A Proof System for a Sequential Object-Based Language

Carlos Camarão de Figueiredo

Technical Report UMCS–95–1–1

A Proof System for a Sequential Object-Based Language

Carlos Camarão de Figueiredo
Department of Computer Science, University of Manchester
Oxford Rd, Manchester M13 9PL, U.K.

December 16, 1994

Abstract

This thesis presents a proof system to support the formal verification of the correctness of sequential object-based programs written in a simple programming language named **A1**. The proof system uses a specification language, named **L1**, that is based on Hoare-style assertions.

The syntax and formal semantics of the programming and specification languages are given. A formal semantics of correctness formulas (for partial correctness) is then defined. Axiom schemes and proof rules associated with commands of **A1**, which compose a proof system named **P1**, are presented. Examples are shown where these axiom schemes and proof rules are used to derive program correctness. The proof system **P1** is shown to be sound according to the given semantics of **A1** and of the correctness formulas.

Finally, conclusions are drawn and suggestions for further work are made.

Chapter 1 provides a background for the rest of the thesis, introduces the main topics discussed and describes the motivations behind this work.

Chapter 2 presents a simple object-based programming language named **A1** and defines its syntax and semantics.

Chapter 3 presents a specification language named **L1** for reasoning about commands of **A1** (i.e. for reasoning about the effect of the execution of the commands in an appropriate computer) and defines its syntax and semantics. The syntax and formal semantics of *correctness formulas* (for partial correctness) are then defined, which give a formalization of the notion of programs satisfying specifications.

Chapter 4 presents a proof system that can be used to show the correctness of programs written in **A1**, according to specifications.

Chapter 5 presents some examples of the use of these rules to show program correctness.

Chapter 6 demonstrates the soundness of the proof system **P1**.

Chapter 7 draws some conclusions and makes suggestions for further work.

Chapter 1

Introduction

This chapter provides a background for the rest of the thesis, introduces the main topics discussed and describes the motivations behind the work.

Programming language definitions traditionally consist of two parts: syntax and semantics.

The syntax of a programming language defines which phrases (i.e. character sequences) are programs in that programming language and also defines the structure of phrases (i.e. how they are formed by means of immediate components, until elementary components are formed).

The semantics of a programming language defines the meaning of programs written in that language, refering to the execution of programs, i.e. refering to what would happen when such a program is executed in an appropriate computer.

Section 1.1 describes the motivations behind the *sequent calculus style* used for the syntax definition of **A1**.

Section 1.2 gives an introduction to formal semantic definitions in general and to the style of semantic definitions used in this thesis in particular.

Taking the view that object-based languages provide a good framework for the study of the problem of verification of correctness of programs that dynamically change data structures, Section 1.3 presents an introduction to object-based programming languages geared towards this context.

Section 1.4 presents an introduction to proof systems for imperative sequential programming languages and describes possibilities and difficulties in extending these proof systems to cope with object-based programming languages.

1.1 Syntax Definitions

The syntax of a programming language defines which phrases (i.e. character sequences) are programs in that programming language and also defines the structure of phrases (i.e. how they are formed by means of immediate components, until elementary components are formed).

This section describes the motivations behind the *sequent calculus style* used for the syntax definition of **A1**.[1]

It is well-known that a very significant limitation of widely used notations for syntax definition like BNF (Backus-Naur Formalism)[N^+63] or syntax diagrams is that they specify only *context-free* aspects of the syntax of a language, i.e. aspects which do not consider the context in which a symbol or sequence of symbols occur in a program. *Context-sensitive constraints*[2] such as name binding rules or type checking rules cannot be defined. An example of a name binding rule is the one that specifies that names must be

[1] This has been inspired from and is an extension of methods used in Tennent[Ten91, Sections 3.1 and 4.1] and Astesiano[Ast91, Section 2.1]. For descriptions of sequent calculi for propositional and predicate logics see e.g. Gallier[Gal86].

[2] Also called *context conditions*, or even "*static semantics*". (In our view, these constraints are clearly not part of semantics, since they are not part of the definition of the *meaning* of (syntactically well-formed) programs. In this case, defining context-sensitive constraints as below has an extra benefit of avoiding clutter in the semantics definition.)

declared before use; an example of a type checking rule is the one that specifies that, in an assignment command, the type of the variable must be compatible with the type of the expression.

There is no widely accepted method for specifying context-sensitive constraints.[3] Often, context conditions are expressed informally.[4] Unfortunately, an informal description is often incomplete or imprecise, particularly if it is not based on a formal description.

The sequent calculus style, which is presented below by means of a simple illustrative example, defines an inference system (more specifically, a sequent calculus) that uses sequents of the form $\Delta \vdash h \circ Pt$ or $\Delta \vdash h \circ Pt[h_1, \ldots, h_n]$, where $n \geq 0$ and Δ is an arbitrary sequence of formulas.[5] In these formulas h and $h_1, \ldots, h_n$ are *phrases* and Pt is a *phrase type*. The phrase h is called the *main phrase* of the formula. The phrases $h_1, \ldots, h_n$ are called *components* or *attributes* of phrase h.

The *sequents* have *succedents* (sequences of formulas on the right of $\vdash$) with just one formula. In lengthier inference rules, the premises are written each in a separate line (instead of all in a single line, separated by spaces, which is more usual).

The symbol Δ is used to denote arbitrary sequences of formulas. The notation $\Delta^{h_1,\ldots,h_n}$ can also be used, representing the sequence of formulas obtained by deleting from Δ any formula for which h_i ($i = 1, \ldots, n$) is the main phrase of the formula.

The formula $h \circ Pt$ is also used in a more general way to abbreviate the sentence "h is a well-formed phrase of type Pt",[6] meaning that for a specified Δ, the sequent $\Delta \vdash h \circ Pt$ is provable in the given inference system.[7]

The symbol $\equiv$ is used for definitions and to express syntactic identity (e.g. $\mathtt{1} + \mathtt{2} \equiv \mathtt{1} + \mathtt{2}$ but not $\mathtt{1} + \mathtt{2} \equiv \mathtt{2} + \mathtt{1}$).

To describe this style, we define below a very simple syntax of a "baby" language with extremely simple type checking and name binding rules.

Programs in this baby language have a single declaration and a single command. The declaration associates an identifier (here simply either x or y) with a type (here simply either `Bool` or `Bin`). The command (an assignment command) assigns a value to an identifier. The value may be a boolean literal (`true` or `false`) or a binary numeral (a sequence of `0`s and/or `1`s).

We want programs in the baby language to satisfy also the following context-sensitive constraints:

1. identifiers must be declared before use (in this small language, this means that the same identifier that is declared is used subsequently in the assignment command);
2. types in an assignment command must be *compatible* (in this small language, this means that the identifier in the assignment command must have type `Bool` if the value assigned to it is either `true` or `false`, and it must have type `Bin` if the value assigned to it is a binary numeral).

It is not difficult to see that the above informal style does not lead in general to precise, concise or complete definitions of context-sensitive constraints.

The syntax of this baby language can be defined using BNF as follows:

$$
\begin{array}{lcl}
< pg > & ::= & < d > ; < c > \\
< d > & ::= & < id > : < t > \\
< t > & ::= & \mathtt{Bool} \mid \mathtt{Bin} \\
< id > & ::= & x \mid y \\
< c > & ::= & < id > \leftarrow < v > \\
< v > & ::= & \mathtt{true} \mid \mathtt{false} \mid < n > \\
< n > & ::= & \mathtt{0} \mid \mathtt{1} \mid < n > \mathtt{0} \mid < n > \mathtt{1}
\end{array}
$$

[3] For some approaches, see e.g. [BJ82, Chapter 2][Ten81, Section 13.5],[Pag81, Sections 2.3.1 and 2.4.2],[Des84, Kap85],[Ast91, Sections 2.5 and 4.2],[vD91].

[4] See e.g. [JW74].

[5] Note that we have, in a sequent calculus, sequents and formulas. The sequents are the basic syntactic elements of the inference system (usually, but not in a sequent calculus, the basic syntactic elements of an inference system are called *formulas* — see Section 1.4.1).

In the example below only formulas of the first form and of the second form with $n = 1$ are used.

[6] "Phrase type Pt" is shortened to just "type Pt" when no confusion can arise.

[7] See Section 1.4.1 for the notion of provability in an inference system.

The simple syntax above, including the (above informally defined) context-sensitive constraints, can be defined, using a sequent calculus style, for example as follows.

We define first the syntactic meta-variables and corresponding phrase types:

Meta-variable	Phrase type	
c	C	commands
t	T	types
id	Id	identifiers
n, v	V	values

Let A denote a list of formulas which correspond to assumptions about predefined names:

$$\begin{aligned} A \equiv\ & \mathtt{true} \circ V[\mathtt{Bool}], \\ & \mathtt{false} \circ V[\mathtt{Bool}], \\ & \mathtt{0} \circ V[\mathtt{Bin}], \\ & \mathtt{1} \circ V[\mathtt{Bin}], \\ & \mathtt{Bool} \circ T, \\ & \mathtt{Bin} \circ T \end{aligned}$$

We have:

$$\frac{x \circ Id, y \circ Id \vdash id \circ Id \quad A \vdash t \circ T \quad A, id \circ Id \vdash c \circ C[t]}{\vdash id : t;\ c \circ Pg}$$

$$\frac{\Delta \vdash id \circ Id \quad \Delta \vdash v \circ V[t]}{\Delta \vdash id \leftarrow v \circ C[t]}$$

$$\frac{\Delta \vdash n \circ V[\mathtt{Bin}]}{\Delta \vdash n\mathtt{0} \circ V[\mathtt{Bin}]}$$

$$\frac{\Delta \vdash n \circ V[\mathtt{Bin}]}{\Delta \vdash n\mathtt{1} \circ V[\mathtt{Bin}]}$$

As an example, the first inference rule above can be read as follows: "if id is either x or y, and if t is a type, and if from the assumptions about predefined names and the fact that id is an identifier (that has been declared) we can deduce that c is a command of type t, then we can infer that $id : t;\ c$ is a program".[8]

The example below shows a derivation of the syntactically well-formed program $x : \mathtt{Bin};\ x \leftarrow \mathtt{10}$:

$$\frac{x \circ Id, y \circ Id \vdash x \circ Id \quad A \vdash \mathtt{Bin} \circ T \quad \dfrac{A, x \circ Id \vdash x \circ Id \quad \dfrac{A, x \circ Id \vdash \mathtt{1} \circ V[\mathtt{Bin}]}{A, x \circ Id \vdash \mathtt{10} \circ V[\mathtt{Bin}]}}{A, x \circ Id \vdash x \leftarrow \mathtt{10} \circ C[\mathtt{Bin}]}}{\vdash x : \mathtt{Bin};\ x \leftarrow \mathtt{10} \circ Pg}$$

1.2 Semantic Definitions

This section gives an introduction to formal semantic definitions and serves as a background to the rest of the thesis in this area.

Formal semantic definitions serve mainly as a basis for understanding and reasoning about how programs behave, for all persons involved with the language in general, be they language designers, implementers or programmers. In particular, they can reveal situations which are unforeseen and not apparent in informal definitions.

There exist three main methods for defining semantics of programming languages: operational, denotational and axiomatic. They are described below.

[8] The word "type" in the phrase "command of type t" is the name of an attribute of the command.

The method used in this thesis is the denotational one. We do not intend to work directly with axiomatic semantic definitions, nor with operational ones. In connection with axiomatic semantics, we will show in Chapter 6 that the proof rules presented for the verification of program correctness are consistent with the given denotational semantics of the programming language and of the specification language.

1.2.1 Notation

We introduce in this section some notational conventions used in relation to semantic definitions throughout this thesis.

Sets and Functions

Basic notation of set theory is used; in particular, we use the following:

Notation	For
X^*	the set of all finite sequences of elements of X
$\wp(X)$	the set of all subsets of X
$X - Y$	$\{x \mid x \in X \text{ and } x \notin Y\}$, the set difference of X and Y
X^n	$X \times \cdots \times X$ (n times)
$X \to Y$	the set of all total functions from X to Y
$f : X \to Y$	the total function f from X to Y
$f : X \rightharpoonup Y$	the partial function f from X to Y
$x \mapsto y$	an element (x, y) of a function
$\mathtt{dom}(f)$	the domain of a function f from X to Y, i.e. the set $\{x \mid (x,y) \in f \text{ for some } y \in Y\}$
$\mathtt{rng}(f)$	the range of a function f from X to Y, i.e. the set $\{y \mid (x,y) \in f \text{ for some } x \in X\}$
$f \dagger \{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$	letting f be a function from X to Y, the function f', from $X \cup \{x_1, \ldots, x_n\}$ to $Y \cup \{y_1, \ldots, y_n\}$, such that: $f'(x) = f(x)$, for $x \notin \{x_1, \ldots, x_n\}$, and $f'(x_i) = y_i$, for $i = 1, \ldots, n$ where $x_1, \ldots, x_n$ are assumed to be pairwise distinct
$\mathbf{Z}$	$\{\ldots, -1, 0, 1, \ldots\}$
$\mathbf{B}$	$\{true, false\}$
$\mathbf{N}$	$\{0, 1, \ldots\}$

A *predicate* is a total function (from any set) to $\mathbf{B}$.
A function $f : X \to Y$ is a *constant function* if, for some $y \in Y$, $f(x) = y$ for all $x \in X$.
A function $f : X \to X$ is an *identity function* if $f(x) = x$, for all $x \in X$.
For $g : X \to Y$ and $f : Y \to Z$, the *composition* $f \circ g : X \to Z$ of f and g is given by:

$$f \circ g(x) = f(g(x))$$

For $f : X \to X$, the *i-th iterate* of f, $i \in \mathbf{N}$, denoted by f^i, is defined inductively by:

$$\begin{aligned} &f^0(x) = x && \text{i.e } f^0 \text{ is an identity function} \\ &f^{i+1} = f \circ f^i \end{aligned}$$

For $i = 1, \ldots, n$, $n \geq 1$, and a cartesian product $X_1 \times \ldots \times X_n$, the *$i$-th projection function* is the function $pr_i : X_1 \times \ldots \times X_n \to X_i$ given by:

$$pr_i(x_1, \ldots, x_n) = x_i$$

We also use the notation $x \downarrow i$ for $pr_i(x)$.

$f' : X' \to Y'$ is an *extension* of $f : X \to Y$ (and, equivalently, f is a *restriction* of f') if $X \subseteq X'$ and $f'(x) = f(x)$ for all $x \in X$.

Let $f : X_1 \times \ldots \times X_n \to X_{n+1}$, $n \geq 1$, be a given function and, for $i = 1, \ldots, n+1$, let $X'_i = X_i \cup \{\perp_i\}$, where $\perp_i \notin X_i$. An extension $f' : X'_1 \times \ldots \times X'_n \to X'_{n+1}$ of f is *strict* if $f'(x_1, \ldots, \perp_i, \ldots, x_n) = \perp_{n+1}$ for all $i = 1, \ldots, n$.

Lambda notation

It is sometimes convenient to use a notation to define functions that provides a way of denoting functions without having to give a name to them. This notation uses λ-*terms* and is called the lambda notation. A function $f : X \to Y$ defined by $f(x) = x + 1$, for example, is given by the λ-term:

$$\lambda x.x + 1$$

The lambda notation we use is the "typed" one: in this notation, a type is associated with each λ-term. The types of λ-terms are not explicitly indicated but are given by the context in which they appear: we always explicitly define the signatures of functions before defining them. For example, the type of $\lambda x.x + 1$ above is $X \to Y$ (or, say, $X \times Y$). The exact syntax for expressing these types will not be of concern for us here.

The reasons for this approach are that untyped lambda notation is too general for the purposes of the definition of functions in this thesis and that it leads to difficult problems connected with self-application of λ-terms.[9]

1.2.2 Operational Semantics

Most of the early works on formal definitions of the semantics of programming languages are mainly characterized by being *operational*: the descriptions specify how programs are executed (on an abstract machine), rather than what are the results of the executions.

Specifying how programs are executed generally involves defining what constitutes a *configuration* (for the abstract machine) and defining a (transition) relation on the set of configurations.

If the transition relation is defined by means of a *calculus*[10] then the method is often called *structural operational semantics*[11].

1.2.3 Denotational Semantics

Denotational semantic definitions are aimed at specifying a value (or *denotation*) for each program.[12] Considering a *state* as a function assigning values to program variables, in the case of imperative programs, characterized by having operations that change the state, the denotation of a program must contain a characterization of the final states of the execution of a program, typically as a function of initial states.

Denotational semantic definitions are typically *compositional*: informally, a semantic definition is said to be compositional if the meaning of every (composite) phrase is expressed as a function of the meanings of its immediate sub-phrases.[13] An immediate benefit of a compositional semantic definition is in showing that the meaning of a (composite) phrase does not change if a sub-phrase is replaced by another with the same meaning. Another benefit is that it allows semantic properties to be proved by induction on the structure of phrases.

[9] Cf. e.g. [Bar84][Sto77, Chapter 5].

[10] I.e. a pair (A, R) where A is finite set of axiom schemes and R is a finite set of inference rules. See Section 1.4.

[11] The method is due to Hennessy and Plotkin[HP79, Plo81]. See also [Ast91], which contains discussions on terminology, with suggestion for the term *structural inductive semantics*.

[12] The work on denotational semantics may be traced back to [Fre92, Tar56, Car48, Car47]. The subject is covered in [Sto77, Sch86, LS87, Ten91, Win93].

[13] Formally, a semantic definition is said to be compositional if and only if it is a homomorphism from the syntactic algebra into a semantic algebra (i.e. any algebra over the same signature whose carriers are the sets of elements in given semantic domains associated to the syntactic elements of the language). For more details see [Ast91].

Programs may fail to terminate due to executions of constructs such as while-commands and recursively defined constructs (see below). When this can occur, then either the function that defines the meaning of a program must be defined as a partial function or its range must contain a special "undefined" value. Moreover, in the case of recursive programs, solutions to corresponding equations may not in general exist or be unique. Possible candidates for solutions to these equations need to be required to satisfy some properties in order that exactly one solution (the really interesting one, hopefully) be characterized. This is done in the theory of complete partial orders, using the Fixed Point Theorem (see Section 1.2.6). A classical example of a recursive program is the recursive factorial, whose corresponding equation can be written as follows:

$$f(x) = \begin{cases} 1 & \text{if x} = 0 \\ x * f(x-1) & \text{otherwise} \end{cases}$$

For non-deterministic and parallel programs, the denotation of a program is in general a set of values, and the semantic domain must then include powersets. If non-termination is possible, then the powersets must also include a special undefined value. These powerset domains defined along with a complete partial ordering are called *powerdomains*.[14]

When it's necessary to specify the evaluation ordering of program constructs, functions in denotational semantic definitions can use arguments called *continuations*. The style of denotational semantics is then called a continuation style. Otherwise, it is called a direct style. A continuation can be informally understood as "the meaning of the rest of the program", but it is best explained by means of an example (see Section 1.2.5).

The continuation style of denotational semantics was first developed for the definition of the semantics of "goto" commands[15], but their utility in controlling evaluation orderings has made them very useful in other cases, such as for example in the semantics of concurrent programs[16].

1.2.4 Axiomatic Semantics

Axiomatic semantics define functions from assertions to assertions (called "*predicate transformers*"), taking the view that everything about the semantics of programs is known if, for any initial state and any assertion p, the (sets of) computations that fail to terminate, that terminate in a state satisfying p and that terminate in a state satisfying $\neg p$ are known.[17] (The terms *wp* and *wlp* used below are derived from "weakest precondition" and "weakest liberal precondition", respectively[18].)

To each such set of computations there corresponds, for each given command c and assertion p, the following three assertions:[19]

wp.c.true: holds in the initial states for which no computation of c fails to terminate;

[14] See e.g. [Plo76].

[15] Cf. e.g. [Sto77, Chapter 11].

[16] Cf. e.g. [AR92].

[17] See [Eds90].

[18] Informally, an assertion p is weaker than p' if p is true in all cases in which p' is true (correspondingly, p' is said to be stronger than p).

[19] Note that *wp.c.true*, *wlp.c.p* and *wlp.c.¬p* can be seen below just as names of assertions.

Note also that, in denotational semantics, with respect to, for example, meaning functions

$$\mathbf{I}[\![c]\!] : \Sigma \rightharpoonup \Sigma$$

for a given command c, where Σ is the set of all states, and

$$\mathbf{I_0}[\![p]\!] : \Sigma \to \mathbf{B}$$

for a given assertion p, the weakest liberal precondition is usually defined in relation to c and $\mathbf{I_0}[\![p]\!]$ as the predicate $\psi : \Sigma \to \mathbf{B}$ given as follows:

$$\psi(\sigma) = \mathit{true} \text{ if and only if: } \begin{cases} \mathbf{I}[\![c]\!](\sigma) \text{ is undefined, or} \\ \mathbf{I}[\![c]\!](\sigma) \text{ is defined and } \mathbf{I_0}[\![p]\!]\big(\mathbf{I}[\![c]\!](\sigma)\big) = \mathit{true} \end{cases}$$

*wlp.c.p***:** holds in the initial states for which no computation of c terminates in a state satisfying $\neg p$;

wlp.c.$\neg$*p***:** holds in the initial states for which no computation of c terminates in a state satisfying p

The semantics of commands can thus be given by defining, for every command c, the assertion *wp.c.true* and the "predicate transformer" *wlp.c* . This is typically done inductively over the structure of commands. These predicate transformers must have some special properties, such as for example the fact that *wlp.c.true* must hold for every command c.

The axiomatic approach grew out of Floyd's work[Flo67] on attaching assertions to the links of flowcharts. The application of the axiomatic approach to the verification of program correctness is described in Section 1.4.

1.2.5 Example

This section illustrates basic characteristics of the denotational method of semantic definition by means of a very simple example. The denotational semantics will be given both in the direct and continuation styles.

Programs in the imperative baby language defined below consist simply of a sequence of assignment commands followed by an expression. Assignment commands assign values to variables, and expressions denote values. Variables are identifiers, the syntax of which will not concern us here. Expressions may be numerals, variables, or may be formed from simpler expressions by using a given set of predefined dyadic operators. The syntax of numerals and operators will not be of concern for us here either.

We want programs in this baby language to satisfy the following (context-sensitive constraint): variables occurring in expressions must have been assigned into (i.e. must have occurred previously in the text of the program in the left-hand side of an assignment command).

Syntax

The syntax of this baby language is defined as follows. We assume that Δ is a list of formulas and Num is a given set (of syntactic elements) such that $\Delta \vdash n \circ Num$ is syntactically derivable (possibly by using inference rules which do not concern us here) if and only if $n \in Num$. Analogous assumptions are valid for Id and Op. (We will assume for simplicity only one type of numerals in this example, say integer numerals).

We use the following corresponding syntactic meta-variables and phrase types:[20]

Meta-variable	Phrase type	
pg	Pg	programs
c	C	commands
e	E	expressions
v	Id	identifiers
v	V	variables
n	Num	numerals
op	Op	operators

We have:

$$\frac{\Delta \vdash c \circ C \quad \Delta \vdash e \circ E}{\Delta \vdash c;\, e \circ Pg}$$

$$\frac{\Delta \vdash v \circ Id \quad \Delta \vdash e \circ E}{\Delta \vdash v \leftarrow e \circ C[v]}$$

$$\frac{\Delta \vdash c_1 \circ C[v_1, \ldots, v_n] \quad \Delta, v_1 \circ V, \ldots, v_n \circ V \vdash c_2 \circ C[v'_1, \ldots, v'_m]}{\Delta \vdash c_1;\, c_2 \circ C[v_1, \ldots, v_n, v'_1, \ldots, v'_m]}$$

[20]Identifiers can occur in the left-hand side of assignment commands but only those which have occurred can be used in expressions, as variables. This is the motivation for the distinction between identifiers and variables.

$$\frac{\Delta \vdash v \circ V}{\Delta \vdash v \circ E}$$

$$\frac{\Delta \vdash n \circ Num}{\Delta \vdash n \circ E}$$

$$\frac{\Delta \vdash e_1 \circ E \quad \Delta \vdash op \circ Op \quad \Delta \vdash e_2 \circ E}{\Delta \vdash e_1 \; op \; e_2 \circ E}$$

Denotational Semantics: direct style

The semantics is given in relation to an interpretation $\mathbf{I}$, which is a pair $(D, \mathcal{I})$, where D is a *semantic domain*[21] of the interpretation $\mathbf{I}$, and $\mathcal{I}$ is an *interpretation function* for constant and function symbols, mapping them to elements and functions over D, respectively.

Here the constant and function symbols are simply the numerals and dyadic operators in Num and Op, respectively.

Let $\sigma = V_1 \to \mathbf{Z}$ be a finite mapping from a (finite) set of variables V_1 to the set of integer numbers, and Σ be the set of all such mappings.

Let $pg \equiv c;\ e$. The meaning $\mathbf{I}[\![pg]\!]$ of program pg is defined as an element of $\Sigma \times \mathbf{Z}$ as follows.

Let $\sigma' = \mathbf{I}[\![c]\!](\emptyset)$. We have:

$$\mathbf{I}[\![pg]\!] = \big(\sigma', \mathbf{I}[\![e]\!](\sigma')\big)$$

For any command c, its meaning is given by a function $\mathbf{I}[\![c]\!] : \Sigma \to \Sigma$ defined inductively as follows:

- if $c \equiv v \leftarrow e$ (for some expression e and variable v), then:

$$\mathbf{I}[\![c]\!](\sigma) = \sigma \dagger \{v \mapsto \mathbf{I}[\![e]\!](\sigma)\}$$

- if $c \equiv c_1; c_2$ (for some commands c_1, c_2), then:

$$\mathbf{I}[\![c]\!](\sigma) = \mathbf{I}[\![c_2]\!]\big(\mathbf{I}[\![c_1]\!](\sigma)\big)$$

For any expression e, its meaning is given by a function $\mathbf{I}[\![e]\!] : \Sigma \to \mathbf{Z}$ defined inductively as follows:

- if $e \equiv n$ (for some $n \in Num$), then:

$$\mathbf{I}[\![e]\!](\sigma) = \mathcal{I}(n)$$

- if $e \equiv v$ (for some variable v), then:

$$\mathbf{I}[\![e]\!](\sigma) = \sigma(v)$$

- if $e \equiv e_1 \; op \; e_2$ (for some expressions e_1, e_2 and some $op \in Op$), then:

$$\mathbf{I}[\![e]\!](\sigma) = \mathcal{I}(op)\big(\mathbf{I}[\![e_1]\!](\sigma), \mathbf{I}[\![e_2]\!](\sigma)\big)$$

[21] Also called *domain of discourse* or *carrier*.

