languages*: *An Introduction*. The MIT Press, Cambridge, Massachusetts, 1993.