Denotational Semantics: continuation style

We will illustrate firstly command continuations and then expression continuations. Command continuations represent the meaning of the rest of the program, and expression continuations the meaning of the rest of the evaluation of an expression.

We let Σ be as above and define the set of command continuations as:

$$CC = \Sigma \to \Sigma$$

For any command c its meaning is given by a function $\mathbf{I}[\![c]\!] : CC \to CC$ defined below.

First let $pg \equiv c;\ e$. The meaning $\mathbf{I}[\![pg]\!]$ of program pg is defined as an element of $\Sigma \times \mathbf{Z}$ as follows. Let $\sigma' = \mathbf{I}[\![c]\!](\lambda\sigma.\sigma)(\emptyset)$. We have:

$$\mathbf{I}[\![pg]\!] = \big(\sigma', \mathbf{I}[\![e]\!](\sigma')\big)$$

For any command c, the function $\mathbf{I}[\![c]\!] : CC \to CC$ is defined inductively as follows:

- if $c \equiv v \leftarrow e$, then:

$$\mathbf{I}[\![c]\!](cc)(\sigma) = cc\big(\sigma \dagger \{v \mapsto \mathbf{I}[\![e]\!](\sigma)\}\big)$$

- if $c \equiv c_1; c_2$, then:

$$\mathbf{I}[\![c]\!](cc) = \mathbf{I}[\![c_1]\!]\big(\mathbf{I}[\![c_2]\!](cc)\big)$$

For any expression e, its meaning is given by a function $\mathbf{I}[\![e]\!] : \Sigma \to \mathbf{Z}$ defined inductively as above.

We illustrate below an example of defining the evaluation order of expressions (in the example, left-to-right evaluation) by using expression continuations. With $Ec = \mathbf{Z} \to \Sigma \to \mathbf{Z}$, we redefine the following:

1. the meaning of a program pg, where $pg \equiv c;\ e$ and σ' is as above, is now given by:

$$\mathbf{I}[\![pg]\!] = \big(\sigma', \mathbf{I}[\![e]\!](\lambda z.\lambda\sigma.z)(\sigma')\big)$$

2. the meaning of a command $c \equiv v \leftarrow e$ is given by:

$$\mathbf{I}[\![c]\!](cc)(\sigma) = cc\big(\sigma \dagger \{v \mapsto \mathbf{I}[\![e]\!](\lambda z.\lambda\sigma.z)(\sigma)\}\big)$$

3. for any expression e, its meaning is given by a function $\mathbf{I}[\![e]\!] : Ec \to \Sigma \to \mathbf{Z}$ defined inductively as follows:

   - if $e \equiv n$, then:

$$\mathbf{I}[\![e]\!](ec) = ec\big(\mathcal{I}(n)\big)$$

   - if $e \equiv v$, then:

$$\mathbf{I}[\![e]\!](ec)(\sigma) = ec\big(\sigma(v)\big)(\sigma)$$

   - if $e \equiv e_1\, op\, e_2$, then:

$$\mathbf{I}[\![e]\!](ec) = \mathbf{I}[\![e_1]\!]\Big(\lambda z.\mathbf{I}[\![e_2]\!]\Big(\lambda z'.ec\big(\mathcal{I}(op)(z, z')\big)\Big)\Big)$$

1.2.6 Complete Partial Orders, Continuous Functions and the Fixed Point Theorem

In this section we present some definitions and results used for the characterization of a unique solution to recursive equations that occur naturally in the definition of the denotational semantics of some programming language constructs. In particular, these results are used to show the existence of a unique solution to equation (i), page 36.[22]

Partial Orders

A *partial order* is a pair $(X, \sqsubseteq)$, where X is a set and $\sqsubseteq$ is a relation on X that is reflexive, transitive and antisymmetric.[23]

l is the *least element* of a subset X' of X (with respect to the partial order $\sqsubseteq$) if $l \sqsubseteq x$, for all $x \in X'$.[24]

u is an *upper bound* of a subset X' of X (with respect to the partial order $\sqsubseteq$) if $x \sqsubseteq u$, for all $x \in X'$.

The *least upper bound* of a subset X' of X is (naturally) the least element (with respect to a given partial order) of the set of all upper bounds of X' (with respect to the same partial order). Such least upper bound, when it exists, is denoted by $\bigsqcup X'$ (where the partial order is supposed to be understood from the context where $\bigsqcup$ is used).[25]

A *chain* of a partial order $(X, \sqsubseteq)$ is a non-empty subset X' of X such that $x_1 \sqsubseteq x_2$ or $x_2 \sqsubseteq x_1$ for all $x_1, x_2 \in X'$.[26]

Complete Partial Orders

A partial order $(X, \sqsubseteq)$ is a *complete partial order (cpo)* if every chain of $(X, \sqsubseteq)$ has a least upper bound.

A cpo $(X, \sqsubseteq)$ is a *cpo-with-bottom* if $(X, \sqsubseteq)$ has a least element.

Any set together with the identity relation on it is a cpo. Any such cpo is called a *flat cpo*.

Let X be a set, $X_\perp = X \cup \{\perp\}$ and $\sqsubseteq$ be a relation on $X_\perp$ given by:

$$x \sqsubseteq x' \text{ if and only if } x = \perp \text{ or } x = x'$$

Then $(X_\perp, \sqsubseteq)$ is a cpo-with-bottom, where the least element is $\perp$, called a *flat cpo-with-bottom*.

Proposition 1 Let X be a set and $(Y, \sqsubseteq_1)$ be a cpo. Let $\sqsubseteq$ be the relation on $X \to Y$ given by:

$$f \sqsubseteq g \text{ if } f(x) \sqsubseteq_1 g(x), \text{ for all } x \in X$$

Then $(X \to Y, \sqsubseteq)$ is a cpo.[27]

Proposition 2 Let $(X_i, \sqsubseteq_i)$ be cpos, for $i = 1, \ldots, n$, $n \geq 1$. Let $\sqsubseteq$ be the relation on $X_1 \times \ldots \times X_n$ given by:

$$(x_1, \ldots, x_n) \sqsubseteq (x'_1, \ldots, x'_n) \text{ if } x_i \sqsubseteq_i x'_i, \text{ for all } i = 1, \ldots, n$$

Then $(X_1 \times \ldots \times X_n, \sqsubseteq)$ is a cpo.[28]

In all cases, the relation $\sqsubseteq$ of a partial order $(X \to Y, \sqsubseteq)$ or of a partial order $(X_1, \ldots, X_n, \sqsubseteq)$ is to be understood as the relation defined in Proposition 1 or 2 above, respectively.

Proposition 3 Let $(X_i, \sqsubseteq_i)$, $i = 1, \ldots, n$, $n \geq 1$, be partial orders which have only finite chains. Then $(X_1 \times \ldots \times X_n, \sqsubseteq)$ has only finite chains.

[22] The results are presented in this section in the form of propositions, for which proofs can be found or are analogous to those in e.g. [dB80, LS87, Win93]

[23] $\sqsubseteq$ is reflexive if, for all $x \in X$, $x \sqsubseteq x$; it is transitive if, for all $x_1, x_2, x_3 \in X$, $x_1 \sqsubseteq x_3$ if $x_1 \sqsubseteq x_2$ and $x_2 \sqsubseteq x_3$; antisymmetric if, for all $x_1, x_2 \in X$, $x_1 = x_2$ if $x_1 \sqsubseteq x_2$ and $x_2 \sqsubseteq x_1$.

[24] In this case, if $X' = X$, l is also called the least element of the partial order.

[25] The definitions of greatest element, lower bound and greatest lower bound are analogous. They will not be used.

[26] In other words, X' is a chain if $\sqsubseteq$ restricted to X' is total.

[27] Analogous propositions are valid for partial orders, cpos-with-bottom and flat cpos (but not for flat cpos-with-bottom).

[28] Analogous lemmas are valid for partial orders, cpos-with-bottom and flat cpos (but not for flat cpos-with-bottom).

Monotonic and Continuous Functions

Let $(X, \sqsubseteq_1)$ and $(Y, \sqsubseteq_2)$ be cpos. A function $f : X \rightarrow Y$ is *monotonic* (with respect to the partial orders $\sqsubseteq_1$ and $\sqsubseteq_2$) if, for all $x, x' \in X$, $f(x) \sqsubseteq_2 f(x')$ if $x \sqsubseteq_1 x'$.

Such a function is *continuous* (with respect to the partial orders $\sqsubseteq_1$ and $\sqsubseteq_2$) if it is monotonic (with respect to the same partial orders) and for all chains of $(X, \sqsubseteq_1)$, we have:[29]

$$\bigsqcup\{f(x) \mid x \in X\} = f(\bigsqcup X)$$

Proposition 4 Let $(X, \sqsubseteq_1)$ and $(Y, \sqsubseteq_2)$ be cpos. A function $f : X \rightarrow Y$ is continuous if and only if, for all chains X' of $(X, \sqsubseteq_1)$, it holds that $\bigsqcup\{f(x) \mid x \in X'\}$ exists and $\bigsqcup\{f(x) \mid x \in X'\} = f(\bigsqcup X')$.

The Fixed Point Theorem

A fixed point of a function $f : X \rightarrow X$ is an element $x \in X$ such that $f(x) = x$.

The least fixed point of f is (naturally) the least element of the set of all fixed points of f.

Theorem 1 (Fixed Point Theorem) Let $(X, \sqsubseteq)$ be a cpo and $f : X \rightarrow X$ be continuous. Then f has a least fixed point $fix(f)$ given by:

$$fix(f) = \bigsqcup\{f^i(\bot) \mid i \in \mathbf{N}\}$$

Showing that Functions are Continuous

The following propositions can be used to show that functions are continuous.

Proposition 5 Let $(X, \sqsubseteq)$ be a flat cpo. Then any function $f : X \rightarrow Y$ is continuous.

Proposition 6 Let $(X, \sqsubseteq)$ be a cpo of which all chains are finite. Then any monotonic function $f : X \rightarrow Y$ is continuous.[30]

Proposition 7 The following functions, assuming that their respective domains and ranges constitute cpos, are continuous:

1. Constant functions;
2. Identity functions;
3. Projection functions.

Proposition 8 Let $(X, \sqsubseteq)$, $(Y_i, \sqsubseteq_i)$ for $1 = 1, \ldots, n$ be cpos, $n \geq 1$. Then $f : X \rightarrow Y_1 \times \ldots \times Y_n$ is continuous if and only if $pr_i \circ f$ is continuous for all $i = 1, \ldots, n$.

[29]In fact it need only be required that:

$$f(\bigsqcup X) \sqsubseteq_2 \bigsqcup\{f(x) \mid x \in X\}$$

since the other inequality, namely:

$$\bigsqcup\{f(x) \mid x \in X\} \sqsubseteq_2 f(\bigsqcup X)$$

comes as a consequence of the monotonicity of f.

[30]A particular case is when $(X, \sqsubseteq)$ is a flat cpo-with-bottom, which has chains containing only one or two elements. A cartesian product of flat cpos-with-bottom is another example (cf. Proposition 3).

Let $(X_i, \sqsubseteq_i)$, for $i = 1, \ldots, n$, and $(Y, \sqsubseteq)$ be cpos, $n \geq 1$. For $i \in \{1, \ldots, n\}$, the function $f : X_1 \times \ldots \times X_n \to Y$ is *continuous in the i-th argument* if for all $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$, the function $f[\![x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n]\!] : X_i \to Y$ given by:

$$f[\![x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n]\!](x_i) = f(x_1, \ldots, x_n)$$

is continuous.

An analogous definition applies with respect to partial orders and monotonicity (instead of cpos and continuity, respectively).

Proposition 9 Let $(X_i, \sqsubseteq_i)$, for $i = 1, \ldots, n$, and $(Y, \sqsubseteq)$ be cpos, $n \geq 1$. Then $f : X_1 \times \ldots \times X_n \to Y$ is continuous if and only if f is continuous in the i-th argument for all $i = 1, \ldots, n$.

Proposition 10 Let $(X_i, \sqsubseteq_i)$, for $i = 1, \ldots, n$, and $(Y, \sqsubseteq)$ be partial orders, $n \geq 1$. Then $f : X_1 \times \ldots \times X_n \to Y$ is monotonic if and only if f is monotonic in the i-th argument for all $i = 1, \ldots, n$.

Proposition 11 For $i = 1, \ldots, n$, $n \geq 1$, let $(X_i, \sqsubseteq_i)$ be flat cpos-with-bottom, with least elements $\perp_i$. Let $(Y, \sqsubseteq)$ be also a flat cpo-with-bottom, with least element $\perp$. Then $f : X_1 \times \ldots \times X_n \to Y$ is monotonic if and only if for all i and all $(n-1)$-tuples $(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) \in X_1 \times \ldots X_{i-1} \times X_{i+1} \times \ldots X_n$ it holds that either:

$$f(x_1, \ldots, x_{i-1}, \perp_i, x_{i+1}, \ldots, x_n) = \perp$$

or, for all $x_i \in X_i$,

$$f(x_1, \ldots, x_{i-1}, \perp_i, x_{i+1}, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n)$$

Proposition 12 Let $(X, \sqsubseteq_1)$, $(Y, \sqsubseteq_2)$ and $(Z, \sqsubseteq_3)$ be cpos and $g : X \to Y$ and $f : Y \to Z$ be continuous. Then $f \circ g$ is also continuous.

Proposition 13 Let $(X, \sqsubseteq_1)$, $(X_i, \sqsubseteq_i)$ for $i = 1, \ldots, n$ and $(Y, \sqsubseteq)$ be cpos, $n \geq 1$. Let $f : X_1 \times \ldots \times X_n \to Y$ and $f_i : X \to X_i$ be continuous, for all $i = 1, \ldots, n$. Then $g : X \to Y$ given by:

$$g(x) = f\big(f_1(x), \ldots, f_n(x)\big)$$

is continuous.[31]

Let $f : X_1 \times \ldots \times X_n \to Y$, $n \geq 2$. We call the *curried function* of f the function $curry[\![f]\!] : X_1 \times \ldots X_{n-1} \to X_n \to Y$ given by

$$curry[\![f]\!](x_1, \ldots, x_{n-1})(x_n) = f(x_1, \ldots, x_n)$$

Proposition 14 Let $(X_i, \sqsubseteq_i)$ be cpos, $i = 1, \ldots, n$, and $f : X_1 \times \ldots \times X_n \to Y$. Then f is continuous if and only if $curry[\![f]\!]$ is continuous.

Examples

The following examples use the above propositions to show that functions are continuous.

Apply

Let $(X, \sqsubseteq_1)$ and $(Y, \sqsubseteq_2)$ be cpos. Then the function $apply : (X \to Y) \times X \to Y$ given by:

$$apply(f, x) = f(x)$$

is continuous.

By Proposition 1, $X \to Y$ is a cpo. Consider then the function $curry[\![apply]\!] : (X \to Y) \to X \to Y$. It is the identity function on $X \to Y$. By Proposition 7, it is continuous. Thus, by Proposition 14, *apply* is continuous.

[31] Proposition 13 is a generalization of Proposition 12.

If-then-else

Let $(\mathbf{B}_\perp, \sqsubseteq_b)$ be a flat cpo-with-bottom, where $\mathbf{B}_\perp = \mathbf{B} \cup \{\perp\}$, $\perp$ the least element of $(\mathbf{B}_\perp, \sqsubseteq_b)$. Let $(\mathbf{N}_{\perp_0}, \sqsubseteq)$ be a flat cpo-with-bottom, where $\mathbf{N}_{\perp_0} = \mathbf{N} \cup \{\perp_0\}$, $\perp_0$ the least element of $(\mathbf{N}_{\perp_0}, \sqsubseteq)$.

The function $\texttt{if-then-else} : \mathbf{B}_\perp \times \mathbf{N}_{\perp_0} \times \mathbf{N}_{\perp_0} \to \mathbf{N}_{\perp_0}$ given by

$$\texttt{if-then-else}(b, n, n') = \begin{cases} \perp_0 & \text{if } b = \perp \\ n & \text{if } b = \mathit{true} \\ n' & \text{if } b = \mathit{false} \end{cases}$$

is continuous.

By Proposition 11, `if-then-else` is monotonic. Its domain is a cpo, by Proposition 2, and contains only finite chains, by Proposition 3.

Therefore, by Proposition 6, it is continuous.

Factorial Functional

Let $\Phi : (\mathbf{N} \to \mathbf{N}_{\perp_0}) \to (\mathbf{N} \to \mathbf{N}_{\perp_0})$ be given by:

$$\Phi(f)(n) = \texttt{if-then-else}\big(n = 0, 1, n \,\underline{*}\, f(n-1)\big)$$

where the function `if-then-else` is as given in the example above and $\underline{*}$ is an operator given by the definition below, whose corresponding function, denoted by $g_{\underline{*}}$, is a strict extension of the function corresponding to the arithmetic multiplication operator $*$:

$$n_1 \,\underline{*}\, n_2 = \begin{cases} \perp_0 & \text{if } n_1 = \perp_0 \text{ or } n_2 = \perp_0 \\ n_1 * n_2 & \text{otherwise} \end{cases}$$

We show below that the functional Φ is continuous.

Consider $\Phi' : (\mathbf{N} \to \mathbf{N}_{\perp_0}) \times \mathbf{N} \to \mathbf{N}_{\perp_0}$ given by $\Phi'(f, n) = \Phi(f)(n)$. Its domain is a cpo, by Proposition 1. Then, by Proposition 9, Φ' is continuous, since:

1. $\Phi'[\![f]\!] : \mathbf{N} \to \mathbf{N}_{\perp_0}$ given by $\Phi'[\![f]\!](n) = \Phi'(f)(n)$ is continuous, by Proposition 5.
2. $\Phi'[\![n]\!] : (\mathbf{N} \to \mathbf{N}_{\perp_0}) \to \mathbf{N}_{\perp_0}$ given by $\Phi'[\![n]\!](f) = \Phi'(f)(n)$ is continuous, by Proposition 13, from the fact that it can be written as:

   $$\Phi'[\![n]\!](f) = \texttt{if-then-else}\big(g_1[\![n]\!](f), g_2[\![n]\!](f), g_3[\![n]\!](f)\big)$$

   where:

   - `if-then-else` is continuous (as shown in the example above).
   - $g_1[\![n]\!] : (\mathbf{N} \to \mathbf{N}_{\perp_0}) \to \mathbf{B}_\perp$ given by:

     $$g_1[\![n]\!](f) = (n = 0)$$

     is continuous, by Proposition 7 (it is a constant function).
   - similarly, $g_2[\![n]\!] : (\mathbf{N} \to \mathbf{N}_{\perp_0}) \to \mathbf{N}_{\perp_0}$ given by:

     $$g_2[\![n]\!](f) = 1$$

     is also a constant function, thus continuous, by Proposition 7.
   - $g_3[\![n]\!] : (\mathbf{N} \to \mathbf{N}_{\perp_0}) \to \mathbf{B}_\perp$ is continuous, by Proposition 13, since it can be written as:

     $$g_3[\![n]\!](f) = g_{\underline{*}}\big(g_4(f), g_5(f)\big)$$

     where:

– $g_{\underline{*}} : \mathbf{N}_{\perp_0} \times \mathbf{N}_{\perp_0} \to \mathbf{N}_{\perp_0}$ is continuous, by Proposition 6, since it is monotonic, by Proposition 11, and its domain is a cpo, by Proposition 2, and contains only finite chains, by Proposition 3.
– $g_4[\![n]\!] : (\mathbf{N} \to \mathbf{N}_{\perp_0}) \to \mathbf{N}_{\perp_0}$ given by:

$$g_4[\![n]\!](f) = n$$

is continuous, by Proposition 7 (it is a constant function).
– $g_5[\![n]\!] : (\mathbf{N} \to \mathbf{N}_{\perp_0}) \to \mathbf{N}_{\perp_0}$ given by:

$$g_5[\![n]\!](f) = f(n-1)$$

is continuous, by Proposition 13, since it can be written as $g_5[\![n]\!](f) = \mathit{apply}(f, n-1)$, where *apply* is continuous (as shown in the example above), and identity and constant functions are continuous, by Proposition 7.

Therefore, Φ is continuous, by Proposition 14.

1.3 Object-Based Languages

This section presents an introduction to object-based programming languages, to serve as a background for the understanding of the proof system for **A1**.

The introduction in this section is based on the view that object-based languages provide a good framework for the study of the problem of verification of correctness of programs that dynamically change data structures (see Section 1.4). It is not a general introduction to the object-oriented approach to problem solving or program design, the main concern in texts about object-oriented programming. This is also the major reason why we do not include in the language an inheritance and/or a subtyping mechanism, which would make our treatment of types more complicated.[32]

We describe in this section the informal concept of objects and the basic characteristics of object-based languages, including those of mechanisms to support concurrency.

1.3.1 Objects

The term *object-based* refers to a style of programming based on the concept of an *object*. This concept is closely related to the concept of data abstraction used in the programming style based on abstract data type programming languages[33]. In this programming style, the data and the operations on these data are viewed as a single abstraction: the data specify its internal state, which can be manipulated only by given operations, which constitute its external interface.

Attempts to identify the fundamental properties of the (informal) object concept have included the following:

- An object is a dynamic entity, i.e. it exists during a program's execution, and during an execution objects can be created in arbitrary numbers.
- An object can have internal variables, or attributes, which determine properties of the object and which can only be changed by procedures that have been chosen to have that ability (e.g. by occurring in the text of an object definition).
- An object has a set of procedures through which other objects can interact with it (this set of procedures constitutes what is called the object's external interface).

[32]For a description and discussion of (static) type checking in an object-oriented language see e.g. [Mey92, Chapter 22]. See e.g. [Coo90, AvdL90] for a discussion on the need to distinguish between inheritance and subtyping.
[33]See e.g [LZ74, LG86, Lis87].

- An object has an identity; this identity provides the ability of sharing, such that changes to the object are automatically reflected in all objects that can interact with it.

The advantages of having linguistic constructs to support the use of objects have been widely discussed and concern simpler interfaces of routines, locality, less data transmission, and greater software reusability and extendibility.[34]

1.3.2 Object Definitions

Data abstractions are supported by a linguistic construct in several languages.[35] Following Simula, this construct has been called *class* in many object-oriented languages[36]; in the definition of **A1** (Section 2) we use the term *object definition*. This construct is used for composition and decomposition of software systems, with information hiding and encapsulation. The interface of the data abstraction corresponds to procedures, or *methods*, of an object definition; each method provides the implementation of a given operation of the corresponding data abstraction. The internal state of the data abstraction corresponds to the object's variables.[37]

In languages based on object definitions, objects are called *instances* of such object definitions. This is based on the facts that the creation of an object (by an object creation commands) specify an object definition, and that the execution of any command which occurs in the text of an object definition is associated with a created object. Instead of saying "the execution of a command associated with an object a" it is common to say "the execution of a command by object a" (in this case it is also usual to take a more anthropomorphic view and refer to "the object executing a command"). When the command or set of commands are understood from the context as those occurring in a given object definition, it is usual to refer to "the current object".

The name *method* has been used instead of *procedure* in object-oriented languages to emphasize the *dynamic binding* aspect of a method call. Dynamic binding is required in a language which permits redefinition of procedure definitions and adopts a typing policy which allows variables to contain values of a certain type together with values of its subtypes. For better understanding, consider as an example a call $x.p$, where x is a variable and p is a procedure (disregard for now the occurrence of parameters and arguments). Let X be the (static) type of x; if p is (re)defined in an object definition Y, and x can (dynamically) contain a value of type Y, then the call $x.p$ must be dynamically bound to this redefinition. In the case of a call $x.p$, it is common in object-oriented terminology to say also that the *message* p has been sent —by the *sender* object— to the object denoted by x, called the *receiver* object.

It is not difficult to see that a similar argument may in general be used in the case of method parameters and arguments of a method call. However, mainly for practical reasons, dynamic binding on arguments of method calls is not usually supported in object-oriented languages.

In the definition of **A1**, we use the name *methods* mainly because it is a short term for *external procedures*, in contraposition to *internal procedures*. However, for simplicity but mainly because this is a matter of language design strictly related to programming methodology, we do not include in **A1** procedures which can be used only internally .

An important matter of language design related to issues of programming methodology is the question of access to object variables. In many languages, such as for example Smalltalk[GR83] or Eiffel[Mey92], in a given object definition, say X, there can only be access to variables of a single instance of X (namely the variables of the current object), whereas in other languages, such as CLU[Lis81] or C++[Str87] for example, in X there can be access to variables of all instances of X.[38] For reasons of simplicity, in **A1** we use the first policy.

---

[34] See. e.g. [Par71, Par72, LG86, Mey88].

[35] The earliest such language was Simula 67[DN66, DH72].

[36] Cf. e.g. [BDMN73, GR83, Mey88, Str87].

[37] These are called, for example, *instance variables* in Smalltalk[GR83] and *attributes* in Eiffel[Mey92]. The more general terminology in Eiffel is used to distinguish between variable and constant attributes. For simplicity, in **A1** and henceforth they are called *object variables*.

[38] This issue has been seen as a way of distinguishing object-oriented languages from languages based on abstract data types (see [Ame89, Section 2.2]).

It is appropriate to make a brief comment on the use of the term "object-based" in contrast to "object-oriented". Initially, the object-oriented style of programming was mainly characterized, informally speaking, as an approach that aimed at identifying active entities of a system (the objects) and specifying how they interact. An anthropomorphic view of objects, that respond to messages addressed to them, was placed at the heart of the object-oriented metaphor, which stressed the importance of the communication mechanism between objects.[39] Several works, which considered a less operational view of object-oriented programming then included objects, classes and inheritance as their basic elements.[40] For programming languages based on classes, the distinction between the terms object-oriented and object-based is now commonly used and understood as meaning, respectively, languages with and without an *inheritance* mechanism. But it is clear that this distinction is not adequate *in general*, as shown mainly by works with languages and systems based on prototypes and delegation[41], instead of classes and inheritance.

1.3.3 Concurrent Object-Based Languages

This section gives an introduction to concurrent object-based languages with the aim of providing a background to Section 7.3 on a development method based on transformation of sequential programs, applicable to a certain class of concurrent object-based programs. It is not our intention to provide a general classification of object-oriented languages[42], but just to introduce relevant issues and concepts.

Further possible extensions of the work in this thesis to handle concurrency constitute another major reason for our stand on working with object-based languages, in contraposition to object-oriented languages. The interference of concurrency features with object-oriented issues, such as the distinction between inheritance and subtyping, makes it more dificult to achieve a satisfactory integration.[43]

There are two basic approaches for providing concurrent execution in object-based languages:[44]

1. Introduce a separate process construct and some mechanism for process communication and synchronization independently of objects;[45]

2. consider objects as potential units of concurrent execution.

The second approach is more appealing with respect to issues like simplicity and modularity, and includes the vast majority of concurrent object-oriented languages. There are still many fundamental differences between them.

Our next consideration is whether:

1. all objects are potential units of concurrent execution;

2. a particular subset of objects are potential units of concurrent execution.

Members of the particular set of objects above are often called *active*, in contraposition to members of its complement set, which are called *passive* objects. As an example of the second approach, we can consider languages which use the inheritance mechanism to characterize such particular subset of objects as those which are instances of subclasses of a particular class or of a particular set of classes.[46]

The first question that arises in the context of objects that are potential units of concurrent execution is whether and in which cases they can have multiple threads of execution active at a time.

[39] See e.g. [Cox86].
[40] Cf. e.g. [Sny86, Weg87, Str88][Mey88, page 59].
[41] See e.g. [Bor86, US87].
[42] Cf. e.g. [Pap92].
[43] For discussions on these issues see e.g. [KL89, TS89].
[44] See also [Pap92, Chapter 3].
[45] For discussions on a variety of concurrent programming constructs see e.g. [AS83, And91]. This approach is adopted in object-oriented languages such as, for example, Smalltalk[GR83], Trellis/Owl[MK87] and Emerald[BHJL86]. The process synchronization constructs are semaphores in Smalltalk, "lock blocks" in Trellis/Owl and monitors in Emerald.
[46] Cf. e.g. Act++[KL89] and a concurrent version of Eiffel[Car90].

A first simple approach is to allow only one active thread of execution for *mutable* objects (i.e. objects which can have their internal variables changed), and an unlimited number of active threads for immutable objects. Examples of languages that use this approach are ABCL/1[YBS86] and POOL2[Ame89]. A less restrictive approach considers this as a variation of the classic *readers/writers* problem[47]. In this case, if a writer method (i.e. a method that can change the internal variables of the receiver object) is being executed then there must exist only one active thread of execution, but otherwise any number of reader methods may concurrently be in execution. Another approach is to support a mechanism for explicitly releasing control in order to allow only one thread of execution, but more than one thread associated with a given object, at a time.[48]

There are two fundamental questions to be considered next:

1. how concurrent execution is started in the first place, and

2. how synchronization between objects is supported.

There are several possible answers to the first question, concerning which concurrent execution can be started when:

1. An object is created.[49] This means that after the execution of a command by object a that creates a', the executions of commands associated with a and with a' can both proceed concurrently. In this case, an object has (usually optionally) a *body*[50], and the execution of the body starts when the object is created. There must be also a way of integrating the execution of the body with the execution of the methods of an object.[51]

2. A message is sent asynchronously.[52] This means that after the execution of a method call by object a, for which the receiver object is a', the executions of commands associated with a and with a' can both proceed concurrently.

3. The result of a method call is returned before the end of that call.[53] Letting objects a and a' be respectively the sender and receiver of a message, then this means that after the execution of a command that specifies the result of the message (and before the end of the execution of the corresponding method) the executions of commands associated with a and with a' can both proceed concurrently.

The question of synchronization between objects, in such a context of objects as potential units of concurrent execution, becomes usually a more basic question of how to specify which methods (among the set of all methods in an object definition) are possible candidates for activation. That is, supposing $m_1, \ldots, m_n$ are the names of all methods in an object definition X, then synchronization between objects is usually done by specifying, for any given object, which of $m_1, \ldots, m_n$ can (or, equivalently, cannot) be activated upon the sending of a message by another object. (It is clear that the sender and receiver objects can possibly in fact be the same.)

This can be done in several ways; the most usual ones are as follows:

1. Use a specific command for that purpose.[54]

[47] See e.g. [And91, Section 4.3].

[48] Cf. e.g. delegated calls in Hybrid[Pap92, Chapter 2].

[49] Examples of languages which adopt such an approach are Simula[DN66] and POOL2[Ame89].

[50] Note that if bodies are optional, this creates a distinction between active (i.e. those objects which have a body) and passive objects (i.e. those which do not have a body).

[51] Cf. e.g. the *answer* statement in POOL2[Ame89, Section 3.2].

[52] This is adopted in many object-oriented languages, such as for example *actor* languages[Hew77, Agh86, Lie87], ABCL/1[YBS86], ConcurrentSmalltalk[YT86] and suggested adaptations of Eiffel to cover concurrent programming[Mey90, Car90, Mey93].

[53] This has been adopted in languages such as for example ConcurrentSmalltalk[YT86], Act++[KL89] and $\pi o \beta \lambda$[Jon93b].

[54] Like, for example, with the *answer* statement in POOL2[Ame89].

2. Use *activation conditions*. These may be predicates associated with methods[55] that may depend on the values denoted by the object variables, on arguments of methods and on other conditions like the number of active threads of another particular method or set of methods, or may be derived from a specification of legal sequences of activation as in path expressions[CH74].

In their simplest form, activation conditions are like the mechanism used for controlling whether and in which cases there may be multiple threads of execution associated with a given object. In this case, it becomes apparent that the two methods above can also be used in conjunction with each other.

1.4 Proof Systems

This section presents an introduction to proof systems[56] for the verification of correctness of programs written in imperative sequential programming languages and describes possibilities and difficulties in extending these proof systems to cope with object-based programming languages.

1.4.1 Proofs

Let W be some set of syntactic elements.[57] A *proof system* (or *calculus*) K is a pair (A, R) where A is a finite set of axiom schemes and R is a finite set of proof rules. An *axiom scheme* is a given recursive subset of W.[58] The elements of an axiom scheme are called *axioms*. A *proof rule* (or *inference rule*) is a recursive subset of $W^n \times W, n \geq 1$.[59]

A proof rule:

$$r = \{\big((w_1, \ldots, w_n), w\big) \mid \big((w_1, \ldots, w_n), w\big) \in W^n \times W \text{ and } P \text{ holds for } w, w_1, \ldots, w_n\}$$

is typically written as:

$$\frac{w_1 \quad \ldots \quad w_n}{w}$$
$$\text{where } P \text{ holds for } w, w_1, \ldots, w_n$$

$w_1, \ldots, w_n$ are called the *premises* of the proof rule and w its *conclusion*.

Now let X be any subset of W. The set of all elements of W which are *provable* (or *derivable*) from X in proof system K is inductively defined by:[60]

1. the basis set $B = X \cup A$, and
2. the constructor set R.

The fact that an element w is provable from a set X in a proof system K is denoted by $X \vdash_K w$ or, when the proof system is understood from the context, simply $X \vdash w$; if X is the empty set, then $\vdash w$ is used.[61]

[55] As in e.g. Guide[KMV+90] and Act++[KL89].

[56] Proof systems are also called inference systems, formal systems, deductive systems, production systems, axiomatic systems, formal deductive systems etc.

[57] Usually, this is a set of well-formed phrases called *formulas*, as given by a language syntax definition.

[58] Informally, a set is said to be *recursive* if there is a program which decides (by giving an appropriate answer) whether any given element is a member of the set or not. A set is said to be *recursively enumerable* if there is a program which decides (by giving an appropriate answer) if any given element is a member of the set, but may fail to terminate for elements which are not members of the set. For a formal definition see e.g. [EC89].

[59] A proof rule in a system of natural deduction is defined below; the definition is different but gives the same effective character for proof rules as they have in a "classical" proof system.

[60] For inductive definitions see e.g. [End72][Gal86, Section 2.3][LS87, Section 1.2].

[61] The motivations for this separation between X and A are that X may contain axioms for extensions of the basic proof system and that X may be any given set, while A must be a recursive set.

A construction sequence for w (from B and R)[62], is called a (formal) *proof* (or *deduction*) of w from X in K. w is called a *theorem*[63].

Proofs in Systems of Natural Deduction

In a proof system of natural deduction, a proof rule is a recursive subset of $(\wp(W) \times W)^n \times (\wp(W) \times W)$, $n \geq 1$. For brevity, we denote below this set by Z.

A proof rule

$$r = \Big\{ \Big(((X_1, w_1), \ldots, (X_n, w_n)), (X, w) \Big) \mid \Big(((X_1, w_1), \ldots, (X_n, w_n)), (X, w) \Big) \in Z, \\ P \text{ holds for } X, X_1, \ldots, X_n, w, w_1, \ldots, w_n \Big\}$$

where we must have $X \subseteq X_1 \cup \ldots \cup X_n$, is typically defined by:

$$\frac{\begin{matrix} [\![w_1']\!] & & [\![w_n']\!] \\ w_1 & \ldots & w_n \end{matrix}}{w}$$
$$\text{where } P \text{ holds for } w, w_1, w_1', \ldots, w_n, w_n'$$

In this notation, it is understood that $X = (X_1 - \{w_1'\}) \cup \ldots \cup (X_n - \{w_n'\})$.

Each of $w_1', \ldots, w_n'$ is called a *discharged hypothesis*.

For any i between 1 and n, $\{w_i'\}$ can in fact be the empty set, and in this case the rule is written without the discharged hypothesis (i.e. without $[\![w_i']\!]$).

The set of all elements provable from X in a proof system K of natural deduction is given by w in the set of all pairs $(X \cup A, w)$ inductively defined by:

1. the basis set $B = (X \cup A, X \cup A)$, and
2. the constructor set R.

1.4.2 Program Correctness

Formal semantic definitions serve mainly as a basis for understanding and reasoning about how programs behave. Proof systems are aimed at establishing that programs execute "as intended", or "as required", or "as specified".

A common approach to this problem is testing: the program is executed on some number of test cases. This approach is very important to increase confidence but in general it cannot demonstrate the absence of errors. It is not difficult to see that in general no way of deciding which or how many test cases to use is going to remedy the situation.

Another approach, adopted in this thesis, uses:[64]

1. A formalization of "as specified". This is done by using a formally defined *specification language*[65].

[62] The construction sequence is as given by the inductive definition of all elements of W provable from X: i.e. a sequence $w_1, \ldots, w_n$ ($n \geq 1$) of elements of W such that, for every $i = 1, \ldots, n$, either:

1. $w_i \in B$, or
2. $((w_{i_1}, \ldots, w_{i_m}), w_i) \in R$, for some $i_1, \ldots, i_m < i$.

[63] The words "theorem" and "proof" have here, of course, a different meaning from their more common usage in mathematics. There a theorem means some statement (in ordinary language) which has been given a proof. Proof is a central concern in mathematics. At the present time what is a proof is not precisely defined; it is a result of imitation, i.e. of generations passing a culture of methods on to the next, and correction (making this culture change). For the fundamental methods of proof in mathematics, more details and interesting expositions on the subject see e.g. [Goo51, Pol65, Kol76, EC89].

[64] This approach has received a great deal of attention mainly after the work of Hoare[Hoa69], which presented proof rules in a syntax directed manner. The origins of the approach can be traced back to Turing[Tur49] and the main influence on Hoare's work is attributed to Floyd[Flo67]. For a tracing of significant steps in the history of research on reasoning about programs see [Jon92b].

[65] Also called assertion language, language of formal logic, logical language, rule meta-language etc.

2. A formalization of the statement "programs execute as specified" or, given the formalization above, a formalization of the statement "programs satisfy a specification".

3. A formal definition of the semantics of the programming language.

4. A proof system which is used to establish correspondences between programs and specifications.

In general, a very significant difficulty with this approach is well-known and it is the same for any proof system (with powerful enough capabilities, like those of proof systems for predicate logic): complexity and size of the proofs. The reason behind this difficulty comes from the very nature of the problem: it is the requirement of formal correctness that makes it necessary for all conditions and all steps of the deduction to be explicitly indicated. In particular, "implicit pruning" of intuitively clear cases and adaptations of similar cases are not parts of such formal proofs.

Tackling this problem constitutes a major and very difficult task. However, possible approaches may rest on:[66]

1. Using proven theorems to derive new ones and possibly new rules of inference, making proofs shorter. This relies on a *principle of abstraction*, which enables singling out *what* is proven and not having to consider *how* to prove. This is essential in the context of many sequential imperative programming languages in general, and in particular for object-based languages, specially in relation to procedure call (message sending) commands.

2. Viewing formal systems for *a posteriori* proofs of program correctness only as a basis on which to design systematic[67] or strictly formal program development methods.[68] The essence of such methods is a *principle of compositionality*, which enables the specification and proofs related to any (composite) syntactic element to depend, respectively, only on the specifications and proofs related to their syntactic components.

3. Computer verification systems. These programs can be used for proving invariants, constructing proofs interactively (i.e. with the assistance of the user) and checking proofs.[69]

Another general difficulty with this approach stems from the fact that proof systems for programming languages are themselves usually fairly complex. Their proof rules contain many conditions or restrictions and either examples of their use are not given or they are, even in small examples, difficult to follow. Many programming languages have constructs or combinations of constructs which make it difficult for simple proof rules to be defined.

There are also difficulties with this approach which are not related to aspects of programming and programming languages, such as the fact that every proof system in which recursive functions are representable is not axiomatizable. Since we want to concentrate on the proof rules relative to the programming language constructs, the tactic assumed here follows a common one[70] in which proofs are done from the set of all axioms valid in a given interpretation (a set that is not even recursively enumerable).

There must be merits to compensate for the difficulties. These are related to gains of knowledge, experience and confidence and are characterized by more precision in general, conciseness of specifications and the prospect of a progressive construction, even if accompanied by errors, of a firmer foundation for works on programming and programming languages.

---

[66] It should be noted that this thesis is focused on the subject of *a posteriori* proofs of correctness of object-based programs. Ways suggested below of tackling the problem of complexity and size of proofs are not neglected but a detailed study is outside the scope of the thesis.

[67] I.e. rigorous but not necessarily strictly formal.

[68] See e.g. [Dij76, Bat79, Bac81, Bac88, Jon90, NHWG89, MV92][Gri81, Part III].

[69] See e.g. [CJE83, NHWG89].

[70] Cf. e.g. [dB80, Fra92].

1.4.3 Object-Based Languages: extending Hoare-style proof systems

Specifications in the formally defined specification language (see Chapter 3)[71] are pairs $(p, (q, t))$, where p, q are formulas and t a term of **L1**. The intended interpretation of a given specification is such that for a command (of the programming language) to satisfy this specification its execution must be such that if it begins in a *state*[72] for which the interpretation of p is *true*, then it terminates in a state for which the interpretation of q is *true* and it returns a value which is given by the interpretation of t. A specification can also be a pair (p, q) (where p, q are formulas) for which the intended interpretation simply does not include the last condition relative to t.

A *triple* $(p, c, (q, t))$, written in the form $\{\,p\,\}c\{\,q, t\,\}$, or a triple (p, c, q), written in the form $\{\,p\,\}c\{\,q\,\}$, where c is a command of the programming language, is called a *correctness formula* (see Section 3.1). For any such pair (specification) or triple (correctness formula), p is called the *precondition*, q the *postcondition* and t the *result value*.[73]

Note that:

- The inclusion of a term t to denote the value returned by a command is a basic extension to traditional Hoare-style correctness formulas. It is motivated by the fact that commands in an object-based language, typically and most importantly the message sending (procedure call) command, can return a value and modify the state.
- For simplicity, a simpler version of the above intended interpretation is often examined first (i.e. one for the so-called *partial correctness*: see Section 3.2.3).

Most works on Hoare-style proof systems[74] restrict themselves to a simple semantic domain whereby the intended sets contain simply integer and boolean values or are simple cartesian products of these sets. In the case of object-based languages, the semantic domain is based on a more complex and difficult to reason about dynamically evolving structure of objects. This domain of computation occurs also when programming languages deal with structures dynamically created and manipulated by means of references (pointers). In this area, works on formal proofs of program correctness are more complex and scarce.[75] They have received relatively little attention and have not really been applied to give formal correctness proofs for programs of any significant size or complexity. We believe that object-based programming languages provide a good basis to extend the semantic domain commonly used in the area of program verification. We also think that this extended semantic domain can provide a basis to enlarge significantly the class of formally proven programs used in practice.

[71] This work uses a *model-oriented* approach to specifications. This is a very general and informal characterization, which is used to distinguish it from an *algebraic* (also called *property-oriented*) approach.

Informally, in the first approach any given specification is a syntactic element (usually, a well-formed phrase as given by a language syntax definition) for which a "model" (of something, say an "object") is defined in terms of more primitive (i.e. simpler or already defined) models.

Informally, in the second approach any given specification is a syntactic element (again a well-formed phrase as given by a language syntax definition) which contains the definition of a set of constant and function symbols together with their signatures, and either:

- equational axioms (possibly conditional), or
- axioms of first-order logic, or
- axioms of first-order logic and inductive definitions,

although the last option goes beyond what most authors would call "algebraic". To every specification a meaning function associates (one or possibly a set of) semantic objects which are *many-sorted algebras* (see e.g. [Gal86, Section 2.4]). For more details see e.g. [Wir91, FJ92].

[72] The formalization of the concept of "state" of an execution for an object-based language is given in Section 2.2. It may be instructive to note here that, because of the nature of object-based languages, the formalization does not lead just to an *assignment* function, which assigns values (in an appropriate semantic domain) to program variables, the natural formal counterpart of the informal concept of state of an execution for many imperative programming languages. Instead, there is an assignment function associated with each non-primitive object.

[73] Correctness formulas are often called *correctness assertions*, in connection with more traditional Hoare-style proof systems, but assertion is a name that becomes inadequate in the presence of the result value.

[74] See e.g. [dB80, Gri81, LS87, AO91, Fra92].

[75] See e.g. [OC75, Kow79, Mor82].

Issues in the design of the specification language

A basic characteristic of Hoare-style proof systems is the fact that the meaning of assertions is a predicate of *one* state (i.e. the initial state in the precondition and the final state in the postcondition).[76] This leads to the use of *logical variables*[77] — which are simply variables that cannot occur in programs (i.e. cannot occur in any phrase of the programming language), but only in phrases of the specification language — in order to refer in postconditions to the values denoted by variables in the initial state.

The first task of extending Hoare-style proof systems to cope with object-based languages involves the design and definition of a specification language. For that the semantic domain, which defines in particular the range of quantification of logical variables, must be defined.

A fundamental characteristic of the proof system presented in this thesis, which provides the basis for a relatively simple extension of Hoare-style proof systems to this rather more complex area, is the view of the semantic domain as one containing *object structures* (or *object graphs*).

An object can be a *primitive* or a *non-primitive* object. Non-primitive objects are those created dynamically, during the execution of a program. Primitive objects are predefined, i.e. supposed to exist before the execution of a program. An object structure is either an object (either primitive or non-primitive), *nil*, or a finite sequence $(a, \hat{d}_1, \ldots, \hat{d}_n)$, where a is a non-primitive object and $\hat{d}_1, \ldots, \hat{d}_n)$ are object structures (cf. Section 3.2.1).

Object structures are values denoted by program variables and by logical variables and terms of *basic* type in the specification language. A basic type is simply an object definition name. In **A1** it is either `Int` or `Bool`, which are primitive types, or any other object definition name occurring in a program, which constitute the non-primitive types.

Values in the semantic domain can also be, besides object structures, so-called *values of primitive type*. A value of primitive type is either *nil*, an integer or a boolean value, or a set, sequence or cartesian product of values of primitive type (see Section 3.2.1). They are denotations for corresponding constant terms of primitive type in the specification language.

Terms of non-primitive type can be indexed.

Indexed terms denote components of the original object structure (or sub object structures). Any component can be denoted via indexing, not only immediate components.[78] Any given set of program variables provides in this way an adequate "view" of the whole system of *existing* objects (namely, of those objects which compose the object structures denoted by those variables). Based on the decision that terms of non-primitive type denote object structures, it is natural to have a way of denoting components of this object structure, such components being of course themselves object structures: this is done by indexing.[79]

The range of quantification, which is determined by the type of logical variables, includes any value in the semantic domain. Because variables of non-primitive type denote object structures and can be indexed, quantifications over sequences of non-primitive objecs (i.e. objects created during the execution of a program) become then unnecessary and are not used. Also, due to the fact that program variables denote a structure of existing objects, it also becomes unnecessary to have a way of specifying quantifica-

[76] An alternative is to view the meaning of postconditions as predicates of two states (see e.g. [Jon90]).

[77] Also called auxiliary variables, ghost variables or specification variables.

[78] For example, if t denotes $(a, \hat{d}_1, \ldots, \hat{d}_n)$, where $n > 0$, then $t[1]$ denotes $\hat{d}_1$, etc. Also, if $\hat{d}_1 = (a', \hat{d}'_1, \ldots, \hat{d}'_m)$, where $m > 0$, then $t[1][1]$, which denotes $\hat{d}'_1$, is equivalent to $t[\underline{k}]$, for some appropriate k (see Section 3.2.1).

[79] An alternative would be to use "selection". For example, instead of an indexed term, say, $x[1]$, we would write, say, $x.y$ (if y was the 1-st object variable in the list of variables of the type of x). Indexing is used due to the following:

- It allows a simple form of quantification over "relevant" existing objects (consider, for example, $\forall i : x[i] \ldots$).
- It enables the use of *textual* substitution in the proof rules. With selection, substitutions in the proof rules would have to be more complicated. Consider a substitution of program variable x by term t' in term t. If indexing is used, any occurrence of x in t implies substitution, whereas this would not be the case if selection was used instead: an occurrence of x in $t_0.x$, for any term t_0 of appropriate type, could require substitution or not, depending on whether t_0 denotes the same object as the current one (i.e. the object with which such variable x is associated) or not.
- It provides a compact notation since one index can be used to refer to a variable which might require the use of many selections (see Section 3.2.2).

tions which are restricted only to such (i.e. the existing) objects. These quantifications create difficulties related to the fact that the range of quantifiers could then possibly be different between the initial and final states in a correctness formula.[80]

Consider a term $f(t_1, \ldots, t_n)$, where f is a function symbol and some of $t_1, \ldots, t_n$ are terms of non-primitive type. A natural interpretation for such a term is a function (taking object structures as arguments) that is recursively defined. We think that these terms appear naturally for specifications refering to sets and/or sequences of primitive objects that are components of an object structure. Their use also generally leads to shorter and simpler proofs.

The way object structures are represented in the semantic domain circumvents the difficulty that results from the fact that an assignment command may cause the modification of several object structures: i.e. those which have the object structure denoted by the modified variable as components (see Section 3.2). Informally speaking, the basic idea is the identification of a given non-primitive object, say a, with the object structure which has a at its "root" and whose immediate components are those denoted by the object variables associated with a.

A related problem is that of denoting the object structure rooted at the current object with one or more modified immediate components. For such purpose a term in the specification language can have immediate components explicitly indicated (see Section 3.1).

Significant issues and difficulties encountered in the construction of a proof system for **A1** were not specific to object-based programming languages and apply to other sequential imperative programming languages.

Among these were those related to proof rules for recursive procedures with parameters, which is still an area of active research.[81] Many proof systems in this area impose restrictions (e.g. only formal parameters allowed in the procedure body) or have basic disadvantages (such as the fact that proofs associated with each procedure call require corresponding separate proofs relative to the procedure body).[82] The complexity in this area is partly due to the complexity of the languages themselves, resulting from the combination of mechanisms such as parameter passing by value, by reference and/or by name, the fact that procedures may be arguments (in procedure calls), procedure nesting, global variables and recursion. Typically, object-based languages are relatively simple in this respect: parameter passing is only by value, procedures cannot be arguments and procedure declarations cannot be nested. However, important issues related to name clashing and recursion constitute a very significant part of the work of constructing a proof system for an object-based language.

By having separate and relatively simple rules for the procedure call (message sending) command (see Section 4.5), instead of just one complex rule, the overall complexity of the proof system is reduced. An important point here (as noted above in connection with the principle of abstraction) is the fact that proofs related to calls of any given procedure are done in general (the special case mentioned below being an exception) by using a single proof concerning the procedure body.

With respect to logical variables, an issue considered specially important in the context of object-based languages, but which is also significant in the context of other languages with support for dynamic creation and manipulation of data structures, is the forumlation of an instantiation rule for logical variables that is *not* based on a syntactic criterion formulated in terms of names of variables changed by programming language commands. The reasons why this is particularly important for object-based languages is that in purely syntactic terms there is no way of distinguishing variables of one object from variables of another object in the case these objects are instances of the same object definition. The instantiation rule used in **P1** (see Section 4.5, Rule i) is based on the invariance of equality between logical variables and terms, and is not based on any syntactic criteria. This invariance is proved, in the particular case just mentioned (of variables of distinct objects that are instances of the same object definition), by using the other rules of the proof system.

[80] See [AdB90a].
[81] See e.g. [Ak89, Hes93].
[82] Cf. e.g. [Apt81].

Chapter 2

The Programming Language

This chapter presents a simple object-based programming language named **A1** and defines its syntax and semantics.

A1 is a sequential imperative object-based language. It is designed so as to provide a simple yet meaningful framework for the study and analysis of formal correctness proofs for object-based programs. An informal description of **A1** is presented below.

The central syntactic component of the language is an *object definition* (cf. Section 1.3.2). A given object definition may introduce definitions of *object variables* and *methods*.[1] Every object created during the execution of a program (by an object creation command, see below) is said to be an instance of a given object definition. Object variables and methods occurring in a given object definition are then said to be associated with an object which is an instance of that object definition.

Object variables denote values. Definitions of object variables specify a *type*, which determines the set of values that the variables may denote.

Methods are said to be executed, activated, or called, by means of message sending commands[2]. A method definition introduces the formal parameters, the type of the result and a command which is called the *body* of the method. The execution of a method consists of an execution of its body in which the formal parameters denote the arguments given in the (message sending) command which caused the method's execution. The value returned by the method is the value returned by its body.

A command may be an assignment, message sending, object creation, sequential composition, conditional or block command.

The language distinguishes the concepts of commands and expressions. An expression denotes a value and does not modify the *state* (see Section 2.2). A command can in general return a value and modify the state: this is motivated by the nature of the message sending operation in object-based languages.

An assignment command is used to assign a value to a variable.

A message sending command is used to activate a method associated with the object denoted by a given expression.[3]

An object creation command is used to create a so-called non-primitive object which is an instance of a given object definition (as specified by an object definition name in that object creation command), give initial values to the object variables associated with that object (as given by expressions specified in that object creation command) and assign the created object to a variable.[4]

A sequential composition command is used to specify a command as being formed by a sequence of two commands in which the order in the sequence indicates the order in which the commands are executed.

[1] Methods can also be called procedures. There is not a way of specifying *internal* procedures, that is, procedures that are to be called only from within the object definition in which it is defined. (Other "restrictions" might be cited here: see comment on restrictions below.)

[2] Message sending commands can also be called method call or procedure call commands.

[3] Cf. informal explanation about method activation above.

[4] The assignment of the created object to a variable is motivated by the proof rule for the object creation command. See Section 4.5.

A conditional command is used to indicate, based on the value denoted by an expression (more specifically, an expression whose type is `Bool`), which of two commands to be executed.

A block command specifies variables to be used locally in a given command. This command is called the body of the block.

For reasons of simplicity — which are essential due to the fact that **A1** is not a language designed with the aim of being an *ultimate* tool for software development, but it is designed to be a simple but meaningful framework for the study and analysis of formal correctness proofs for object-based programs — **A1** does not include language constructs and mechanisms which would require the inclusion of material not fundamental for our main objectives and which would make us and readers deviate from our main focus of attention. In particular, we comment on the following:

1. Exclusion of an iteration command (e.g. a *while* command). The problems related with iteration commands are now significantly well understood, they are not specific to object-based languages and can be seen as a special case of the more general problem related to recursive procedure calls.

2. Exclusion of procedures for object creation. The problems of object creation can be studied under the availability of a predefined construct for such purpose, for which arguments may be passed as initial values of object variables, and the inclusion of procedures for object creation would duplicate the material related to recursive procedure calls (or require particular restrictions related to recursive procedures for object creation).

3. Restriction on the use of names of variables in a given object definition. For simplicity, in the semantics definition of **A1** we consider that only pairwise distinct names of variables occur in each object definition. The elimination of this restriction would introduce no significant changes; but it would make the treatment of assignments[5] less straightforward and the proof rule for blocks less simple[6].

The next two sections define respectively the syntax and semantics of **A1**.

2.1 Syntax

The syntax definition of **A1** is presented in this section, in the sequent calculus style introduced in Section 1.1.

Syntactic meta-variables are used in the meta-language (i.e. the language in which the description is written) to stand for *arbitrary* phrases of a corresponding phrase type of the object language (i.e. the language being defined).[7]

When used as the main phrase of a formula, the symbol n represents any integer constant, i.e. one of $\ldots, \mathtt{-1}, \mathtt{0}, \mathtt{1}, \ldots$.

For formulas of the form $h \circ Pt[h_1, \ldots, h_n]$, where $h, h_1, \ldots, h_n$ are phrases and Pt is a phrase type, we first introduce the following rule:

$$\frac{\Delta \vdash h \circ Pt[h_1, \ldots, h_n]}{\Delta \vdash h \circ Pt[h_i, h_{i+1}, \ldots, h_j]} \tag{1}$$

where $0 \leq i \leq j \leq n$.

[5]An assignment is a total function $s : X_s \to D$ from a domain containing a finite set of variables or formal parameters to a given semantic domain (see Section 2.2.2). Without this restriction, instead of assignments $s : X_s \to D$ we would have assignments $s : (V_s \to D) \times (Fp_s \to D) \times (Tp_s \to D)$. Then, instead of writing $s(x)$ for a variable or formal parameter $x \in X_s$, we would have to distinguish between object variables, formal parameters and temporary variables and write, for $s = (s_1, s_2, s_3)$, $s_1(x)$ for an object variable $x \in V_s$, $s_2(x)$ for a formal parameter $x \in Fp_s$ or $s_3(x)$ for a temporary variable $x \in Tp_s$.

[6]The proof rule for blocks would be similar to the proof rule for blocks already developed for a more traditional Hoare-style proof system (see e.g. [dB80, Page 243 (rule for blocks) and Lemma 6.26, Section 6.3]). It would just require additional definitions of free variables and textual substitution relative to commands of the programming language.

[7]If pg is a meta-variable that stands for arbitrary programs, whose phrase type is denoted by Pg, then the formula $pg \circ Pg$ can be read as "pg is a phrase of type Pg", or more simply as "pg is a program".

Informally, this means that if h is a phrase of type $Pt[h_1, \ldots, h_n]$, then h is also a phrase of type $Pt[s]$, where s is any sub-list of $h_1, \ldots, h_n$ (including the empty sub-list). We let ϵ denote the empty sub-list, and consider Pt, $Pt[\epsilon]$ and $Pt[]$ as denotations for the same phrase type.

Henceforth we use the following corresponding syntactic meta-variables and phrase types:

Meta-variable	Phrase type	
pg	Pg	programs
od	Od	object definitions
τ, odn	Odn	object definition names
md	Md	method definitions
m	Mn	method names
v	V	object variables
tp	Tp	temporary variables
x	X	variables
fp	Fp	formal parameters
c	C	commands
e	E	expressions
bk	Bk	blocks

The abstract syntax rules are given in Figure 2.1.

In Rule 1, for programs, A_1 corresponds to assumptions about the predefined names, as follows:[8]

$$
\begin{array}{rll}
A_1 \equiv & \texttt{Int} \circ Odn[& \texttt{1}, +, \texttt{Int}, \texttt{Int}, \\
 & & \texttt{1}, -, \texttt{Int}, \texttt{Int}, \\
 & & \texttt{1}, \ldots, \\
 & & \texttt{1}, >, \texttt{Bool}, \texttt{Int}, \\
 & & \texttt{1}, <, \texttt{Bool}, \texttt{Int}, \\
 & & \texttt{1}, \ldots], \\
 & \texttt{Bool} \circ Odn[& \texttt{1}, \neg, \texttt{Bool}, \\
 & & \texttt{1}, \vee, \texttt{Bool}, \texttt{Bool}, \\
 & & \texttt{1}, \ldots], \\
 & n \circ E[\texttt{Int}], & \\
 & \texttt{true} \circ E[\texttt{Bool}], & \\
 & \texttt{false} \circ E[\texttt{Bool}] &
\end{array}
$$

Note that above we write, for example, + followed by two occurrences of `Int`: the first `Int` represents the result type of (the method name) + and the second `Int` represents the parameter type. The fact that there is only one parameter reflects the "object-oriented view" of + as that of a message sending to an integer object, with another integer object as argument. We will, as usual, adopt the syntactic convention that $e_1 + e_2$ is syntactically equivalent to $e_1.+(e_2)$, for any expressions e_1, e_2 of type `Int`.

A_2 below corresponds to assumptions about names defined in the program (i.e. typing information), required to derive each well-formed object definition phrase. A_2 is given as follows:

$$A_2 \equiv \left\{ odn_i \circ Odn[\texttt{0}, \tau_i', \ldots, \tau_{is_i}', \texttt{1}, m_{ij}, \tau_{ij}, \tau_{ij1}, \ldots, \tau_{ijn_{ij}}, \texttt{1}] \right\}$$

where we have:

1. i ranges over 1 to k, for some $k \geq 1$,
2. for all such i, $s_i \geq 0$ and `0` occurs only once as an attribute of odn_i,
3. for all such i, j ranges over 1 to r_i, for some $r_i \geq 1$, $n_{ij} \geq 0$ and m_{ij} occurs only once as an attribute of odn_i

Program

$$\frac{\begin{array}{l} A_1, A_2, \texttt{self} \circ E[odn_1] \vdash od_1 \circ Od[\ \begin{array}[t]{l} \texttt{0}, \tau'_{11}, \ldots, \tau'_{1s_1}, \\ \texttt{1}, m_{11}, \tau_{11}, \tau_{111}, \ldots, \tau_{11n_{11}}, \\ \texttt{1}, \vdots, \\ \texttt{1}, m_{1r_1}, \tau_{1r_1}, \tau_{1r_1 1}, \ldots, \tau_{1r_1 n_{1r_1}}, \texttt{1}] \end{array} \\ \qquad\qquad\vdots \\ A_1, A_2, \texttt{self} \circ E[odn_k] \vdash od_k \circ Od[\ \begin{array}[t]{l} \texttt{0}, \tau'_{k1}, \ldots, \tau'_{ks_k}, \\ \texttt{1}, m_{k1}, \tau_{k1}, \tau_{k11}, \ldots, \tau_{k1n_{k1}}, \\ \texttt{1}, \vdots, \\ \texttt{1}, m_{kr_k}, \tau_{kr_k}, \tau_{kr_k 1}, \ldots, \tau_{kr_k n_{kr_k}}, \texttt{1}] \end{array} \end{array}}{\vdash odn_1\ od_1; \ldots; odn_k\ od_k \circ Pg} \tag{2}$$

where $odn_1, \ldots, odn_k$ are pairwise distinct elements

Object Definition

$$\frac{\begin{array}{c} \Delta, v_1 \circ V[\tau_1], \ldots, v_k \circ V[\tau_k] \vdash md_1 \circ Md[m_{i1}, \tau_{i1}, \tau_{i11}, \ldots, \tau_{i1n_{i1}}] \\ \vdots \\ \Delta, v_1 \circ V[\tau_1], \ldots, v_k \circ V[\tau_k] \vdash md_n \circ Md[m_{ir_i}, \tau_{ir_i}, \tau_{ir_i 1}, \ldots, \tau_{ir_i n_{ir_1}}] \end{array}}{\Delta \vdash v_1 : \tau_1, \ldots, v_k : \tau_k;\ md_1; \ldots; md_n \circ Od[\ \begin{array}[t]{l} \texttt{0}, \tau_1, \ldots, \tau_k, \\ \texttt{1}, m_{i1}, \tau_{i1}, \tau_{i11}, \ldots, \tau_{i1n_{i1}}, \\ \texttt{1}, \vdots, \\ \texttt{1}, m_{ir_i}, \tau_{ir_i}, \tau_{ir_i 1}, \ldots, \tau_{ir_i n_{ir_i}}, \texttt{1}] \end{array}} \tag{3}$$

where $v_1, \ldots, v_k$ are pairwise distinct elements

Method Definition

$$\frac{\Delta^{fp_1, \ldots, fp_n}, fp_1 \circ Fp[\tau_1], \ldots, fp_n \circ Fp[\tau_n] \vdash c \circ C[\tau]}{\Delta \vdash \texttt{method}\, m\, (fp_1 : \tau_1, \ldots, fp_n : \tau_n) : \tau\ c \circ Md[m, \tau, \tau_1, \ldots, \tau_n]} \tag{4}$$

where $fp_1, \ldots, fp_n$ are pairwise distinct elements

Variable

$$\frac{\Delta \vdash v \circ V[\tau]}{\Delta \vdash v \circ X[\tau]} \tag{5}$$

$$\frac{\Delta \vdash tp \circ Tp[\tau]}{\Delta \vdash tp \circ X[\tau]} \tag{6}$$

Figure 2.1: Abstract Syntax of **A1**

Expression as Command

$$\frac{\Delta \vdash e \circ E[\tau]}{\Delta \vdash e \circ C[\tau]} \tag{7}$$

Assignment Command

$$\frac{\begin{array}{c}\Delta \vdash x \circ X[\tau] \\ \Delta \vdash c \circ C[\tau]\end{array}}{\Delta \vdash x \leftarrow c \circ C[\tau]} \tag{8}$$

where $\tau \not\equiv \epsilon$

Message Sending Command

$$\frac{\begin{array}{c}\Delta \vdash \tau \circ Odn[\mathtt{1}, m, \tau', \tau_1, \ldots, \tau_n, \mathtt{1}] \\ \Delta \vdash e_0 \circ E[\tau] \\ \Delta \vdash e_1 \circ E[\tau_1] \\ \vdots \\ \Delta \vdash e_n \circ E[\tau_n]\end{array}}{\Delta \vdash e_0.m(e_1, \ldots, e_n) \circ Pt[\tau']} \tag{9}$$

where: $\begin{cases} Pt \equiv E \text{ if } \tau \equiv \mathtt{Int} \text{ or } \tau \equiv \mathtt{Bool}, \\ Pt \equiv C \text{ if } \tau \not\equiv \mathtt{Int} \text{ and } \tau \not\equiv \mathtt{Bool} \end{cases}$

Object Creation Command

$$\frac{\begin{array}{c}\Delta \vdash x \circ X[odn] \\ \Delta \vdash e_1 \circ E[\tau_1] \\ \vdots \\ \Delta \vdash e_n \circ E[\tau_n] \\ \Delta \vdash odn \circ Odn[\mathtt{0}, \tau_1, \ldots, \tau_n, \mathtt{1}]\end{array}}{\Delta \vdash x \leftarrow odn!\mathtt{new}(e_1, \ldots, e_n) \circ C[odn]} \tag{10}$$

Sequential Composition

$$\frac{\begin{array}{c}\Delta \vdash c_1 \circ C[\tau] \\ \Delta \vdash c_2 \circ C[\tau']\end{array}}{\Delta \vdash c_1; c_2 \circ C[\tau']} \tag{11}$$

Figure 2.1: Abstract Syntax of **A1** (continued)

Conditional

$$\frac{\begin{array}{c}\Delta \vdash h \circ Pt[\texttt{Bool}] \\ \Delta \vdash h_1 \circ Pt[{h_1}'] \\ \Delta \vdash h_2 \circ Pt[{h_2}']\end{array}}{\Delta \vdash \texttt{if}\ h\ \texttt{then}\ h_1\ \texttt{else}\ h_2 \circ Pt[h']} \tag{12}$$

where: $\begin{cases} Pt \equiv C \text{ or } Pt \equiv E, \\ h' \equiv {h_2}' \equiv h_1 \text{ and } h' \not\equiv \epsilon, \text{ if } Pt \equiv E, \\ h' \equiv \epsilon \text{ if } Pt \equiv C \end{cases}$

Block as Command

$$\frac{\Delta \vdash bk \circ Bk[\tau]}{\Delta \vdash bk \circ C[\tau]} \tag{13}$$

Variable as Expression

$$\frac{\Delta \vdash x \circ X[\tau]}{\Delta \vdash x \circ E[\tau]} \tag{14}$$

Formal Parameter as Expression

$$\frac{\Delta \vdash fp \circ Fp[\tau]}{\Delta \vdash fp \circ E[\tau]} \tag{15}$$

Nil

$$\frac{\Delta \vdash odn \circ Odn}{\Delta \vdash \texttt{nil} \circ E[odn]} \tag{16}$$

Equality

$$\frac{\begin{array}{c}\Delta \vdash e_1 \circ E[\tau] \\ \Delta \vdash e_2 \circ E[\tau]\end{array}}{\Delta \vdash e_1 \doteq e_2 \circ E[\texttt{Bool}]} \tag{17}$$

Block

$$\frac{\Delta^{tp_1,\ldots,tp_n}, tp_1 \circ Tp[\tau_1], \ldots, tp_n \circ Tp[\tau_n] \vdash c \circ C[\tau]}{\Delta \vdash tp_1 : \tau_1, \ldots, tp_n : \tau_n\ c \circ Bk[\tau]} \tag{18}$$

where $n \geq 0$ and $tp_1, \ldots, tp_n$ are pairwise distinct elements

Figure 2.1: Abstract Syntax of **A1** (continued)

We often make use of terms given by the following:

Definition 1 Let

$$pg \equiv \tau_1\, od_1; \ldots \tau_k\, od_k$$

For an arbitrary $\tau_i \in \{\tau_1, \ldots, \tau_k\}$, let then:

$$od_i \equiv v_1 : odn_1, \ldots, v_n : odn_n;\ md_1; \ldots; md_l$$

Then, for all $j = 1, \ldots, l$, let:

$$md_j \equiv \texttt{method}\, m_j\, (fp_{j1} : \tau_{j1}, \ldots, fp_{jr_j} : \tau_{jr_j})\, \tau'_j\, c_j$$

where $k \geq 1$, $l \geq 1$, $n \geq 0$ and $r_j \geq 1$, for all such j; we call:

- $\{\tau_1, \ldots, \tau_k\}$ the *set of object definition names* of pg;
- $v_1, \ldots, v_n$ the *list of variables* of τ_i, and $\{m_1, \ldots, m_l\}$ the *set of method names* of τ_i;
- od_i the *object definition associated with* τ_i or simply the *object definition* τ_i;
- c_j the *body* of m_j and $fp_1, \ldots, fp_n$ the (list of) *formal parameters* of m_j.

The concept of an expression, command or block *type* (and indeed all the type rules of the language) is defined here under a syntactic characterization: for any expression e, if $e \circ E[\tau]$ (i.e. if e is an expression of phrase type $E[\tau]$), then we say that *e is of type τ*, or similarly that *τ is the type of e*. The same is valid for commands and blocks. Thus, more generally we have the following:

Definition 2 (Type of an expression, command or block) Let h denote an expression, command or block, and Pt denote respectively phrase type E, C or Bk. We say *h is of type τ* (similarly, *h has type τ* or τ is the type of h) if $h \circ Pt[\tau]$ and $\tau \circ Odn$. Otherwise, if $h \circ Pt[\epsilon]$, we say that *h has no type*.

Thus, in **A1** a *type* is any element (well-formed phrase) of phrase type Odn. `Int` and `Bool` are called *primitive types* and other types referenced in the program are called *non-primitive types*.

Figure 2.2 presents a more compact (but not equivalent) syntax definition, using *pattern rules*[9], which is more suitable for quick references.

The syntax rules given describe an *abstract* syntax. By this we mean that we are not interested in the concrete syntactic representation of a phrase but in its abstract structure, i.e. of it being formed, also abstractly, by phrases of the given phrase types. Terminal symbols, which wouldn't be necessary in a "pure" abstract syntax definition, are used for readability reasons.

2.1.1 Example

A short example to show how the syntactic rules are used to deduce that a program is (syntactically) well-formed is given below. The program is as follows:

[8] The symbols `0` and `1`, when used as attributes in formulas used to define the syntax of **A1**, serve as delimiters, separating information (i.e. other phrases) relative to different attributes. They are used to distinguish, for example (cf. syntax below), the attributes of a phrase that is an object definition with one variable x and a parameterless method m from those of a phrase that is a method definition with two parameterless methods m and x.

[9] Cf. nomenclature used in [Ast91]. The use of the name "pattern rules" is not common; the notation is simply a variant of BNF (Backus-Naur formalism), where indices are used for non-terminals (in the right hand side of rules) and terminals and non-terminals are distinguished simply by giving a separate definition.

$$
\begin{array}{lcl}
pg & ::= & odn_1\ od_1; \ldots; odn_k\ od_k \\[1ex]
od & ::= & v_1 : odn_1, \ldots, v_k : odn_k;\ md_1 \ldots md_n \\[1ex]
md & ::= & \texttt{method}\, m(fp_1 : odn_1, \ldots\ fp_n : odn_n) : odn\ c \\[1ex]
x & ::= & tp \mid\ v \\[1ex]
c & ::= & e \\
 & \mid & x \leftarrow\ c \\
 & \mid & e_0.m(e_1, \ldots, e_n) \\
 & \mid & x \leftarrow\ odn!\texttt{new}(e_1, \ldots, e_n) \\
 & \mid & c_1; c_2 \\
 & \mid & \texttt{if}\, c\, \texttt{then}\, c_1\, \texttt{else}\, c_2 \\
 & \mid & bk \\[1ex]
e & ::= & x \mid fp \mid n \mid \texttt{false} \mid \texttt{true} \mid \texttt{self} \mid \texttt{nil} \\
 & \mid & e_1 \doteq e_2 \mid \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \\[1ex]
n & ::= & \texttt{0} \mid \texttt{-1} \mid \texttt{1} \mid \ldots \\
bk & ::= & tp_1 : odn_1, \ldots\ tp_n : odn_n\ c
\end{array}
$$

Figure 2.2: Abstract Syntax of A1 using pattern rules

Cnt
x : `Int`

`method` *inc*
$x \leftarrow x + 1$

`method` *val* : `Int`
x

A syntactic convention is used in the above program and throughout this document; a method definition

$$\texttt{method}\, m(fp_1 : \tau_1, \ldots, fp_n : \tau_n) : \tau\ c;\ \texttt{self}$$

occurring in an object definition associated with τ (see Definition 1) is abbreviated to:

$$\texttt{method}\, m(fp_1 : \tau_1, \ldots, fp_n : \tau_n)\ c$$

For brevity, let:

- od_1 be (syntactically equivalent to) the piece of code following *Cnt*
- $md_1 \equiv \texttt{method}\, inc\ \ x \leftarrow x + 1$
- $md_2 \equiv \texttt{method}\, val : \texttt{Int}\ \ x$
- A_1 be as defined above
- $A_2 \equiv Cnt \circ Odn[\texttt{0}, \texttt{Int}, \texttt{1}, inc, Cnt, \texttt{1}, val, \texttt{Int}, \texttt{1}]$

- $\Delta \equiv A_1, A_2, \texttt{self} \circ E[Cnt]$
- $\Delta' \equiv \Delta, x \circ V[\texttt{Int}]$
- $p \equiv \;\; \texttt{Int} \circ Odn[\;\; \begin{array}{l} \texttt{1}, +, \texttt{Int}, \texttt{Int}, \\ \texttt{1}, -, \texttt{Int}, \texttt{Int}, \\ \texttt{1}, \ldots, \\ \texttt{1}, >, \texttt{Bool}, \texttt{Int}, \\ \texttt{1}, <, \texttt{Bool}, \texttt{Int}, \\ \texttt{1}, \ldots] \end{array}$
- $p' \equiv \texttt{Int} \circ Odn[\texttt{1}, +, \texttt{Int}, \texttt{Int}, \texttt{1}]$

The following derivation uses the syntactic rules in Figure 2.1:

$$\dfrac{\dfrac{\dfrac{(A) \quad \dfrac{\dfrac{\dfrac{\Delta' \vdash p}{\Delta' \vdash p'}\,{\scriptstyle 1} \quad (B)}{\Delta' \vdash x + 1 \circ E[\texttt{Int}]}\,{\scriptstyle 9}}{\Delta' \vdash x + 1 \circ C[\texttt{Int}]}\,{\scriptstyle 7}}{\Delta' \vdash x \leftarrow x + 1 \circ C[\texttt{Int}]}\,{\scriptstyle 8} \quad \dfrac{\Delta' \vdash \texttt{self} \circ E[Cnt]}{\Delta' \vdash \texttt{self} \circ C[Cnt]}\,{\scriptstyle 7}}{\dfrac{\Delta' \vdash x \leftarrow x + 1;\ \texttt{self} \circ C[Cnt]}{\Delta' \vdash md_1 \circ Md[inc, Cnt]}\,{\scriptstyle 4}}\,{\scriptstyle 11} \qquad \dfrac{\dfrac{(B)}{\Delta' \vdash x \circ C[\texttt{Int}]}\,{\scriptstyle 7}}{\Delta' \vdash md_2 \circ Md[val, \texttt{Int}]}\,{\scriptstyle 4}}{\dfrac{\Delta \vdash od_1 \circ Od[\texttt{0}, \texttt{Int}, \texttt{1}, inc, Cnt, \texttt{1}, val, \texttt{Int}, \texttt{1}]}{\vdash Cnt\ od_1 \circ Pg}\,{\scriptstyle 2}}\,{\scriptstyle 3}$$

and:

$$\dfrac{\Delta' \vdash x \circ V[\texttt{Int}]}{\Delta' \vdash x \circ X[\texttt{Int}]}\,{\scriptstyle 5} \tag{A}$$

and:

$$\dfrac{(A)}{\Delta' \vdash x \circ E[\texttt{Int}]}\,{\scriptstyle 14} \tag{B}$$

2.2 Semantics

This section defines the semantics of **A1**, using the method of denotational semantics. An introduction to this method is given in Section 1.2.3.

The semantics of **A1** is defined in relation to an interpretation **I** which is a pair $(D, \mathcal{I})$, where D is a *semantic domain* of the interpretation **I**, and $\mathcal{I}$ is an *interpretation function* for constant and function symbols, mapping them to elements and functions over D, respectively.

The semantic functions, defined in Section 2.2.2, are given in relation to a standard interpretation **I** with a semantic domain D as defined in the next section and an interpretation function $\mathcal{I}$ such that:

$$\begin{array}{l} \mathcal{I}(\texttt{true}) = true, \\ \mathcal{I}(\texttt{false}) = false, \\ \mathcal{I}(\texttt{nil}) = nil, \\ \mathcal{I}(\texttt{0}) = 0, \\ \mathcal{I}(\texttt{-1}) = -1, \\ \mathcal{I}(\texttt{1}) = 1, \\ \vdots \end{array}$$

where the elements $true, false, nil, 0, -1, 1, \ldots$ are all members of D (as defined in the next section).

The semantic functions are functions that generally depend on *assignments* of values in D to program variables. It should also be noted that:

- In **A1** there is a (finite) set of variables associated with each *non-primitive object*[10]. Given two distinct non-primitive objects a, a', a variable x associated with a is distinct from variable x associated with a'. For this reason the semantics of **A1** uses a (countably infinite) set of assignments, one for each object, where all such assignments are finite mappings.

- In **A1** there is a (finite) set of method names associated with each object definition, and for each one of them there corresponds a semantic function.[11]

We hereafter use the fact that to every phrase type in the syntax there corresponds respectively a set, denoted by the same name, the set being formed by all well-formed phrases of that phrase type. Using this, we also make an abuse out of the notation and write, for example, $h \circ Pt$ instead of $h \in Pt$, if h is a member of the set Pt.[12]

2.2.1 Semantic domain

For each type τ, we consider a semantic domain D_τ associated with it, called the *domain* (or *carrier*) of τ.

Definition 3 (Domains of primitive types) For the *primitive types* `Int` and `Bool`, we define:

$$D_{\mathtt{Int}} = \mathbf{Z} \cup \{nil\}$$

$$D_{\mathtt{Bool}} = \mathbf{B} \cup \{nil\}$$

where $\mathbf{Z} = \{\ldots, -1, 0, 1, \ldots\}$ and $\mathbf{B} = \{false, true\}$ are respectively the set of integers and boolean values.

The integers and boolean values are called *primitive objects*.

The element nil is such that $\mathcal{I}(\mathtt{nil}) = nil$ (as assumed above for the interpretation function) and $nil \notin (\mathbf{Z} \cup \mathbf{B})$.

Definition 4 (Domains of non-primitive types) For a non-primitive type τ, we consider a countably infinite set A_τ of *non-primitive objects* of type τ, such that:

1. $nil \notin A_\tau$
2. $A_\tau \cap A_{\tau'} = \emptyset$, if $\tau' \not\equiv \tau$

We define:

$$D_\tau = A_\tau \cup \{nil\}$$

The set A is the union of all A_τ:

$$A = \bigcup_{\tau \in \mathcal{T}} A_\tau$$

The meta-variables a and α denote arbitrary elements of A.
We also let:

$$\Theta = A \cup D_{\mathtt{Bool}} \cup D_{\mathtt{Int}}$$

[10] See definition 4 below.
[11] It can thus be said that a "function variable" is a pair composed of a method name and an object definition name.
[12] We want with this notation to make apparent that (semantic) predicates such as $h \in Pt$ are a result of the syntactic structure of a program.

The elements of A are called *non-primitive objects*.

The elements of Θ distinct from *nil* are called *objects*.[13] When the context eliminates any risk of confusion, we sometimes use the shorter term *object* instead of *non-primitive object*.

For a type τ and sequences of type τ^* we define $D_{\tau^*} = (D_\tau)^*$, and analogously for sets, cartesian products and related operations.

2.2.2 The semantic functions

We define in this section the semantics of blocks, programs, commands and expressions of **A1**.

Generally, for a given semantic domain, say D, we consider:

$$D_\perp = D \cup \{\perp_D\}$$

where $\perp_D$ is an element indicating a situation of non-termination, $\perp_D \notin D$. Similarly, we have:

$$D_{error} = D \cup \{error\}$$

where *error* is an element indicating an error, distinct from $\perp_D$.

The symbol ε is used to denote an element which is either *error* or $\perp_D$. The corresponding semantic domain D_ε is given by:[14]

$$D_\varepsilon = D \cup \{error, \perp_D\}$$

We call an *assignment* a total function $s : X_s \to D$ from a domain containing a finite set of variables or formal parameters to a given semantic domain.

This section considers assignments s for which the domain is a finite set $X_s \cup \{\texttt{self}\}$. X_s is a set containing the variables in the list of variables of a non-primitive type τ (cf. Definition 1) and is such that if $x \in X_s$ is of type τ then $s(x) \in D_\tau$. The set of all such finite functions is denoted by S.

We call a *state* σ a finite total function $\sigma : A_\sigma \to S$ from a finite set of non-primitive objects[15] to the set of all assignments S. The set of all such finite functions is denoted by Σ.

We need to define first how a program uniquely determines an assignment of functions to "function variables" (i.e. pairs of object definition names and method names). These functions are functions:

$$g : A \to \Theta^k \to \Sigma \to (\Sigma \times \Theta)_\varepsilon$$

where k is a given number, $k \geq 1$.

The set of all such functions is denoted by G_ε.[16]

We call a *function assignment* a finite total function $\gamma : (Odn \times Mn)_\gamma \to G_\varepsilon$ from a finite set of function variables (i.e. pairs whose first elements are object definition names and whose second elements are method names)[17] to the domain G_ε, such that:

- if $(\tau, m) \in \texttt{dom}(\gamma)$ then $\tau \notin \{\texttt{Int}, \texttt{Bool}\}$;
- if $(\tau, m) \in \texttt{dom}(\gamma)$ then m is a member of the set of method names of τ (cf. Definition 1).

---

[13] What we call *object* in relation to a semantic domain is generally refered to, in object-oriented programming language terminology, as *object identity*.

[14] The notation D_ε is meant to lead to a partial order $\big(D_\varepsilon, \sqsubseteq\big)$ which is a flat cpo-with-bottom, with least element $\perp_D$.

[15] I.e. A_σ is a finite subset of A.

[16] Note that the partial order $\big((\Sigma \times \Theta)_\varepsilon, \sqsubseteq\big)$ is a (flat) cpo-with-bottom, with least element $\perp_{\Sigma\times\Theta}$.

The partial order $\big(G_\varepsilon, \sqsubseteq\big)$ is a cpo-with-bottom (cf. the analogous, for cpos-with-bottom, of Proposition 1, page 11), for which we let:

$$\perp_G = \lambda a.\lambda\bar{\theta}.\lambda\sigma.\perp_{\Sigma\times\Theta}$$

where $\bar{\theta}$ indicates an element of Θ^k.

[17] I.e. $(Odn \times Mn)_\gamma$ is a given finite subset of $Odn \times Mn$.

The set of all such function assignments is denoted by Γ.

We now show how a program uniquely determines a function assignment, denoted by γ_f.

Let pg be a program, $\{odn_1, \ldots, odn_n\}$ be the set of object definition names of pg, $n \geq 1$. For all $i = 1, \ldots, n$, let $\{m_{i1}, \ldots, m_{ik_i}\}$ be the set of method names of odn_i, $k_i \geq 1$, and for all $j = 1, \ldots, k_i$ let c_{ij} be the body of m_{ij} and $fp_{ij1}, \ldots, fp_{ijk_{ij}}$ the formal parameters of m_{ij}.

Let then γ_f be generally given by:

$$\gamma_f = \{(odn_i, m_{ij}) \mapsto f_{ij}\}$$

where: $\begin{cases} i = 1, \ldots, n \\ \text{for all such } i\text{, we have } j = 1, \ldots, k_i \end{cases}$

Then, considering $\mathbf{I}[\![c_{ij}]\!]$ as semantic functions for commands (cf. Definition 7 below), the sequence

$$(f_{11}, \ldots, f_{1k_1}, \ldots, f_{n1}, \ldots, f_{nk_n})$$

of the elements f_{ij} is defined as the least fixed point of the functional $\Phi[\![pg]\!] : (G_\varepsilon)^k \to (G_\varepsilon)^k$, where $k = k_1 + \cdots + k_n$, given by:

$$\begin{aligned} \Phi[\![pg]\!](g) = \; & \Big(\lambda\alpha.\lambda\bar{\theta}_{11}.\lambda\sigma.\mathbf{I}[\![c_{11}]\!](\alpha)(\gamma)(\sigma_{11}), \ldots, \\ & \lambda\alpha.\lambda\bar{\theta}_{1k_1}.\lambda\sigma.\mathbf{I}[\![c_{1k_1}]\!](\alpha)(\gamma)(\sigma_{1k_1}), \ldots, \\ & \lambda\alpha.\lambda\bar{\theta}_{n1}.\lambda\sigma.\mathbf{I}[\![c_{n1}]\!](\alpha)(\gamma)(\sigma_{n1}), \ldots, \\ & \lambda\alpha.\lambda\bar{\theta}_{nk_n}.\lambda\sigma.\mathbf{I}[\![c_{nk_n}]\!](\alpha)(\gamma)(\sigma_{nk_n})\Big) \end{aligned} \tag{i}$$

where we have, for all i, j:

1. γ is the function assignment given by:

   $$\gamma = \{(odn_i, m_{ij}) \mapsto g_{ij}\}$$

   for $g = (g_{11}, \ldots, g_{1k_1}, \ldots, g_{n1}, \ldots, g_{nk_n})$

2. $\bar{\theta}_{ij} = (\theta_{ij1}, \ldots, \theta_{ijk_{ij}})$

3. $\sigma_{ij} = \sigma \dagger \big\{\alpha \mapsto \sigma(\alpha) \dagger \{fp_{ij1} \mapsto \bar{\theta}_{ij1}, \ldots, fp_{ijk_{ij}} \mapsto \bar{\theta}_{ijk_{ij}}\}\big\}$

The functional $\Phi[\![pg]\!]$ is continuous (see Appendix B) and, as a consequence of the Fixed Point Theorem (page 12), it has a unique least fixed point. Therefore, the function assignment γ_f is uniquely determined.

To define the semantics of programs, we define first the semantics of blocks.

Definition 5 (Meaning of blocks) For each block $bk \equiv tp_1 : odn_1, \ldots, tp_n : odn_n\ c$, if bk has type τ, then we define a function

$$\mathbf{I}[\![bk]\!] : A \to \Gamma \to \Sigma \to (\Sigma \times D_\tau)_\varepsilon$$

and if bk has no type, then we define a function

$$\mathbf{I}[\![bk]\!] : A \to \Gamma \to \Sigma \to \Sigma_\varepsilon$$

as follows:

$$\mathbf{I}[\![bk]\!](\alpha)(\gamma)(\sigma) = \mathbf{I}[\![c]\!](\alpha)(\gamma)\big(\sigma \dagger \big\{\alpha \mapsto \sigma(\alpha) \dagger \{tp_1 \mapsto nil, \ldots, tp_n \mapsto nil\}\big\}\big)$$

Definition 6 (Meaning of programs) Consider the function assignment γ_f defined above and an object α_0 such that, for all $\sigma \in \Sigma$, we have:

1. $\sigma(\alpha_0)(\texttt{self}) = nil$,
2. $\sigma(\alpha)(\texttt{self}) = \alpha$, for all $\alpha \in \texttt{dom}(\sigma), \alpha \neq \alpha_0$

The meaning of a program pg is the set of all functions (one function for each block):

$$\mathbf{I}[\![pg]\!][\![bk]\!] : \Sigma \to (\Sigma \times D_\tau)_\varepsilon$$

if bk has type τ, and

$$\mathbf{I}[\![pg]\!][\![bk]\!] : \Sigma \to \Sigma_\varepsilon$$

if bk has no type, given by:

$$\mathbf{I}[\![pg]\!][\![bk]\!](\sigma) = \mathbf{I}[\![bk]\!](\alpha_0)(\gamma_f)(\sigma)$$

Informally, $\sigma(\alpha_0)(\texttt{self}) = nil$ indicates that `self` denotes no object in any expression that occurs in block bk. Note that block bk is not part of program pg. In operational terms, object α_0 is "the object executing block bk"; α_0 may be called an "external" object, which interacts with the program via message sending and object creation commands.

Note that we could have adopted the approach of considering the meaning of a program pg as being the set of all values $(D_\tau)_\varepsilon$ given by $\mathbf{I}[\![bk]\!](\alpha_0)(\gamma_f)(\emptyset)$. In this case the initial state would be empty and any non-predefined object would be created by commands in block bk. We have not done so in order to take a view more directed to persistency of programs, common in object-based environments and applications.

We define now the semantic functions for commands.

Definition 7 (Meaning of commands) For each command c, if c has type τ, then we define a function

$$\mathbf{I}[\![c]\!] : A \to \Gamma \to \Sigma \to (\Sigma \times D_\tau)_\varepsilon$$

and if c has no type, then we define a function

$$\mathbf{I}[\![c]\!] : A \to \Gamma \to \Sigma \to \Sigma_\varepsilon$$

by induction on the structure of commands, as follows:

- if $c \equiv e$, then:

$$\mathbf{I}[\![c]\!](\alpha)(\gamma)(\sigma) = \begin{cases} (\sigma, \mathbf{I}[\![e]\!](\sigma(\alpha))) & \text{if } \mathbf{I}[\![e]\!](\sigma(\alpha)) \neq \varepsilon \\ \varepsilon & \text{otherwise} \end{cases}$$

- if $c \equiv x \leftarrow c_1$, then consider:
  1. $\mathbf{I}[\![c_1]\!](\alpha)(\gamma)(\sigma) = (\sigma', \theta)$; then[18]:

$$\mathbf{I}[\![c]\!](\alpha)(\gamma)(\sigma) = \big(\sigma' \dagger \{\alpha \mapsto \sigma'(\alpha) \dagger \{x \mapsto \theta\}\}, \theta\big)$$

  2. $\mathbf{I}[\![c_1]\!](\alpha)(\gamma)(\sigma) = \varepsilon$; then:

$$\mathbf{I}[\![c]\!](\alpha)(\gamma)(\sigma) = \varepsilon$$

[18] As an illustrative example of the specification of context conditions simplifying the semantic definition, note that we use here the fact the syntactic rule for assignment commands (Rule (8)) forces c_1 to have a type.

- if $c \equiv e_0.m(e_1, \ldots, e_n)$, let τ be the type of e_0; if τ is a primitive type then c is an expression (se Rule 9) and the definition is that given in the first item above; otherwise consider:
  - $\mathbf{I}[\![e_0]\!](\sigma(\alpha)) = \varepsilon$; then:
$$\mathbf{I}[\![c]\!](\alpha)(\gamma)(\sigma) = \varepsilon$$
  - $\mathbf{I}[\![e_0]\!](\sigma(\alpha)) = nil$; then:
$$\mathbf{I}[\![c]\!](\alpha)(\gamma)(\sigma) = error$$
  - otherwise, consider:
    * $\mathbf{I}[\![e_i]\!](\sigma(\alpha)) = \varepsilon$, for some i between 1 and n; then:
$$\mathbf{I}[\![c]\!](\alpha)(\gamma)(\sigma) = \varepsilon$$
    * otherwise, let $\mathbf{I}[\![e_i]\!](\sigma(\alpha)) = \theta_i$, for $i = 0, 1, \ldots, n$; then:
$$\mathbf{I}[\![c]\!](\alpha)(\gamma)(\sigma) = \gamma(\tau, m)(\theta_0)(\theta_1, \ldots, \theta_n)(\sigma)$$
- if $c \equiv x \leftarrow odn!\mathtt{new}(e_1, \ldots, e_n)$, let $v_1, \ldots, v_n$ be the list of variables of odn, $\mathbf{I}[\![e_i]\!](\sigma(\alpha)) = \theta_i$, for $i = 1, \ldots, n$ and consider any $a \in A_{odn}$ such that $a \notin \mathtt{dom}(\sigma)$; we have:
$$\begin{aligned}\mathbf{I}[\![c]\!](\alpha)(\gamma)(\sigma) = \big(\sigma\ & \dagger\big\{a \mapsto \{\mathtt{self} \mapsto a, v_1 \mapsto \theta_1, \ldots, v_n \mapsto \theta_n\}\big\} \\ & \dagger\big\{\alpha \mapsto \sigma(\alpha) \dagger \{x \mapsto a\}\big\}, a\big)\end{aligned}$$
- if $c \equiv c_1; c_2$, then consider:
  1. $\mathbf{I}[\![c_1]\!](\alpha)(\gamma)(\sigma) = (\sigma', \theta)$ or $\mathbf{I}[\![c_1]\!](\alpha)(\gamma)(\sigma) = \sigma'$; then:
$$\mathbf{I}[\![c]\!](\alpha)(\gamma)(\sigma) = \mathbf{I}[\![c_2]\!](\alpha)(\gamma)(\sigma')$$
  2. $\mathbf{I}[\![c_1]\!](\alpha)(\gamma)(\sigma) = \varepsilon$; then:
$$\mathbf{I}[\![c]\!](\alpha)(\gamma)(\sigma) = \varepsilon$$
- if $c \equiv \mathtt{if}\ c_0\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2$, then:
$$\mathbf{I}[\![c]\!](\alpha)(\gamma)(\sigma) = \begin{cases} \mathbf{I}[\![c_1]\!](\alpha)(\gamma)(\sigma') & \text{if } \mathbf{I}[\![c_0]\!](\alpha)(\gamma)(\sigma) = (\sigma', true) \\ \mathbf{I}[\![c_2]\!](\alpha)(\gamma)(\sigma') & \text{if } \mathbf{I}[\![c_0]\!](\alpha)(\gamma)(\sigma) = (\sigma', false) \\ \varepsilon & \text{if } \mathbf{I}[\![c_0]\!](\alpha)(\gamma)(\sigma) = \varepsilon \\ error & \text{if } \mathbf{I}[\![c_0]\!](\alpha)(\gamma)(\sigma) = nil \end{cases}$$
- if $c \equiv bk$, then
$$\mathbf{I}[\![c]\!] = \mathbf{I}[\![bk]\!]$$

We give next the semantic functions for expressions.

Definition 8 (Meaning of expressions) For each expression e, where $e \circ E[\tau]$, we define a function
$$\mathbf{I}[\![e]\!] : S \to (D_\tau)_\varepsilon$$
by induction on the structure of expressions, as follows:

- if $e \equiv x$ or $e \equiv fp$ or $e \equiv \texttt{self}$, then:

$$\mathbf{I}[\![e]\!](s) = s(e)$$

- if e is either $\texttt{true}, \texttt{false}, \texttt{nil}, \texttt{0}, \texttt{1}, -\texttt{1}, \ldots$, then:

$$\mathbf{I}[\![e]\!](s) = \mathcal{I}(e)$$

- if $e \equiv e_1 \doteq e_2$, then:

$$\mathbf{I}[\![e]\!](s) = \begin{cases} \big(\mathbf{I}[\![e_1]\!](s) = \mathbf{I}[\![e_2]\!](s)\big) & \text{if } \mathbf{I}[\![e_1]\!](s) \neq \varepsilon \text{ and } \mathbf{I}[\![e_2]\!](s) \neq \varepsilon \\ \varepsilon & \text{otherwise} \end{cases}$$

- if $e \equiv \texttt{if}\; e_0 \;\texttt{then}\; e_1 \;\texttt{else}\; e_2$, then:

$$\mathbf{I}[\![e]\!](s) = \begin{cases} \mathbf{I}[\![e_1]\!](s) & \text{if } \mathbf{I}[\![e_0]\!](s) = true \\ \mathbf{I}[\![e_2]\!](s) & \text{if } \mathbf{I}[\![e_0]\!](s) = false \\ \varepsilon & \text{if } \mathbf{I}[\![e_0]\!](s) = \varepsilon \\ error & \text{if } \mathbf{I}[\![e_0]\!](s) = nil \end{cases}$$

- if $e \equiv e_0.m(e_1, \ldots, e_n)$, where $e_0 \circ E[\tau]$ and τ is a primitive type, then:

$$\mathbf{I}[\![e]\!](s) = \begin{cases} error & \text{if } \mathbf{I}[\![e_0]\!](s) = nil \\ \varepsilon & \text{if } \mathbf{I}[\![e_0]\!](s) \neq nil \\ & \text{and } \mathbf{I}[\![e_i]\!](s) = \varepsilon, \\ & \text{ for some } i \in \{0, \ldots, n\} \\ \mathcal{I}(m)\big(\mathbf{I}[\![e_0]\!](s), \mathbf{I}[\![e_1]\!](s), \ldots, \mathbf{I}[\![e_n]\!](s)\big) & \text{otherwise} \end{cases}$$

Chapter 3

The Specification Language

This chapter defines the syntax and semantics of a specification language named **L1**. It defines also the syntax and formal semantics of *correctness formulas* (for partial correctness), which give a formalization of the notion of programs satisfying specifications.

The phrases of **L1** are terms and formulas. Formulas use logical connectives of a language of predicate logic. The basic terms of **L1** are expressions of **A1** and logical variables. Terms of non-primitive type can be indexed, can have direct components explicitly indicated and can have a syntactic form that indicates a possible change of interpretation of a component.

Commands of the programming language **A1** are phrases occurring in correctness formulas, which also include terms and formulas of **L1**.

The expressiveness of **L1** comes mainly from the fact that terms of non-primitive type can be indexed — providing a way of referencing objects which are part of a certain object structure and a way of specifying quantifications over the existing objects of such object structure — and from the fact that arguments of non-primitive type are given (expanded) interpretations which are sequences upon which the corresponding functions can be recursively defined. This provides a tool which allows us to deal formally and in a simple way with the problem of reasoning about "dynamically evolving" data structures.

We consider a countable set $\mathcal{T}$ of types (or *sorts*) of the language, where a type is either:

1. a *non-primitive type*, which is a well-formed phrase of phrase type *Odn*, $Odn \not\equiv$ `Int` and $Odn \not\equiv$ `Bool`, as given by the syntax rules of **A1**, or

2. a *primitive type*, which is either `Int`, `Bool`, `Z`, `B` or a set, sequence or cartesian product of elements of primitive type. `Z` is a type with the same constant symbols of `Int` but `nil`. Similarly for `B` and `Bool`.

A well-formed phrase of phrase type *Odn* is also called a *basic type*.

The alphabet of the language includes countable sets Lv of logical variables and F of function symbols.

Any logical variable has a type: the set Lv is the union of countable sets Lv_τ, where each logical variable in Lv_τ has type τ.

Any function symbol has a *functional type*, of the form $(\tau_1, \ldots, \tau_n \to \rho)$, where $n > 0$. n is called the *arity* of the function symbol. ρ is called the *result type* and τ_i the i-th *parameter type*, $i = 1, \ldots, n$. When speaking generally about types, we do not refer to functional types.

Constant indices in indexed terms are written in base n_0, where n_0 is the length of the list of variables of non-primitive type with greatest length (occurring in a given program).

For any decimal *number* i (in the meta-language), $\underline{i}$ is used to denote a corresponding *constant* term (in the object language). If $\underline{i}$ is an index in an indexed term, then it is written in base n_0.

The next two sections define respectively the syntax and semantics of **L1**.

3.1 Syntax

This section defines the syntax of terms and formulas of **L1** and the syntax of *correctness formulas*.

Terms and formulas are the phrase types of **L1**. Correctness formulas include terms and formulas of **L1** and commands of the programming language **A1**.

We use below the following corresponding syntactic meta-variables and phrase types:

Meta-variable	Phrase type	
e	E	expressions
ν	Lv	logical variables
f	F	function symbols
t	T	terms
p, q	P	formulas

The following syntactic meta-variables are also used (for which no phrase type is explicitly defined): τ ranges over types, ρ ranges over primitive types and ϱ ranges over non-primitive types.

We write:

1. $\nu \circ Lv[\tau]$ if ν has type τ
2. $f \circ F[\rho, \tau_1, \ldots, \tau_n]$ if f has type $(\rho, \tau_1, \ldots, \tau_n)$

The type of an indexed term is given by a function $Ti : Odn \times \mathbf{Z} \to Odn$ defined recursively as follows. In this definition, if τ is a non-primitive type let n_τ be the length of the list of variables of τ, and $\tau_1, \ldots, \tau_n$ be, respectively, the types of the variables in this list. We have:

$$Ti(\tau, i) = \begin{cases} \tau & \text{if } \tau \text{ is a primitive type, or} \\ & \tau \text{ is a non-primitive type and } (i < 1 \text{ or } n_\tau < i < n_0) \\ \tau_i & \text{if } \tau \text{ is a non-primitive type and } 1 \leq i \leq n_\tau \\ Ti(\tau_{hd(i)}, tl(i)) & \text{if } \tau \text{ is a non-primitive type and } i \geq n_0 \end{cases}$$

where $hd, tl : \mathbf{Z} \to \mathbf{Z}$ are as follows:

$$hd(i) = \begin{cases} i \,\mathtt{div}\, n_0 & \text{if } i \,\mathtt{div}\, n_0 < n_0 \\ hd(i \,\mathtt{div}\, n_0) & \text{otherwise} \end{cases}$$

$$tl(i) = \begin{cases} i \,\mathtt{mod}\, n_0 & \text{if } i \,\mathtt{div}\, n_0 < n_0 \\ tl(i \,\mathtt{div}\, n_0) * n_0 + i \,\mathtt{mod}\, n_0 & \text{otherwise} \end{cases}$$

The set T of well-formed terms is then (inductively) defined by:

$$\frac{e \circ E[\tau]}{e \circ T[\tau]}$$

$$\frac{\nu \circ Lv[\tau]}{\nu \circ T[\tau]}$$

$$\frac{t_0 \circ T[\mathtt{Bool}] \quad t_1 \circ T[\tau] \quad t_2 \circ T[\tau]}{\mathtt{if}\ t_0\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 \circ T[\tau]}$$

$$\frac{t \circ T[\tau] \quad i \circ T[\mathtt{Z}]}{t[i] \circ T}$$

where i is not a constant.[1]

$$\frac{t \circ T[\varrho]}{t[\underline{i}] \circ T[Ti(\varrho, i)]}$$

$$\frac{t_0 \circ T[\varrho] \quad t_1 \circ T[\tau_1] \quad \ldots \quad t_n \circ T[\tau_n]}{<t_0 : t_1, \ldots, t_n> \circ T[\varrho]}$$

where $\tau_1, \ldots, \tau_n$ are the types of the variables in the list of variables of ϱ, respectively.

$$\frac{f \circ F[\rho, \tau_1, \ldots, \tau_n] \quad t_1 \circ T[\tau_1] \quad \ldots \quad t_n \circ T[\tau_n]}{f(t_1, \ldots, t_n) \circ T[\rho]}$$

$$\frac{t_0 \circ T[\varrho_0] \quad t_s \circ T[\varrho] \quad t \circ T[\tau_i]}{t_0[t_s, \underline{i}, t] \circ T[\varrho_0]}$$

where τ_i is the type of the i-th variable in the list of variables of ϱ.

The set P of *formulas* is (inductively) defined by:

$$\frac{t_1 \circ T \quad t_2 \circ T}{t_1 \doteq t_2 \circ P}$$

$$\frac{t \circ T[\texttt{Bool}]}{t \circ P}$$

$$\frac{p \circ P}{\neg p \circ P}$$

$$\frac{p \circ P \quad \nu \circ Lv[\tau]}{\forall \nu : p \circ P}$$

$$\frac{p_1 \circ P \quad p_2 \circ P}{p_1 \wedge p_2 \circ P}$$

Formulas using logical connectives other than $\wedge, \neg, \forall$ can be seen as abbreviations of formulas using one or more of these symbols.

We use the same equality symbol $\doteq$ for expressing equality between (programming language) expressions and between terms.

The set $\mathcal{W}$ of *correctness formulas* is defined by:

$$\frac{p \circ P \quad c \circ C \quad q \circ P}{\{ p \} \, c \, \{ q \} \circ \mathcal{W}}$$

$$\frac{p \circ P \quad c \circ C[\tau] \quad t \circ T[\tau] \quad q \circ P}{\{ p \} \, c \, \{ q, t \} \circ \mathcal{W}}$$

where $\tau \not\equiv \epsilon$

Formula p is called the *precondition*, c the command, q the *postcondition* and t the *result value* of a correctness formula $\{ p \} \, c \, \{ q, t \}$ (and similarly for $\{ p \} \, c \, \{ q \}$).

[1] Constant indices are written in base n_0 (see below). Terms which have no type can be used in formulas only in equalities (see the syntax of formulas below).

3.2 Semantics

This section defines the semantics of **L1** and the semantics of correctness formulas.

The semantics of **L1** is defined in relation to an interpretation $\widehat{\mathbf{I}}$ which is a pair $(\widehat{D}, \widehat{\mathcal{I}})$. $\widehat{D}$ is the *semantic domain* of the interpretation $\widehat{\mathbf{I}}$, and $\widehat{\mathcal{I}}$ is the *interpretation function* for constant and function symbols, mapping them to elements and functions over $\widehat{D}$, respectively.

3.2.1 Semantic domain

The semantic domain $\widehat{D}$ contains values called *object structures*, defined below.

Definition 9 $\hat{d}$ is an object structure if $\hat{d} \in \Theta$ or $\hat{d}$ is a finite sequence $(a, \hat{d}_1, \ldots, \hat{d}_n)$, where $n > 0$, $a \in A$ and $\hat{d}_1, \ldots, \hat{d}_n$ are object structures.

When $\hat{d}$ is a non-primitive object or a sequence $(a, \hat{d}_1, \ldots, \hat{d}_n)$ (where $a \in A$ and $n > 0$), we say that $\hat{d}$ is an object structure rooted at a.

Equality between object structures is defined as equality between elements and between finite sequences, and by defining no element as equal to any sequence.

The semantic domain $\widehat{D}$ contains also members of the domain D (defined in Section 2.2.1) which are here called *values of primitive type*:

Definition 10 $\hat{d}$ is a value of primitive type if $\hat{d}$ is a primitive object — i.e. $\hat{d} \in \mathbf{Z}$, $\hat{d} \in \mathbf{B}$ or $\hat{d} = nil$ — or $\hat{d}$ is a set, sequence or cartesian product of values of primitive type.

Values of primitive type are denotations for corresponding constant terms of primitive type in the specification language (see Section 3.1).

Note that if $\hat{d} \in \mathbf{Z}$, $\hat{d} \in \mathbf{B}$ or $\hat{d} = nil$, then $\hat{d}$ is both an object structure and a value of primitive type.

3.2.2 Semantics of terms and formulas

L1 being a language of formal logic for reasoning about **A1** programs, the interpretation $\widehat{\mathbf{I}}$ is related to the interpretation $\mathbf{I}$ defined in Section 2.2 as follows. D is a subset of $\widehat{D}$ and, furthermore, all values in $\widehat{D}$ which are not in D (i.e. finite sequences containing non-primitive objects) are constructed from values in D (see Definition 9). The interpretation function $\widehat{\mathcal{I}}$ is an extension of $\mathcal{I}$ (or, equivalently, $\mathcal{I}$ is a restriction of $\widehat{\mathcal{I}}$ to the set of predefined constant and function symbols of **A1**). That is, if f is a constant or function symbol such that $f \in \mathtt{dom}(\mathcal{I})$, then $\widehat{\mathcal{I}}(f) = \mathcal{I}(f)$.

We consider assignments $\hat{s} : (X_{\hat{s}} \cup \{\mathtt{self}\} \cup Lv_s) \to \Theta$, where $X_{\hat{s}}$ is as in Section 2.2 a given finite set of program variables or formal parameters, and Lv_s is a finite subset of the set Lv of logical variables.

Assignments s and $\hat{s}$, which give values to variables in a correctness formula, for interpretations $\mathbf{I}$ of **A1** and $\widehat{\mathbf{I}}$ of **L1**, respectively, are both relative to a given state and a non-primitive object.

Given a state σ and a non-primitive object α, an assignment s for $\mathbf{I}$ is said to be relative to that state and object — and is more precisely denoted by $s_{\sigma,\alpha}$ — if $s = \sigma(\alpha)$ (cf. Definition 7, first bullet, Section 2.2.2).

Given a state σ and a non-primitive object α, an assignment $\hat{s}$ for $\widehat{\mathbf{I}}$ is said to be relative to that state and object — and is more precisely denoted by $\hat{s}_{\sigma,\alpha}$ — if $\hat{s}(x) = \sigma(\alpha)(x)$, for all $x \in \mathtt{dom}\big(\sigma(\alpha)\big)$ (i.e. $\hat{s}_{\sigma,\alpha}$ is an extension of $s_{\sigma,\alpha}$). The restriction of $\hat{s}_{\sigma,\alpha}$ to the domain of $s_{\sigma,\alpha}$ is denoted by $\hat{s}_{\sigma,\alpha} \restriction \mathtt{dom}\big(\sigma(\alpha)\big)$.

Any given assignments $\hat{s}_{\sigma,\alpha}$ and $\hat{s}_{\sigma',\alpha'}$, i.e. those denoted by the same "$\hat{s}$" (also, for example, $\hat{s}'_{\sigma,\alpha}$ and $\hat{s}'_{\sigma',\alpha'}$ etc.) — relative to states σ and σ' and non-primitive objects α and α', respectively — are such that $\mathtt{dom}(\hat{s}_{\sigma,\alpha}) = \mathtt{dom}(\hat{s}_{\sigma',\alpha'})$ (i.e. their domains are the same) and $\hat{s}_{\sigma,\alpha}(x) = \hat{s}_{\sigma',\alpha'}(x)$ for all $x \in \mathtt{dom}(\hat{s}_{\sigma,\alpha}) - \mathtt{dom}\big(\sigma(\alpha)\big)$. Of course, we can and usually have $\mathtt{dom}(\hat{s}_{\sigma,\alpha}) \neq \mathtt{dom}(\hat{s}'_{\sigma',\alpha'})$, for different $\hat{s}$ and $\hat{s}'$.

Arguments of functions of non-primitive type are given expanded interpretations which are sequences upon which the function is evaluated.

Terms

The meaning of a term depends on pairs $u = (\sigma, \hat{s}_{\sigma,\alpha})$.[2] We denote by U the set of all such pairs.

The meaning of a term t is given by a function:

$$\widehat{\mathbf{I}}[\![t]\!] : U \to \widehat{D}_\varepsilon$$

defined by induction over the structure of t:

- if t is an expression (of the programming language), then:[3]

$$\widehat{\mathbf{I}}[\![t]\!](\sigma, \hat{s}_{\sigma,\alpha}) = \mathbf{I}[\![t]\!]\Big(\hat{s}_{\sigma,\alpha} \restriction \mathtt{dom}\big(\sigma(\alpha)\big)\Big)$$

- if t is a logical variable or a program variable such that $t \notin \mathtt{dom}\big(\sigma(\alpha)\big)$, then:

$$\widehat{\mathbf{I}}[\![t]\!](\sigma, \hat{s}_{\sigma,\alpha}) = \hat{s}_{\sigma,\alpha}(t)$$

- if $t \equiv \mathtt{if}\, t_0 \,\mathtt{then}\, t_1 \,\mathtt{else}\, t_2$, then the semantics is analogous to that of an expression with the same ($\mathtt{if\,then\,else}$) operator (see Definition 8, Section 2.2.2);

- if $t \equiv \,<t_0 : t_1, \ldots, t_n>$, then, letting $\hat{d} = \big(\widehat{\mathbf{I}}[\![t_0]\!](u) \downarrow 1, \widehat{\mathbf{I}}[\![t_1]\!](u), \ldots, \widehat{\mathbf{I}}[\![t_n]\!](u)\big)$, we have:

$$\widehat{\mathbf{I}}[\![t]\!](u) = \begin{cases} nil & \text{if } \widehat{\mathbf{I}}[\![t_0]\!](u) = nil \\ \widehat{\mathbf{I}}[\![t_0]\!](u) \downarrow 1 & \text{if } \widehat{\mathbf{I}}[\![t_0]\!](u) \neq nil \text{ and } \widehat{\mathbf{I}}[\![t_i]\!](u) = \widehat{\mathbf{I}}[\![t_0[\underline{i}]]\!](u), \\ & \text{for all } i = 1, \ldots, n \\ \hat{d} & \text{otherwise} \end{cases}$$

- if $t \equiv t_1[k]$, consider:
  - $\widehat{\mathbf{I}}[\![t_1]\!](\sigma, \hat{s}) = \varepsilon$; then:

  $$\widehat{\mathbf{I}}[\![t]\!](\sigma, \hat{s}) = \varepsilon$$

  - $\widehat{\mathbf{I}}[\![t_1]\!](\sigma, \hat{s}) = nil$; then:

  $$\widehat{\mathbf{I}}[\![t]\!](\sigma, \hat{s}) = error$$

  - otherwise, let τ be the type of t_1; if τ is a primitive type, let $n = 0$ and otherwise let $n \geq 0$ and $v_1, \ldots, v_n$ be the list of variables of τ; let $k \equiv \underline{i}$, if k is a constant term[4], and $\widehat{\mathbf{I}}[\![k]\!](\sigma, \hat{s}) = i$, otherwise; let $t_1' \equiv (t_1[k \,\mathtt{div}\, \underline{n}_0])[k \,\mathtt{mod}\, \underline{n}_0]$; we have:

  $$\widehat{\mathbf{I}}[\![t]\!](\sigma, \hat{s}) = \begin{cases} \widehat{\mathbf{I}}[\![t_1]\!](\sigma, \hat{s}) \downarrow (i+1) & \text{if } i \leq n \text{ and, for some } a \in A_\tau, \\ & \text{ and some } \hat{d}_1, \ldots, \hat{d}_n \in \widehat{D}, \\ & \widehat{\mathbf{I}}[\![t_1]\!](\sigma, \hat{s}) = (a, \hat{d}_1, \ldots, \hat{d}_n) \\ \sigma(a)(v_i) & \text{if } i \leq n \text{ and, for some } a \in A_\tau, \\ & \widehat{\mathbf{I}}[\![t_1]\!](\sigma, \hat{s}) = a \\ error & \text{if } n < i \leq n_0 \text{ or } i \leq 0 \\ \widehat{\mathbf{I}}[\![t_1']\!](\sigma, \hat{s}) & \text{if } i > n_0 \end{cases}$$

---

[2]That is, it depends on pairs $u = (\sigma, \hat{s})$ where $\hat{s}$ is an assignment for $\widehat{\mathbf{I}}$ relative to σ and some non-primitive object α.

[3]The induction could of course be extended further so as to give the meaning of terms which are expressions only in terms of values given by $\hat{s}_{\sigma,\alpha}$. The definition given uses the fact that for all variables x occurring in an expression, $x \in \mathtt{dom}\big(\sigma(\alpha)\big)$ (see Section 2.2.2).

[4]A term t is *constant* if $Fn(t) = \emptyset$ ($Fn(t)$ is as in Definition 11).

- if $t \equiv f(t_1, \ldots, t_n)$, then:

$$\widehat{\mathbf{I}}[\![t]\!](u) = \widehat{\mathcal{I}}(f)\big(\widehat{\mathbf{I}}_{\mathbf{x}}[\![t_1]\!](u), \ldots, \widehat{\mathbf{I}}_{\mathbf{x}}[\![t_n]\!](u)\big)$$

The function $\widehat{\mathbf{I}}_{\mathbf{x}}[\![t]\!] : U \to \widehat{D}_\varepsilon$, which gives expanded interpretations for terms of non-primitive type, is defined by:

$$\widehat{\mathbf{I}}_{\mathbf{x}}[\![t]\!](u) = \mu_x[\![t]\!](\emptyset)(u)$$

The function $\mu_x[\![t]\!] : \wp(A) \to U \to \widehat{D}_\varepsilon$ is defined as follows. If t has a non-primitive type, let n be the length of the list of variables of the type of t. Consider:

1. $\widehat{\mathbf{I}}[\![t]\!](u) = \hat{d}$, for some $\hat{d} \in \widehat{D}_\varepsilon$ such that $\hat{d} \notin A$ or $\hat{d} \in sa$; then:

$$\mu_x[\![t]\!](sa)(u) = \hat{d}$$

2. otherwise, let $\widehat{\mathbf{I}}[\![t]\!](u) = \alpha$, for some $\alpha \in A$; then:[5]

$$\mu_x[\![t]\!](sa)(u) = \big(\alpha, \mu_x[\![t[1]]\!](sa \cup \{\alpha\})(u), \ldots, \mu_x[\![t[\underline{n}]]\!](sa \cup \{\alpha\})(u)\big)$$

- if $t \equiv t_0[t_1, \underline{i}, t']$, then, letting v_i be the i-th variable in the list of variables of the type of t_1, and:

$$\widehat{\mathbf{I}}[\![t_1]\!](\sigma, \hat{s}) = \alpha$$

$$\sigma' = \sigma \dagger \big\{\alpha \mapsto \sigma(\alpha) \dagger \{v_i \mapsto \widehat{\mathbf{I}}[\![t']\!](\sigma, \hat{s}_{\sigma,\alpha})\}\big\}$$

we have:

$$\widehat{\mathbf{I}}[\![t]\!](\sigma, \hat{s}_{\sigma,\alpha}) = \widehat{\mathbf{I}}[\![t_0]\!]\big(\sigma', \hat{s}_{\sigma',\alpha}\big)$$

Formulas

The meaning of a formula p is given by a function:

$$\widehat{\mathbf{I}}[\![p]\!] : U \to \mathbf{B}$$

defined as follows:

$$\widehat{\mathbf{I}}[\![p]\!](u) = \begin{cases} \mu[\![p]\!](u) & \text{if } \mu[\![p]\!](u) \neq \varepsilon \\ \mathit{false} & \text{otherwise} \end{cases}$$

where $\mu[\![p]\!] : U \to \mathbf{B}_\varepsilon$ is defined by induction over the structure of p as follows:[6]

- if $p \equiv t_1 \doteq t_2$, then

$$\mu[\![p]\!](u) = \begin{cases} \widehat{\mathbf{I}}[\![t_1]\!](u) = \widehat{\mathbf{I}}[\![t_2]\!](u) & \text{if } \widehat{\mathbf{I}}[\![t_1]\!](u) \neq \varepsilon \\ & \text{or } \widehat{\mathbf{I}}[\![t_2]\!](u) \neq \varepsilon \\ \varepsilon & \text{otherwise} \end{cases}$$

[5] This definition of $\mu_x[\![t]\!](sa)(u)$ does not give "efficient" (with respect to the length of sequences) expanded interpretations: it does not consider in sa objects of object structures expanded for the interpretation of (say) $t[i]$ $(1 \leq i < n)$ for the interpretation of $t[i+k]$ $(i < i+k \leq n)$. This may cause the same object structure to be "written" more than once.

[6] μ is used so that the meaning $\widehat{\mathbf{I}}[\![p]\!](u)$ can be set to false if $\mu[\![p]\!](u) = \varepsilon$.

- if p is a term t which is a boolean expression, then

$$\mu[\![p]\!](u) = \widehat{\mathbf{I}}[\![t]\!](u)$$

- if $p \equiv \neg p_1$, then

$$\mu[\![p]\!](u) = \begin{cases} \big(\widehat{\mathbf{I}}[\![\neg]\!](\mu[\![p_1]\!])\big)(u) & \text{if } \mu[\![p_1]\!](u) \neq \varepsilon \\ \varepsilon & \text{otherwise} \end{cases}$$

where $\widehat{\mathbf{I}}[\![\neg]\!] : (U \to \mathbf{B}) \to U \to \mathbf{B}$ is the function defined by:

$$\widehat{\mathbf{I}}[\![\neg]\!](f)(u) = f_{\neg}\big(f(u)\big)$$

where $f_{\neg} : \mathbf{B} \to \mathbf{B}$ is the function given by:

$$f_{\neg} = \{(true, false), (false, true)\}$$

- if $p \equiv p_1 \wedge\ p_2$, then

$$\mu[\![p]\!](u) = \begin{cases} \big(\widehat{\mathbf{I}}[\![\wedge]\!](\mu[\![p_1]\!], \mu[\![p_2]\!])\big)(u) & \text{if } \mu[\![p_1]\!](u) \neq \varepsilon \\ & \text{and } \mu[\![p_2]\!](u) \neq \varepsilon \\ \varepsilon & \text{otherwise} \end{cases}$$

where $\widehat{\mathbf{I}}[\![\wedge]\!] : ((U \to \mathbf{B}) \times (U \to \mathbf{B})) \to U \to \mathbf{B}$ is the function defined by:

$$\widehat{\mathbf{I}}[\![\wedge]\!](f, g)(u) = f_{\wedge}\big(f(u), g(u)\big)$$

where $f_{\wedge} : (\mathbf{B} \times \mathbf{B}) \to \mathbf{B}$ is the function given by:

$$\begin{array}{lll} f_{\wedge} = & \{ & \big((true, true), true\big), \\ & & \big((true, false), false\big), \\ & & \big((false, true), false\big), \\ & & \big((false, false), false\big) \\ & \} & \end{array}$$

- if $p \equiv \forall \nu : p_1$, then

$$\mu[\![p]\!](\sigma, \hat{s}) = \begin{cases} \big(\widehat{\mathbf{I}}[\![\forall \nu]\!](\mu[\![p_1]\!])\big)(\sigma, \hat{s}) & \text{if } \mu[\![p_1]\!](\sigma, \hat{s} \dagger \{\nu \mapsto \hat{d}\}) \neq \varepsilon, \\ & \text{for all } \hat{d} \in \widehat{D} \\ \varepsilon & \text{otherwise} \end{cases}$$

where $\widehat{\mathbf{I}}[\![\forall \nu]\!] : (U \to \mathbf{B}) \to U \to \mathbf{B}$ is the function defined by:

$$\widehat{\mathbf{I}}[\![\forall \nu]\!](f)(u) = \nu_{\forall}(f, u)$$

where $\nu_{\forall} : (U \to \mathbf{B}) \to U \to \mathbf{B}$ is the function given by:

$$\nu_{\forall}\big(f, (\sigma, \hat{s})\big) = \begin{cases} true & \text{if } f(\sigma, \hat{s} \dagger \{\nu \mapsto \hat{d}\}) = true, \text{ for all } \hat{d} \in \widehat{D} \\ false & \text{otherwise} \end{cases}$$

3.2.3 Correctness Formulas

This section defines the so-called *partial correctness* semantics for correctness formulas.[7]

The meaning of a correctness formula is given in relation to interpretation $\widehat{\mathbf{I}}$, as follows. Note that interpretation $\mathbf{I}$ is related to $\widehat{\mathbf{I}}$ as defined in Section 3.2.2.

Consider c a command of **A1** of type τ occurring in an object definition associated to ϱ. Let p and q be formulas of **L1**, and t be a term of **L1** of type τ.

The meaning $\widehat{\mathbf{I}}[\![w]\!] : \Gamma \to U \to \mathbf{B}$ of a correctness formula $w \equiv \{\, p \,\}\; c \;\{\, q, t \,\}$ is defined as follows. For all function assignments γ, states $\sigma \in \Sigma$, non-primitive objects $\alpha \in A_\varrho \cap \mathtt{dom}(\sigma)$ and all assignments $\hat{s}_{\sigma,\alpha} \in \widehat{S}$ for $\widehat{\mathbf{I}}$, relative to state σ and non-primitive object α, we have:

$$\widehat{\mathbf{I}}[\![w]\!](\gamma)(\sigma, \hat{s}_{\sigma,\alpha}) = \mathit{true} \text{ if and only if: } \begin{cases} \text{if } \widehat{\mathbf{I}}[\![p]\!](\sigma, \hat{s}_{\sigma,\alpha}) = \mathit{true} \text{ and} \\ \text{if } \mathbf{I}[\![c]\!](\alpha)(\gamma)(\sigma) = (\sigma', \theta) \\ \text{then } \widehat{\mathbf{I}}[\![q]\!](\sigma', \hat{s}_{\sigma',\alpha}) = \mathit{true} \\ \text{and } \widehat{\mathbf{I}}[\![t]\!](\sigma', \hat{s}_{\sigma',\alpha}) = \theta \end{cases}$$

The meaning $\widehat{\mathbf{I}}[\![w]\!] : \Gamma \to U \to \mathbf{B}$ of a correctness formula $w \equiv \{\, p \,\}\; c \;\{\, q \,\}$, where c is a command of **A1** occurring in an object definition associated to ϱ, and p, q are formulas, is defined as follows. For all $\gamma \in \Gamma$, $\sigma \in \Sigma$, $\alpha \in A_\varrho \cap \mathtt{dom}(\sigma)$ and $\hat{s}_{\sigma,\alpha}$, we have:

$$\widehat{\mathbf{I}}[\![w]\!](\gamma)(\sigma, \hat{s}_{\sigma,\alpha}) = \mathit{true} \text{ if and only if: } \begin{cases} \text{if } \widehat{\mathbf{I}}[\![p]\!](\sigma, \hat{s}_{\sigma,\alpha}) = \mathit{true} \text{ and} \\ \text{if } \mathbf{I}[\![c]\!](\alpha)(\gamma)(\sigma) \downarrow 1 = \sigma' \\ \text{then } \widehat{\mathbf{I}}[\![q]\!](\sigma', \hat{s}_{\sigma',\alpha}) = \mathit{true} \end{cases}$$

[7] For total correctness, see Section 7.1.

Chapter 4

The Proof System

This chapter presents a proof system called **P1** for proving partial correctness of programs in **A1**.

Before giving the proof rules, we need first to introduce some definitions and operations based on the syntax of terms and formulas.

4.1 Free Names

Free names of terms and formulas are defined as follows.

Definition 11 (Free names of terms) Given a term t, the set $Fn(t)$ of *free names* of t is defined by induction on the structure of t as follows:

1. if $t \equiv x$, where x is a (program or logical) variable, a formal parameter or `self`, then $Fn(t) = \{x\}$;
2. if t is `false`, `true`, `nil` or an integer constant then $Fn(t) = \emptyset$;
3. if $t \equiv e_1 \doteq e_2$ then $Fn(t) = Fn(e_1) \cup Fn(e_2)$;
4. if $t \equiv \texttt{if}\, t_0\, \texttt{then}\, t_1\, \texttt{else}\, t_2$, then $Fn(t) = Fn(t_0) \cup Fn(t_1) \cup Fn(t_2)$;
5. if $t \equiv e_0.m(e_1, \ldots, e_n)$, then $Fn(t) = Fn(e_0) \cup Fn(e_1) \cup \ldots \cup Fn(e_n)$;
6. if $t \equiv t'[i]$, then $Fn(t) = Fn(t') \cup Fn(i)$;
7. if $t \equiv \,<t_0 : t_1, \ldots, t_n>$, then $Fn(t) = Fn(t_0) \cup Fn(t_1) \cup \ldots \cup Fn(t_n)$;
8. if $t \equiv f(t_1, \ldots, t_n)$, then $Fn(t) = Fn(t_1) \cup \ldots \cup Fn(t_n)$;
9. if $t \equiv t_0[t_s, i, t']$, then $Fn(t) = Fn(t_0) \cup Fn(t_s) \cup Fn(t')$.

Definition 12 (Free names of formulas) For a formula p, the set $Fn(p)$ is defined inductively as follows:

1. if $p \equiv t_1 \doteq t_2$, then $Fn(p) = Fn(t_1) \cup Fn(t_2)$;
2. if $p \equiv t$, where $t \circ T[\texttt{Bool}]$, then $Fn(p) = Fn(t)$;
3. if $p \equiv \neg p'$, then $Fn(p) = Fn(p')$;
4. if $p \equiv \forall \nu : p'$, then $Fn(p) = Fn(p') - \{\nu\}$;
5. if $p \equiv p_1 \wedge p_2$, then $Fn(p) = Fn(p_1) \cup Fn(p_2)$.

The notation $Fn(p_1, \ldots, p_n, t_1, \ldots, t_n)$ is used to abbreviate $Fn(p_1) \cup \ldots \cup Fn(p_n) \cup Fn(t_1) \cup \ldots \cup Fn(t_n)$.

We also define, for a term or formula w, the set $Fv(w)$ of *free variables* of w as $Fv(w) = Fn(w) \cap Lv$.

4.2 Textual Substitution

The *textual substitution* of a term for a free name is defined below.

Definition 13 (Textual substitution in terms) Let t and t' be terms. The result of the textual substitution of t' for x in t, where x is a (program or logical) variable, a formal parameter or `self`, denoted by $t^x_{t'}$, is defined inductively as follows:

1. if $t \equiv x$, then $t^x_{t'} \equiv t'$;
2. if $t \equiv x'$ (where $x' \not\equiv x$ and x' is a variable, a formal parameter or `self`), then $t^x_{t'} \equiv t$;
3. if t is either `true`, `false`, `nil` or an integer constant, then $t^x_{t'} \equiv t$;
4. if $t \equiv e_1 \doteq e_2$ then $t^x_{t'} \equiv (e_1)^x_{t'} \doteq (e_2)^x_{t'}$;
5. if $t \equiv \texttt{if}\ t_0\ \texttt{then}\ t_1\ \texttt{else}\ t_2$, then $t^x_{t'} \equiv \texttt{if}\ (t_0)^x_{t'}\ \texttt{then}\ (t_1)^x_{t'}\ \texttt{else}\ (t_2)^x_{t'}$;
6. if $t \equiv e_0.m(e_1, \ldots, e_n)$, then $t^x_{t'} = (e_0)^x_{t'}\,.m\big((e_1)^x_{t'}, \ldots, (e_n)^x_{t'}\big)$;
7. if $t \equiv t_1[i]$, then $t^x_{t'} \equiv (t_1)^x_{t'}[i^x_{t'}]$;
8. if $t \equiv\ < t_0 : t_1, \ldots, t_n >$, then $t^x_{t'} \equiv\ < (t_0)^x_{t'} : (t_1)^x_{t'}, \ldots (t_n)^x_{t'} >$;
9. if $t \equiv f(t_1, \ldots, t_n)$, then $t^x_{t'} \equiv f\big((t_1)^x_{t'}, \ldots, (t_n)^x_{t'}\big)$;
10. if $t \equiv t_0[t_s, i, t_1]$, then $t^x_{t'} \equiv (t_0)^x_{t'}[(t_s)^x_{t'}, i, (t_1)^x_{t'}]$.

Definition 14 (Textual substitution in formulas) For a formula p, we define p^x_t by using the principle that a bound variable can be renamed, and we do so until we get a formula p' equivalent to p for which no variable in t occurs bound in p'. Using this procedure, we can assume p is so renamed if necessary and p^x_t can be then defined inductively as follows:

1. if $p \equiv t_1 \doteq t_2$, then $p^x_t \equiv (t_1)^x_t \doteq (t_2)^x_t$;
2. if $p \equiv t$, where $t \circ T[\texttt{Bool}]$, then $p^x_t \equiv t^x_t$;
3. if $p \equiv \neg p'$, then $p^x_t \equiv \neg (p')^x_t$;
4. if $p \equiv \forall \nu : p'$, then $p^x_t \equiv \left\{ \begin{array}{ll} \forall \nu : p' & \text{if } x \equiv \nu \\ \forall \nu : (p')^x_t & \text{otherwise} \end{array} \right\}$;
5. if $p \equiv p_1 \wedge p_2$, then $p^x_t \equiv (p_1)^x_t \wedge (p_2)^x_t$.

The definition of textual substitution can be generalized to more than one variable and term. If $x_1, \ldots, x_n$ are distinct program variables or `self`, and $t_1, \ldots, t_n$ are terms, and h is a term or a formula, then $h^{x_1,\ldots,x_n}_{t_1,\ldots,t_n}$ denotes the simultaneous substitution of $x_1, \ldots, x_n$ by the terms $t_1, \ldots, t_n$, respectively.

The notation $\bar{x}$ (or $\bar{t}$) is generally used to denote a specified list of variables (or terms).[1]

The textual substitution notation for more than one variable can be used also for the case when the variables are not all distinct. If $x_i \equiv x_j$, for any $1 \leq i < j \leq n$, then $h^{x_1,\ldots,x_i,\ldots,x_{j-1},x_j,x_{j+1},\ldots,x_n}_{t_1,\ldots,t_i,\ldots,t_{j-1},t_j,t_{j+1},\ldots,t_n}$ is (syntactically) equivalent to $h^{x_1,\ldots,x_i,\ldots,x_{j-1},x_{j+1},\ldots,x_n}_{t_1,\ldots,t_i,\ldots,t_{j-1},t_{j+1},\ldots,t_n}$.

[1] We thus usually let $\bar{x} \equiv x_1, \ldots, x_n$ and $\bar{t} \equiv t_1, \ldots, t_n$ and write $h^{\bar{x}}_{\bar{t}}$ as an abbreviated form of $h^{x_1,\ldots,x_n}_{t_1,\ldots,t_n}$.

4.3 Substitution

The motivations behind a textual substitution are those of instantiating logical variables and renaming, due to a "change of scope" as occurring in a message sending command. Substitution as defined in this section is used when there is a change of interpretation, which occurs in the case of an assignment command. This substitution deals specifically with two cases for which textual substitution is not adequate when there is such a change of interpretation:

1. Aliasing: An indexed term may have always the same interpretation as an object variable. If the object variable changes, such an indexed term must also change accordingly.

2. Arguments of functions refering to the current object: Arguments of functions have expanded interpretations. Considering x as an object variable, the interpretation of a term (for example, the interpretation of another object variable, say y) which denotes an argument of a function changes if this term (y) defines an object structure in which the current object occurs and the value assigned to x changes.

In the first case, according to the types of terms and values of indices, a conditional expression may be required to test whether a term t in $t[i]$ denotes the current object or not.[2] In the second case, suppose x is assigned a value denoted by a term t, and suppose that y, which denotes an argument of a function, can (in syntactic terms) refer to the current object.[3] The effect of the substitution of x for t in y, denoted in the meta-language by $y[t/x]$, is the term $y[\texttt{self}, i, t]$, where i is the index of x in the list of variables of the type of `self`. Axiom (xxvi) (see Section 4.5) can then be used to transform formulas valid for y into formulas valid for $y[\texttt{self}, i, t]$.

In the following we make use of function $typ : Odn \to \wp(Odn)$, which defines the non-primitive types transitively referred to by object variables of a given non-primitive type; it is given by:

$$typ(\tau) = types(\tau, \emptyset)$$

where $types : Odn \times \wp(Odn) \to \wp(Odn)$ is given by defining $types(\tau, st)$ as equal to:

- $\emptyset$, if $\tau \in st$ or τ is a primitive type;
- $\{\tau\} \cup types(\tau_1, st \cup \{\tau\}) \cup \ldots \cup types(\tau_n, st \cup \{\tau\})$, otherwise

where $\tau_1, \ldots, \tau_n$ are the types of the list of variables of τ (in case τ is a non-primitive type).

We define next the syntactic operation of substitution.

If x is a temporary variable, then for any formula p and any term t, the substitution $p[t/x]$ is defined as equivalent to the textual substitution p^x_t. Thus, in the following we consider x as an object variable.

Definition 15 (Substitution in terms) Let x be the j-th variable in the list of variables of ϱ. For any terms t_0, t and any object variable x, $t_0[t/x]$ is defined by induction on the structure of t_0 as follows:

1. if $t_0 \equiv x$, then $t_0[t/x] \equiv t$;

2. if $t_0 \equiv x'$ (where $x' \not\equiv x$ and x' is a variable or a formal parameter), then $t_0[t/x] \equiv t_0$;

3. if t_0 is either $\texttt{true}, \texttt{false}, \texttt{nil}, \texttt{self}$ or an integer constant, then $t_0[t/x] \equiv t_0$;

4. if $t_0 \equiv e_1 \doteq e_2$ then $t_0[t/x] \equiv e_1[t/x] \doteq e_2[t/x]$;

5. if $t_0 \equiv \texttt{if}\, t' \,\texttt{then}\, t_1 \,\texttt{else}\, t_2$, then $t_0[t/x] \equiv \texttt{if}\, t'[t/x] \,\texttt{then}\, t_1[t/x] \,\texttt{else}\, t_2[t/x]$;

6. if $t_0 \equiv e_0.m(e_1, \ldots, e_n)$, then $t_0[t/x] = e_0[t/x].m(e_1[t/x], \ldots, e_n[t/x])$;

[2] This approach is basically the same used in [AdB90a, Section 5.1.2].

[3] Note that in the case of object variables of the current object, refering to the current object implies denoting a cyclic object structure.

7. if $t_0 \equiv t_1[k]$, then if $t_1 \equiv \mathtt{nil}$, then $t_0[t/x] \equiv t_0$; otherwise let τ_1 be the type of t_1, n be the length of the list of variables of τ_1, $t_1' \equiv t_1[t/x]$, $k' \equiv k[t/x]$ and consider:

   - k is a constant term; then, letting $k \equiv \underline{i}$, we have:
     - if $i > n_0$, then
     $$t_0[t/x] \equiv \big((t_1[k\,\mathtt{div}\,\underline{n}_0])[k\,\mathtt{mod}\,\underline{n}_0]\big)[t/x]]$$
     - if $\tau_1 \equiv \varrho$ and $i \equiv j$, then:
     $$t_0[t/x] \equiv \begin{array}{l} \mathtt{if}\, t_1' \doteq \mathtt{self} \\ \quad \mathtt{then}\, t \\ \quad \mathtt{else}\, t_1'[k] \end{array}$$
     - otherwise $t_0[t/x] \equiv t_1'[k]$;
   - otherwise (i.e. k is not a constant term), we have:
   $$t_0[t/x] \equiv \begin{cases} t_0[\mathtt{self}, \underline{j}, t] & \text{if } \mathit{typ}(\tau_1) \supseteq \{\varrho\} \\ t_1'[k'] & \text{otherwise} \end{cases}$$

8. if $t_0 \equiv <t' : t_1, \ldots, t_n>$, then $t_0[t/x] \equiv <t'[t/x] : t_1[t/x], \ldots, t_n[t/x]>$;
9. if $t_0 \equiv f(t_1, \ldots, t_n)$, then $t_0[t/x] \equiv f(t_1[x \rightsquigarrow t], \ldots, t_n[x \rightsquigarrow t])$;
10. if $t_0 \equiv t_1[t_2, \underline{i}, t']$, then $t_0[t/x] \equiv t_0[\mathtt{self}, \underline{j}, t]$.

We define next the substitution $t_0[x \rightsquigarrow t]$, applied to terms denoting arguments of functions.

Definition 16 (Substitution in arguments) Let t_0, t be terms, x the i-th object variable in the list of variables of ϱ, and ϱ_0 the type of t_0. The term denoted by $t_0[x \rightsquigarrow t]$ is defined by induction on the structure of t_0 as follows:

1. if t_0 is a variable or a formal parameter or `self`, then:
$$t_0[x \rightsquigarrow t] \equiv \begin{cases} t_0[\mathtt{self}, \underline{i}, t] & \text{if } \mathit{typ}(\varrho_0) \supseteq \{\varrho\} \\ t & \text{if } \mathit{typ}(\varrho_0) \not\supseteq \{\varrho\} \text{ and } t_0 \equiv x \\ t_0 & \text{otherwise} \end{cases}$$
2. if t_0 is either $\mathtt{true}, \mathtt{false}, \mathtt{nil}$ or an integer constant, then $t_0[x \rightsquigarrow t] \equiv t_0$;
3. if $t_0 \equiv e_1 \doteq e_2$ then $t_0[x \rightsquigarrow t] \equiv e_1[x \rightsquigarrow t] \doteq e_2[x \rightsquigarrow t]$;
4. if $t_0 \equiv \mathtt{if}\, t'\, \mathtt{then}\, t_1\, \mathtt{else}\, t_2$, then:
$$t_0[x \rightsquigarrow t] \equiv \mathtt{if}\, t'[x \rightsquigarrow t]\, \mathtt{then}\, t_1[x \rightsquigarrow t]\, \mathtt{else}\, t_2[x \rightsquigarrow t]$$
5. if $t_0 \equiv e_0.m(e_1, \ldots, e_n)$, then $t_0[x \rightsquigarrow t] \equiv e_0[x \rightsquigarrow t].m\big(e_1[x \rightsquigarrow t], \ldots, e_n[x \rightsquigarrow t]\big)$;
6. if $t_0 \equiv t_1[j]$, then the definition is analogous to that of substitution (with substitution in arguments where we have substitution).
7. if $t_0 \equiv <t' : t_1, \ldots, t_n>$, then $t_0[x \rightsquigarrow t] \equiv <t'[t/x] : t_1[x \rightsquigarrow t], \ldots t_n[x \rightsquigarrow t]>$;
8. if $t_0 \equiv f(t_1, \ldots, t_n)$, then $t_0[x \rightsquigarrow t] \equiv f(t_1[x \rightsquigarrow t], \ldots, t_n[x \rightsquigarrow t])$;
9. if $t_0 \equiv t_1[t_2, \underline{i}, t']$, then $t_0[x \rightsquigarrow t] \equiv t_0[\mathtt{self}, \underline{j}, t]$.

Definition 17 (Substitution in formulas) For a formula p and an object variable x, $p[t/x]$ is defined by induction on the structure of p analogously as for textual substitution.

4.4 Variables changed by a command

The following definition is used in the Invariance Axiom Scheme (xiv).

Definition 18 (Variables changed by a command) Consider a command c which occurs in object definition ϱ and method definition m. The set $Chg(c)$ of variables changed by c is given by:

$$Chg(c) = Change(c)(\emptyset)$$

where $Change(c) : \wp(Odn \times M) \rightharpoonup \wp(Odn \times X \times \{V, Tp\})$ is defined by induction on the structure of c as follows:

1. if $c \equiv e$ then:

$$Change(c)(sm) = \emptyset$$

2. if $c \equiv x \leftarrow c'$ then:

$$Change(c)(sm) = \begin{cases} \{(\varrho, x, V)\} \cup Change(c')(sm) & \text{if } x \in V \\ \{(\varrho, x, Tp)\} \cup Change(c')(sm) & \text{if } x \in Tp \end{cases}$$

3. if $c \equiv e_0.m_0(e_1, \ldots, e_n)$, let ϱ_0 be the type of e_0 and let c_0 be the body of m_0; then:

$$Change(c)(sm) = \begin{cases} \emptyset & \text{if } (\varrho_0, m_0) \in sm \\ Change(c_0)\big(sm \cup \{(\varrho, m)\}\big) & \text{otherwise} \end{cases}$$

4. if $c \equiv x \leftarrow odn!\mathtt{new}(e_1, \ldots, e_n)$, then:

$$Change(c)(sm) = \begin{cases} \{(\varrho, x, V)\} & \text{if } x \in V \\ \{(\varrho, x, Tp)\} & \text{if } x \in Tp \end{cases}$$

5. if $c \equiv c_1; c_2$, then:

$$Change(c)(sm) = Change(c_1)(sm) \cup Change(c_2)(sm)$$

6. if $c \equiv \mathtt{if}\, c_0\, \mathtt{then}\, c_1\, \mathtt{else}\, c_2$, then:

$$Change(c)(sm) = Change(c_0)(sm) \cup Change(c_1)(sm) \cup Change(c_2)(sm)$$

7. if $c \equiv tp_1 : odn_1, \ldots\ tp_n : odn_n\ c'$, then:

$$Change(c)(sm) = Change(c')(sm) - \{(\varrho, tp_1, Tp), \ldots, (\varrho, tp_n, Tp)\}$$

4.5 The Proof Rules

The proof rules of **P1** are presented next, using the method of *natural deduction*[Pra65]. A correctness formula w between double square brackets (i.e. $[\![w]\!]$) indicates that w is a *discharged* hypothesis.

Instantiation Rule:

$$\frac{\{\, p \,\}\, c\, \{\, q, t \,\} \quad \{\, \nu \doteq t_0 \wedge p \,\}\, c\, \{\, \nu \doteq t_0 \,\}}{\{\, p^{\nu}_{t_0} \,\}\, c\, \{\, q^{\nu}_{t_0}, t^{\nu}_{t_0} \,\}} \qquad \text{(i)}$$

Consequence Rule:

$$\frac{p \Rightarrow p' \quad \{\, p' \,\}\, c\, \{\, q', t \,\} \quad q' \Rightarrow q}{\{\, p \,\}\, c\, \{\, q, t \,\}} \tag{ii}$$

Expression Axiom scheme:

$$\{\, p \,\}\, e\, \{\, p, e \,\} \tag{iii}$$

Assignment Rule:

$$\frac{\{\, p \,\}\, c\, \{\, q[t/x], t \,\}}{\{\, p \,\}\, x \leftarrow c\, \{\, q, x \,\}} \tag{iv}$$

Message Sending: Consider ϱ a non-primitive type such that $v_1, \ldots, v_k$ is the list of variables of ϱ and such that m is a method in the set of method names of ϱ (cf. Definition 1).

Consider c_0 the body of m, $fp_1, \ldots, fp_n$ the formal parameters of m, and $c \equiv \texttt{self}.m(fp_1, \ldots, fp_n)$, where $\texttt{self}$ has type ϱ. We have:

Recursion Rule:

$$\frac{\begin{array}{c}[\![\{\, p \,\}\, c\, \{\, q, t \,\}]\!] \\ \{\, p \,\}\, c_0\, \{\, q, t \,\}\end{array}}{\{\, p \,\}\, c_0\, \{\, q, t \,\}} \tag{v}$$

Invocation Rule:

$$\frac{\{\, p \,\}\, c_0\, \{\, q, t \,\}}{\{\, p \,\}\, c\, \{\, q, t \,\}} \tag{vi}$$

Consider a message sending command $e_0.m(e_1, \ldots, e_n)$ occurring in an object definition associated with ϱ' (cf. Definition 1) such that e_0 has type ϱ, and let $v'_1, \ldots, v'_r$ be the list of variables of ϱ'. We have:

Substitution Rule:

$$\frac{\{\, p \,\}\, c\, \{\, q, t \,\}}{\{\, \bar{p} \,\}\, e_0.m(e_1, \ldots, e_n)\, \{\, \bar{q}, \bar{t} \,\}} \tag{vii}$$

where, letting h denote either q or t, formulas $p, q, \bar{p}, \bar{q}$ and terms $t, \bar{t}$ are such that:

$$p' \equiv p^{\texttt{self}, \overline{fp}, \bar{v}}_{e_0, \bar{e}, \bar{e}_0}$$

$$\bar{h} \equiv h^{\texttt{self}, \overline{fp}, \bar{v}}_{e'_0, \bar{e}', \bar{e}'_0}$$

$\bar{p} \equiv p_n$ and, for all $i = 1, \ldots, n$, we have:

$$p_i \equiv \begin{cases} p_{i-1} & \text{if } p_{i-1} \equiv e_i \doteq \nu_i \wedge p'', \text{ for some } p'', \nu_i, \text{ or} \\ & Fn(q) \cap \{fp_i\} = \emptyset, \text{ or } Fn(e_i) \cap \{v'_1, \ldots, v'_r\} = \emptyset \\ p_{i-1} \wedge e_i \doteq \nu_i & \text{otherwise, where } \nu_i \notin Fn(p_{i-1}) \end{cases}$$

$$p_0 \equiv \begin{cases} p' & \text{if } p' \equiv e_0 \doteq \nu' \wedge p'', \text{ for some } p'', \nu', \text{ or} \\ & Fn(q) \cap \{\texttt{self}\} = \emptyset, \text{ or } Fn(e_0) \cap \{v'_1, \ldots, v'_r\} = \emptyset \\ p' \wedge e_0 \doteq \nu' & \text{otherwise, where } \nu' \notin Fn(p') \end{cases}$$

$$\begin{aligned} \bar{e} &\equiv e_1, \ldots, e_n \\ \bar{e}_0 &\equiv e_0[\mathtt{1}], \ldots, e_0[\underline{k}] \\ \overline{fp} &\equiv fp_1, \ldots, fp_n \\ \bar{v} &\equiv v_1, \ldots, v_k \end{aligned}$$

$$\begin{aligned} \bar{e}'_0 &\equiv e'_0[\mathtt{1}], \ldots, e'_0[\underline{k}] \\ \bar{e}' &\equiv e'_1, \ldots, e'_n \end{aligned}$$

$$e_0' \equiv \begin{cases} e_0 & \text{if } Fn(e_0) \cap \{v_1', \ldots, v_r'\} = \emptyset \\ \nu' & \text{otherwise} \end{cases}$$

For all $i = 1, \ldots, n$, we have:

$$e_i' \equiv \begin{cases} e_i & \text{if } Fn(e_i) \cap \{v_1', \ldots, v_r'\} = \emptyset \\ \nu_i & \text{otherwise} \end{cases}$$

Invariance Rule: If $\varrho \equiv \varrho'$ then:

$$\frac{\{\, p \,\}\, c\, \{\, p \,\}}{\{\, p \,\}\, e_0.m(e_1, \ldots, e_n)\, \{\, p \,\}} \tag{viii}$$

where $Fn(p) \subset Lv \cup \{\texttt{self}\}$

Object Creation: We have the following two axiom schemes:[4]

First Axiom scheme:

$$\{\, p_0 \,\}\, x \leftarrow odn!\texttt{new}(e_1, \ldots, e_n)\, \{\, p \wedge x[\texttt{1}] \doteq (e_1)^x_{t_0} \wedge \ldots \wedge x[\underline{n}] \doteq (e_n)^x_{t_0}, x \,\} \tag{ix}$$

where: t_0 is a term such that $x \notin Fn(t_0)$, and

$$p_0 \equiv \begin{cases} \forall \nu : p[\nu/x] & \text{if } x \notin Fn(e_1, \ldots, e_n) \\ x \doteq t_0 \wedge \forall \nu : p[\nu/x] & \text{otherwise} \end{cases}$$

Second Axiom scheme:

$$\{\, \forall \nu : p[\nu/x] \,\}\, x \leftarrow odn!\texttt{new}(e_1, \ldots, e_n)\, \{\, p \wedge t \not\doteq x, x \,\} \tag{x}$$

where t is a term such that $x \notin Fn(t)$

Rule for Sequential Composition:

$$\frac{\{\, p \,\}\, c_1\, \{\, p' \,\} \quad \{\, p' \,\}\, c_2\, \{\, q, t \,\}}{\{\, p \,\}\, c_1; c_2\, \{\, q, t \,\}} \tag{xi}$$

Rule for If-Then-Else:

$$\frac{\{\, p \,\}\, c_0\, \{\, p', t \,\} \quad \{\, p' \wedge\ t \,\}\, c_1\, \{\, q \,\} \quad \{\, p' \wedge\ \neg t \,\}\, c_2\, \{\, q \,\}}{\{\, p \,\}\, \texttt{if}\ c_0\ \texttt{then}\ c_1\ \texttt{else}\ c_2\, \{\, q \,\}} \tag{xii}$$

Rule for Blocks:

$$\frac{\{\, p \wedge tp_1 \doteq \texttt{nil} \wedge \ldots \wedge tp_n \doteq \texttt{nil} \,\}\, c\, \{\, q, t \,\}}{\{\, p \,\}\, tp_1 : \tau_1, \ldots, tp_n : \tau_n\ c\, \{\, q, t \,\}} \tag{xiii}$$

where $tp_i \notin Fv(p, q, t)$, for all $i = 1, \ldots, n$

Invariance Axiom scheme:

$$\{\, p \,\}\, c\, \{\, p \,\} \tag{xiv}$$

where p is such that:

1. c occurs in an object definition ϱ such that, for all $x \in Fn(p)$, $(\varrho, x, V) \cap Chg(c) = \emptyset$, if $x \in V$, and $(\varrho, x, Tp) \cap Chg(c) = \emptyset$, if $x \in Tp$.
2. p has no indexed terms, i.e. for no terms t and i, $t[i]$ occurs in p;

[4] The two axiom schemes can be merged into a single axiom scheme for object creation; they are separated only for clarity.

3. no term $f(t_1,\ldots,t_n)$ occurs in p where $n \geq 1$ and for some i between 1 and n, t_i has a non-primitive type.

Conjunction Rule:

$$\frac{\{\,p\,\}\,c\,\{\,q,t\,\} \quad \{\,p'\,\}\,c\,\{\,q'\,\}}{\{\,p \wedge p'\,\}\,c\,\{\,q \wedge q',t\,\}} \tag{xv}$$

Rule of Result Elimination:

$$\frac{\{\,p\,\}\,c\,\{\,q,t\,\}}{\{\,p\,\}\,c\,\{\,q\,\}} \tag{xvi}$$

Rule of Result Substitution:

$$\frac{\{\,p\,\}\,c\,\{\,q,t\,\} \quad q \Rightarrow t \doteq t'}{\{\,p\,\}\,c\,\{\,q,t'\,\}} \tag{xvii}$$

Rule of Logical Variable Elimination:

$$\frac{\{\,p\,\}\,c\,\{\,q,t\,\}}{\{\,p^{\nu}_{t'}\,\}\,c\,\{\,q,t\,\}} \tag{xviii}$$

where $Fn(q,t) \cap \{\nu\} = \emptyset$.

Each of the Rules (i), (ii), (v), (vi), (vii), (xi), (xiii), (xv) and (xviii) has a corresponding variant without the result values in all the correctness formulas of the premise and in the correctness formula of the conclusion, indicated by the same rule label.

We introduce next two derived rules.

Sequential Composition of n commands: The following rule for sequential composition results from $n-1$ applications of Rule (xi):

$$\frac{\{\,p\,\}\,c_1\,\{\,p_1\,\} \quad \{\,p_1\,\}\,c_2\,\{\,p_2\,\} \quad \ldots \quad \{\,p_{n-2}\,\}\,c_{n-1}\,\{\,p_{n-1}\,\} \quad \{\,p_{n-1}\,\}\,c_n\,\{\,q,t\,\}}{\{\,p\,\}\,c_1;\ c_2; \cdots ;\ c_n\,\{\,q,t\,\}} \tag{xix}$$

Simple Instantiation Rule: The Simple Instantiation Rule uses a weaker precondition in one of the premises of the Instantiation Rule (i):

$$\frac{\{\,p\,\}\,c\,\{\,q,t\,\} \quad \{\,\nu \doteq t_0\,\}\,c\,\{\,\nu \doteq t_0\,\}}{\{\,p^{\nu}_{t_0}\,\}\,c\,\{\,q^{\nu}_{t_0}, t^{\nu}_{t_0}\,\}} \tag{xx}$$

A formal proof consists of a sequence of applications of the inference rules of the proof system. At each step of such a sequence, we present the proofs by showing the inference rule which has been applied.

However, instead of introducing other derived rules, such as for example the one associated with a conjunction rule having equal preconditions in the premises, we often write a step of a proof indicating the application of more than one rule. In these cases, the "main" rule applied is indicated by a label in boldface font and the "extra" rules, separated by commas, correspond to trivial applications. Common examples of such trivial applications are applications of the Consequence Rule for which a formula is substituted by a (trivially) logically equivalent one and applications of the Result Elimination Rule.

We are interested in doing proofs from a set of formulas which include the set of all formulas valid in the interpretation $\hat{\mathbf{I}}$ defined in Section 3.2. The *universal closures* of the following formulas[5] on equality of object structures are used, for which validity is a direct consequence of the meaning functions defined in Section 3.2.

[5]A formula p is called *closed* when no free variable occurs in p (i.e. when $Fv(p) = \emptyset$). The closed formula $\forall \nu_1,\ldots,\nu_n : p$, where $\nu_1,\ldots,\nu_n$ are the variables that occur free in p (i.e. where $Fv(p) = \{\nu_1,\ldots,\nu_n\}$) is called the *universal closure* of p.

In the following consider ν_0 a logical variable of non-primitive type, n and k, respectively, the lengths of the lists of variables of the types of ν_0 and `self`, $v_1, \ldots, v_k$ the list of variables of the type of `self`, n_0 as defined in Chapter 3 (page 40), $1 \leq i \leq k$ and $1 \leq j \leq n$. We have:

$$(\iota \doteq \underline{i} \wedge \nu \doteq \mathtt{self}) \Rightarrow \nu[\iota] \doteq v_i \tag{xxi}$$

$$\iota \doteq \underline{j} \Rightarrow <\nu_0 : \nu_1, \ldots, \nu_n> [\iota] \doteq \nu_j \tag{xxii}$$

$$<\nu_0 : \nu_1, \ldots, \nu_n> \doteq <\nu_0' : \nu_1', \ldots, \nu_n'> \Leftrightarrow \nu_0 \doteq \nu_0' \wedge \nu_1 \doteq \nu_1' \wedge \ldots \wedge \nu_n \doteq \nu_n' \tag{xxiii}$$

$$\nu_0 \doteq <\nu_0 : \nu_0[\mathtt{1}], \ldots, \nu_0[\underline{n}]> \tag{xxiv}$$

$$\iota' < n_0 \Rightarrow \nu_0[\iota][\iota'] \doteq \nu_0[\iota * \underline{n}_0 + \iota'] \tag{xxv}$$

$$\nu_0[\iota] \not\doteq \nu_1 \Rightarrow \nu_0 \doteq \nu_0[\nu_1, \underline{i}, \nu_2] \tag{xxvi}$$

Chapter 5

Examples

This chapter presents some examples of the use of the proof rules to show program correctness.

We introduce first some notational conventions used in the examples:

- Logical variables with free occurrences in a given correctness formula (i.e. which occur free in the precondition, postcondition or result value of a correctness formula) are denoted by a capitalized name (i.e. whose first letter is a capital letter), possibly primed or subscripted. The capital letter A is generally used (possibly primed or subscripted) for terms which are ultimately instantiated to `self`. Otherwise, logical variables are denoted by ν or ι (also possibly primed or subscripted). In some cases and when it is explicitly indicated, i, j or k are used as names of logical variables.
- $h_1 \not\doteq h_2$ is equivalent to $\neg(h_1 \doteq h_2)$, for any phrases h_1 and h_2.
- A command such as:

  $$\texttt{if}\ c_1\ \texttt{then}\ c_2$$

  where $c_2 \circ C$ but not $c_2 \circ E[\tau]$, for any τ, is equivalent to

  $$\texttt{if}\ c_1\ \texttt{then}\ c_2\ \texttt{else}\ e$$

  for any expression e.[1] Note that such a conditional command has no type and is not an expression: see Rule 17, Figure 2.1.
- $odn!\texttt{new}(\texttt{nil}, \ldots, \texttt{nil})$ is abbreviated to $odn!\texttt{new}$ (where "..." indicate as many `nil`s as the length of the list of variables of odn).

5.1 A First Example

We start with an introductory, extremely simple example. Consider an object definition for objects representing coins which can be created, turned and tested upon its current state:

$$\boxed{\begin{array}{l} \mathit{Coin} \\ \quad x : \texttt{Bool} \\ \\ \quad \texttt{method}\ \mathit{turn} \\ \qquad x \leftarrow \neg x \\ \\ \quad \texttt{method}\ \mathit{val} : \texttt{Bool} \\ \qquad x \end{array}}$$

[1] I.e. for any e such that $e \circ E[\tau]$, where τ is arbitrary.

Let us consider the following two simple correctness formulas (where p denotes any formula):[2]

$$\{\, x \doteq X \,\}\, x \leftarrow \neg x \,\{\, x \doteq \neg X \,\} \tag{1}$$

$$\{\, p \,\}\, x \,\{\, p, x \,\} \tag{2}$$

Correctness formula (2) is simply an instance of Axiom scheme (iii).

Correctness formula (1) has the following trivial proof:

$$\dfrac{\dfrac{\{\, x \doteq X \,\}\, \neg x \,\{\, x \doteq X, \neg x \,\} \quad x \doteq X \Rightarrow \neg x \doteq \neg X}{\{\, x \doteq X \,\}\, \neg x \,\{\, \neg x \doteq \neg X, \neg x \,\}}\ \mathrm{ii}}{(1)}\ \mathbf{iv},\mathrm{xvi}$$

Let c denote the following command:

$$\begin{array}{l} (\, t : \mathit{Coin} \\ \quad t \leftarrow \mathit{Coin}!\mathtt{new}(\mathtt{false}); \\ \quad t.\mathit{turn}; \\ \quad t.\mathit{val}\,) \end{array}$$

We now prove the following:

$$\{\, \mathtt{true} \,\}\, c \,\{\, t[1], t[1] \,\} \tag{3}$$

Consider:

$$\{\, \mathtt{true} \,\}\, t \leftarrow \mathit{Coin}!\mathtt{new}(\mathtt{false}) \,\{\, \neg t[1] \,\} \tag{4}$$

$$\{\, \neg t[1] \,\}\, t.\mathit{turn} \,\{\, t[1] \,\} \tag{5}$$

$$\{\, t[1] \,\}\, t.\mathit{val} \,\{\, t[1], t[1] \,\} \tag{6}$$

The proof is firstly decomposed as:

$$\dfrac{(4) \quad (5) \quad (6)}{(3)}\ \mathrm{xix}$$

The proofs of (4), (5) and (6) are respectively as follows:

$$\dfrac{\{\, \mathtt{true} \,\}\, t \leftarrow \mathit{Coin}!\mathtt{new}(\mathtt{false}) \,\{\, \mathtt{true} \wedge t[1] \doteq \mathtt{false}, t \,\}}{(4)}\ \mathbf{xvi},\mathrm{ii}$$

Correctness formula $\{\, \mathtt{true} \,\}\, t \leftarrow \mathit{Coin}!\mathtt{new}(\mathtt{false}) \,\{\, \mathtt{true} \wedge t[1] \doteq \mathtt{false}, t \,\}$ is an instance of Axiom scheme (ix).

$$\dfrac{\dfrac{\dfrac{(1)}{\{\, x \doteq X \,\}\, \mathtt{self}.\mathit{turn} \,\{\, x \doteq \neg X \,\}}\ \mathrm{vi} \quad \{\, X \doteq \mathtt{false} \,\}\, \mathtt{self}.\mathit{turn} \,\{\, X \doteq \mathtt{false} \,\}}{\{\, \neg x \,\}\, \mathtt{self}.\mathit{turn} \,\{\, x \,\}}\ \mathbf{xx},\mathrm{ii}}{(5)}\ \mathrm{vii}$$

$$\dfrac{\dfrac{(2)}{\{\, x \,\}\, \mathtt{self}.\mathit{val} \,\{\, x, x \,\}}\ \mathrm{vi}}{(6)}\ \mathrm{vii}$$

[2] These are correctness formulas associated with method bodies. They are proven and subsequently used in the derivation of correctness formulas whose commands are corresponding message sending commands.

5.2 Sorting

The following is a sequential version of a sorting program used by Cliff Jones in his work on a design notation for concurrent object-based programs[Jon92a].[3]

$Sort$
  m : `Int`, l : $Sort$

  method $add\,(e$: `Int`$)$
    if $m \doteq$ `nil`
      then $(\, m \leftarrow e;\ l \leftarrow Sort!$`new`$\,)$
      else if $m < e$
        then $l.add(e)$
        else $(\, l.add(m);\ m \leftarrow e\,)$

  method rem : `Int`
    t : `Int`
    $(\ t \leftarrow m;$
     if $m \not\doteq$ `nil` **then**
      $(\ m \leftarrow l.rem;$
       if $m \doteq$ `nil` **then** $l \leftarrow$ `nil` $)$
     $t\ )$

In this section consider i as a logical variable of type $\mathtt{Z}$.
Let, where t is any given term:[4]

$$\not\circlearrowleft t \equiv (\forall i : t[i] \not\doteq t)$$

Considering S_0 as a logical variable of type $\mathtt{Z}^*$, let:[5]

$$p \equiv \big(S_0 \doteq seq(\mathtt{self}) \land e \not\doteq \mathtt{nil} \land nondec(S_0) \land \not\circlearrowleft \mathtt{self}\big)$$

[3] We give in this section a proof of a correctness formula relative to the body of method add only. Method rem is not used in the sequel (since the proofs involved would not introduce significant new material).

[4] Informally, the formula $\forall i : t[i] \not\doteq t$ below indicates that t denotes an acyclic object structure.

[5] Informally, the conjunctions in p indicate, respectively, that:

- S_0 is the sequence of integer numbers (i.e. primitive objects different from nil) that is extracted (by seq) from the object structure denoted by `self`,
- e is not `nil`,
- S_0 is a non-decreasing sequence, and
- the object structure denoted by `self` is acyclic.

For any S of type $\mathtt{Z}$, $nondec(S)$ is an abbreviation for:

$$\forall i : i < length(S) \Rightarrow S(i) \leq S(i+1)$$

The function symbol $length$ above, with type $(\mathtt{Z}^* \rightarrow \mathtt{Z})$, is a member of a basis set (which also includes $\frown$, see below) for which the usual interpretation defines a *theory of sequences of integers*. The following formulas (and their universal closures) are valid in this theory:

$$length([\,]) = 0$$
$$length(S^\frown[i]) = length(S) + 1$$

$$S^\frown[\,] \doteq S$$
$$S_1{}^\frown(S_2{}^\frown[i]) \doteq (S_1{}^\frown S_2)^\frown[i]$$

Let, where textual substitution is used for brevity:[6]

$$q \equiv \not\circlearrowleft \texttt{self} \wedge \left(\exists i : \nu(i) \doteq e \wedge del(\nu, i) \doteq S_0 \wedge nondec(\nu)\right)^{\nu}_{seq(\texttt{self})}$$

For brevity, let $c;\texttt{self}$ denote the command which is the body of method *add* above, and let also $c \equiv \texttt{if}\, m \doteq \texttt{nil}\, \texttt{then}\, c_1\, \texttt{else}\, c_2$.

We assume that $seq : Sort \rightharpoonup \texttt{Z}^*$ is such that the following formulas are valid, where t has type *Sort* (and i has type $\texttt{Z}$):

$$t \doteq \texttt{nil} \Rightarrow seq(t) \doteq [\,] \tag{7}$$

$$t[1] \doteq \texttt{nil} \Rightarrow seq(t) \doteq [\,] \tag{8}$$

$$t[1] \not\doteq \texttt{nil} \Rightarrow \left(i = t[1] \Rightarrow seq(t) \doteq [i]^\frown seq(t[2])\right) \tag{9}$$

We prove next the following correctness formula:[7]

$$\{\, p \,\}\, c\, \{\, q \,\} \tag{10}$$

Considering:

$$\{\, p \wedge m \doteq \texttt{nil} \,\}\, c_1\, \{\, q \,\} \tag{11}$$

$$\{\, p \wedge m \not\doteq \texttt{nil} \,\}\, c_2\, \{\, q \,\} \tag{12}$$

the proof is firstly decomposed as follows:

$$\frac{\{\, p \,\}\, m \doteq \texttt{nil}\, \{\, p, m \doteq \texttt{nil} \,\} \quad (11) \quad (12)}{(10)}\ \text{xii}$$

(11) is proved first; considering, for brevity:

$$\begin{array}{l} p_1 \equiv S_0 \doteq [\,] \wedge e \not\doteq \texttt{nil} \\ p' \equiv p_1 \wedge m \doteq e \\ p_1' \equiv p_1 \wedge e \doteq e \\ q_1 \equiv p' \wedge l[1] \doteq \texttt{nil} \\ q' \equiv q_1 \wedge l[2] \doteq \texttt{nil} \\ c_0 \equiv l \leftarrow Sort!\texttt{new} \end{array}$$

the proof is as follows:

---

[6] Again informally speaking, the conjunctions in q below indicate, respectively, that:

- the object structure denoted by `self` is acyclic,
- the sequence of integer numbers that is extracted (by *seq*) from the object structure denoted by `self` contains e, if e is removed from this sequence then a sequence equal to S_0 is obtained, and this sequence is non-decreasing.

The function symbol *del* occurring in q, with type $(\texttt{Z}^*, \texttt{Z} \to \texttt{Z}^*)$ is a member of the basis set (which includes *length* and $^\frown$) for which the usual interpretation defines a *theory of sequences of integers*. The following formulas (and their universal closures) are valid in this theory ($a < b < c$ is an abbreviation of $a < b \wedge b < c$, and similarly for $\leq$ etc.):

$$0 < i \leq length(S) \Rightarrow del(S, i) \doteq S(1, i-1)^\frown S(i+1, length(S))$$

$$0 < i \wedge i < j \Rightarrow S(i, j) = [\,]$$
$$0 < i \leq j \wedge i \leq length(S) \Rightarrow S(i, j) = S(i)^\frown S(i+1, j)$$

[7] From which $\{\, p \,\}\, c;\texttt{self}\, \{\, q, \texttt{self} \,\}$ can be proved trivially.

$$\dfrac{p \wedge m \doteq \mathtt{nil} \Rightarrow p_1 \qquad \dfrac{\dfrac{\dfrac{\{\,p_1\,\}\,e\,\{\,p_1, e\,\} \quad p_1 \Rightarrow p_1'}{\{\,p_1\,\}\,e\,\{\,p_1', e\,\}}\,\mathrm{ii}}{\{\,p_1\,\}\,m \leftarrow e\,\{\,p'\,\}}\,\mathbf{iv},\mathrm{xvi} \quad \dfrac{\{\,p'\,\}\,c_0\,\{\,q', l\,\}}{\{\,p'\,\}\,c_0\,\{\,q_1\,\}}\,\mathbf{xvi},\mathrm{ii}}{\{\,p_1\,\}\,c_1\,\{\,q_1\,\}}\,\mathrm{xi} \qquad q_1 \Rightarrow q}{(11)}\,\mathrm{ii}$$

The formula $p \wedge m \doteq \mathtt{nil} \Rightarrow p_1$ can be proved, by using a calculus for predicate logic, from Axioms (8) and (xxi).

The formula $q_1 \Rightarrow q$ can be proved, by using a calculus for predicate logic, from Axioms (8), (9) and (xxi), and axioms on sequence indexing, *del* and *nondec*, whose logical consequences give, respectively, $s \doteq [x] \Rightarrow s(\mathtt{1}) \doteq x$, $s \doteq [x] \Rightarrow del(s, \mathtt{1}) \doteq [\,]$ and $s \doteq [x] \Rightarrow nondec(s)$.

We prove (12) next; consider now:

$$p' \equiv p \wedge m \not\doteq \mathtt{nil} \wedge m < e$$
$$p'' \equiv p \wedge m \not\doteq \mathtt{nil} \wedge m \geq e$$

$$\{\,p'\,\}\; l.add(e)\; \{\,q\,\} \tag{13}$$

$$\{\,p''\,\}\; l.add(m);\; m \leftarrow e\; \{\,q\,\} \tag{14}$$

The proof is further decomposed as:

$$\dfrac{\{\,p \wedge m \not\doteq \mathtt{nil}\,\}\; m < e\; \{\,p \wedge m \not\doteq \mathtt{nil}, m < e\,\} \quad (13) \quad (14)}{(12)}\,\mathrm{xii}$$

Considering, for brevity:

$$c_3 \equiv \quad l.add(e)$$
$$c_4 \equiv \quad \mathtt{self}.add(e)$$

$$\{\,A \doteq \mathtt{self}\,\}\; c_3\; \{\,A \doteq \mathtt{self}\,\} \tag{15}$$

and:

$$\begin{aligned}
p_1 &\equiv \quad l \doteq A[2] \wedge (p')^{\mathtt{self}, m}_{A, A[1]} \\
q_1 &\equiv \quad q^{\mathtt{self}}_{A} \\
p_2 &\equiv \quad (p_1)^{l}_{\mathtt{self}} \\
p_3 &\equiv \quad \mathtt{self} \doteq A[2] \wedge S_0 \doteq [A[1]]^\frown S_1 \wedge S_1 \doteq seq(\mathtt{self}) \wedge e \not\doteq \mathtt{nil} \wedge nondec(S_0) \wedge \\
&\quad\;\; \not\emptyset\, A \wedge A[1] \not\doteq \mathtt{nil} \wedge A[1] < e \\
p_i &\equiv \quad \mathtt{self} \doteq A[2] \wedge S_0 \doteq [A[1]]^\frown S_1 \wedge A \not\doteq \mathtt{self} \wedge A[1] \not\doteq \mathtt{nil} \wedge A[1] < e \\
q_i &\equiv \quad S_0 \doteq S_1
\end{aligned}$$

(13) is proved as follows:

$$\dfrac{\dfrac{\dfrac{p_3 \Rightarrow p^{S_0}_{S_1} \wedge p_i \qquad \dfrac{\dfrac{[\![(10)]\!] \quad \{\,q_i\,\}\,c_4\,\{\,q_i\,\}}{\{\,p^{S_0}_{S_1}\,\}\,c_4\,\{\,q^{S_0}_{S_1}\,\}}\,\mathrm{xx} \qquad \{\,p_i\,\}\,c_4\,\{\,p_i\,\}}{\{\,p^{S_0}_{S_1} \wedge p_i\,\}\,c_4\,\{\,q^{S_0}_{S_1} \wedge p_i\,\}}\,\mathrm{xv} \qquad q^{S_0}_{S_1} \wedge p_i \Rightarrow q_1}{\dfrac{\dfrac{\{\,p_3\,\}\,c_4\,\{\,q_1\,\}}{\{\,p_2\,\}\,c_4\,\{\,q_1\,\}}\,\mathbf{xviii},\mathrm{ii}}{\{\,p_1\,\}\,c_3\,\{\,q_1\,\}}\,\mathrm{vii}}\,\mathrm{ii}\qquad (15)}{(13)}\,\mathbf{xx},\mathrm{ii}$$

$\{ p_i \}\, c_4\, \{ p_i \}$ can be shown as follows. Consider:

$$\begin{array}{ll}
p_i[e/x] \equiv & \mathtt{self} \doteq A[2] \wedge S_0 \doteq [(\mathtt{if}\ A \doteq \mathtt{self}\ \mathtt{then}\ e\ \mathtt{else}\ A[1])]^\frown S_1 \wedge \\
& A \not\doteq \mathtt{self} \wedge (\mathtt{if}\ A \doteq \mathtt{self}\ \mathtt{then}\ e\ \mathtt{else}\ A[1]) \not\doteq \mathtt{nil} \wedge \\
& (\mathtt{if}\ A \doteq \mathtt{self}\ \mathtt{then}\ e\ \mathtt{else}\ A[1]) < e \\
\\
p_i[\nu/l] \equiv & \mathtt{self} \doteq (\mathtt{if}\ A \doteq \mathtt{self}\ \mathtt{then}\ \nu\ \mathtt{else}\ A[2]) \wedge \\
& S_0 \doteq [A[1]]^\frown S_1 \wedge A \not\doteq \mathtt{self} \wedge A[1] \not\doteq \mathtt{nil} \wedge A[1] < e \\
\\
p_0 \equiv & \forall \nu : p_i[\nu/l] \\
p_i' \equiv & p_i[e/x] \\
p_i'' \equiv & p_i \wedge l[1] \doteq \mathtt{nil} \wedge l[2] \doteq \mathtt{nil}
\end{array}$$

then the fact that p_i is invariant is clear from:

$$\dfrac{\dfrac{\{ p_i \}\, e\, \{ p_i \} \quad p_i \Rightarrow p_i'}{\{ p_i \}\, e\, \{ p_i' \}}\ \mathrm{ii}}{\{ p_i \}\, m \leftarrow e\, \{ p_i \}}\ \mathrm{iv} \qquad \dfrac{p_i \Rightarrow p_0 \quad \{ p_0 \}\, c_0\, \{ p_i'' \} \quad p_i'' \Rightarrow p_i}{\{ p_i \}\, c_0\, \{ p_i \}}\ \mathrm{ii} \dfrac{[\![\{ p_i \}\, c_4\, \{ p_i \}]\!]}{\{ p_i \}\, c_3\, \{ p_i \}}\ \mathrm{viii}$$

$$\vdots$$

$$\dfrac{\{ p_i \}\, c\, \{ p_i \}}{\{ p_i \}\, c_4\, \{ p_i \}}\ \mathrm{vi}$$

Considering, for brevity:

$$\begin{array}{ll}
c_5 \equiv & l.add(m) \\
c_6 \equiv & m \leftarrow e \\
c_7 \equiv & \mathtt{self}.add(e)
\end{array}$$

$$\{ E \doteq e \}\, c_5\, \{ E \doteq e \} \tag{16}$$

$$\{ A \doteq \mathtt{self} \}\, c_5\, \{ A \doteq \mathtt{self} \} \tag{17}$$

$$\begin{array}{ll}
p_1 \equiv & q^{\mathtt{self}}_{<\mathtt{self}:e,l[\mathtt{self},1,e]>} \\
q' \equiv & q^{\mathtt{self}}_{<\mathtt{self}:m,l>} \\
q_1 \equiv & q^{\mathtt{self}}_{<\mathtt{self}:e,l>} \wedge \forall \iota : l[\iota] \not\doteq \mathtt{self} \\
q_2 \equiv & q^{\mathtt{self}}_{<\mathtt{self}:e,l>} \\
p_2 \equiv & l \doteq A[2] \wedge (p'')^{\mathtt{self}}_{A} \\
q_3 \equiv & (q_2)^{\mathtt{self}}_{A} \\
p_3 \equiv & (p_2)^{e}_{E} \\
q_4 \equiv & (q_3)^{e}_{E} \\
p_4 \equiv & (p_2)^{l,m}_{\mathtt{self},e} \\
q_5 \equiv & (q_4)^{l}_{\mathtt{self}} \\
p_5 \equiv & \emptyset\ \mathtt{self} \wedge S_0 \doteq [e]^\frown seq(\mathtt{self}) \wedge e \not\doteq \mathtt{nil} \wedge nondec(S_0) \wedge \\
& E \not\doteq \mathtt{nil} \wedge E \leq e \wedge A \not\doteq \mathtt{self} \wedge \mathtt{self} \doteq A[2] \\
q_6 \equiv & A \not\doteq \mathtt{self} \wedge seq(\mathtt{self}) \doteq S_0 \wedge nondec(S_0) \wedge \\
& E \not\doteq \mathtt{nil} \wedge \forall \iota : E \leq S_0(\iota) \wedge \emptyset\ \mathtt{self} \wedge \mathtt{self} \doteq A[2] \\
p_i \equiv & S_0 \doteq [e]^\frown S_1 \wedge nondec(S_0) \wedge E \not\doteq \mathtt{nil} \wedge E \leq e \wedge A \not\doteq \mathtt{self} \wedge \mathtt{self} \doteq A[2] \\
p_6 \equiv & p^{S_0}_{S_1} \wedge p_i \\
q_7 \equiv & q^{S_0}_{S_1} \wedge p_i
\end{array}$$

(14) is proved as follows:

$$\dfrac{p_4 \Rightarrow p_5 \qquad \dfrac{\dfrac{\dfrac{\dfrac{[\![(10)]\!] \quad \{\, S_0 \doteq S_1 \,\}\; c\; \{\, S_0 \doteq S_1 \,\}}{\{\, p^{S_0}_{S_1} \,\}\; c\; \{\, q^{S_0}_{S_1} \,\}}\;\mathbf{xx},\text{vi} \qquad \{\, p_i \,\}\; c_7\; \{\, p_i \,\}}{\{\, p_6 \,\}\; c_7\; \{\, q_7 \,\}}\;\text{xv} \qquad q_7 \Rightarrow q_6}{\{\, p_6 \,\}\; c_7\; \{\, q_6 \,\}}\;\text{ii}}{\{\, p_5 \,\}\; c_7\; \{\, q_6 \,\}}\;\mathbf{xviii},\text{ii} \qquad q_6 \Rightarrow q_5}{\{\, p_4 \,\}\; c_7\; \{\, q_5 \,\}}\;\text{ii}$$

and:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\{\, p_4 \,\}\; c_7\; \{\, q_5 \,\}}{\{\, p_3 \,\}\; c_5\; \{\, q_4 \,\}}\;\text{vii} \qquad (16)}{\{\, p_2 \,\}\; c_5\; \{\, q_3 \,\}}\;\text{xx} \qquad (17)}{\{\, p'' \,\}\; c_5\; \{\, q_2 \,\}}\;\mathbf{xx},\text{ii} \qquad q_2 \Rightarrow q_1}{\{\, p'' \,\}\; c_5\; \{\, q_1 \,\}}\;\text{ii} \qquad q_1 \Rightarrow p_1}{\{\, p'' \,\}\; c_5\; \{\, p_1 \,\}}\;\text{ii} \qquad \{\, p_1 \,\}\; c_6\; \{\, q \,\}}{(14)}\;\text{xi}$$

and:

$$\dfrac{\dfrac{\{\, p_1 \,\}\; e\; \{\, p_1, e \,\}}{\{\, p_1 \,\}\; c_6\; \{\, q' \,\}}\;\mathbf{iv},\text{xvi} \qquad q' \Rightarrow q}{\{\, p_1 \,\}\; c_6\; \{\, q \,\}}\;\text{ii}$$

5.3 Mutual Recursion

This section is intended to illustrate proofs applied to programs involving mutually recursive methods. The following program contains a method that computes the remainder modulo 2 of a natural number given as an argument:[8]

$Mod2$
  $\texttt{method}\ m\,(x : \texttt{Int}) : \texttt{Int}$
    $\texttt{if}\ x \doteq \texttt{0}\ \texttt{then}\ \texttt{0}\ \texttt{else}\ \texttt{self}.m'(x - \texttt{1})$

  $\texttt{method}\ m'\,(x : \texttt{Int}) : \texttt{Int}$
    $\texttt{if}\ x \doteq \texttt{0}\ \texttt{then}\ \texttt{1}\ \texttt{else}\ \texttt{self}.m(x - \texttt{1})$

Letting, for brevity:

$$p \equiv x \doteq X \wedge X \geq \texttt{0}$$

we prove (denoting also for brevity by c the body of m and c' the body of m'):

$$\{\, p \,\}\; c\; \{\, \texttt{true}, X \,\texttt{mod}\, \texttt{2} \,\} \tag{18}$$

$$\{\, p \,\}\; c'\; \{\, \texttt{true}, (X + \texttt{1}) \,\texttt{mod}\, \texttt{2} \,\} \tag{19}$$

Considering (for brevity):

$$\begin{aligned} c_1 &\equiv \quad \texttt{self}.m'(x - \texttt{1}) \\ c_1' &\equiv \quad \texttt{self}.m(x - \texttt{1}) \\ c &\equiv \quad \texttt{self}.m(x) \end{aligned}$$

[8]The example is from [LS87, Section 3.3, Example 3.13].

$$
\begin{array}{rl}
p_1 \equiv & p \wedge x \doteq 0 \\
p_2 \equiv & p \wedge x \neq 0 \\
p_3 \equiv & x \doteq X \wedge X \doteq 0 \\
p_4 \equiv & (p_2)^{X}_{X+1} \\
p_5 \equiv & p^{x}_{x-1} \\
p_6 \equiv & X \geq 0 \Rightarrow (X+2) \,\mathtt{mod}\, 2 \doteq X \,\mathtt{mod}\, 2
\end{array}
$$

$$\{\, p_2 \,\}\; c_1 \;\{\, \mathtt{true}, X \,\mathtt{mod}\, 2 \,\} \tag{20}$$

$$\{\, p_2 \,\}\; c_1' \;\{\, \mathtt{true}, (X+1) \,\mathtt{mod}\, 2 \,\} \tag{21}$$

$$\{\, X \doteq X - 1 \,\}\; c_1 \;\{\, X \doteq X - 1 \,\} \tag{22}$$

$$\{\, X \geq 0 \,\}\; c \;\{\, X \geq 0 \,\} \tag{23}$$

$$\{\, p \,\}\; x \doteq 0 \;\{\, \mathtt{true}, x \doteq 0 \,\} \tag{24}$$

the proofs are as follows:

$$
\dfrac{\dfrac{p_4 \Rightarrow p_5 \qquad \dfrac{\dfrac{\dfrac{[\![\{\, p \,\}\; c \;\{\, \mathtt{true}, X \,\mathtt{mod}\, 2 \,\}]\!] \quad (23)}{\{\, p \,\}\; c \;\{\, X \geq 0, X \,\mathtt{mod}\, 2 \,\}}\,\mathrm{xx} \qquad p_6}{\{\, p \,\}\; c \;\{\, X \geq 0, (X+2) \,\mathtt{mod}\, 2 \,\}}\,\mathrm{xvii}}{\{\, p_5 \,\}\; c_1' \;\{\, X \geq 0, (X+2) \,\mathtt{mod}\, 2 \,\}}\,\mathrm{vii} \qquad X \geq 0 \Rightarrow \mathtt{true}}{\{\, p_4 \,\}\; c_1' \;\{\, X \geq 0, (X+2) \,\mathtt{mod}\, 2 \,\}}\,\mathrm{ii} \qquad (22)}{(21)}\,\mathbf{xx},\mathrm{ii}
$$

and:

$$
\dfrac{(24) \qquad \dfrac{p_1 \Rightarrow p_3 \qquad \dfrac{\{\, p_3 \,\}\; 1 \;\{\, p_3, 1 \,\} \quad p_3 \Rightarrow (X+1) \,\mathtt{mod}\, 2 \doteq 1}{\{\, p_3 \,\}\; 1 \;\{\, p_3, (X+1) \,\mathtt{mod}\, 2 \,\}}\,\mathrm{xvii} \qquad p_3 \Rightarrow \mathtt{true}}{\{\, p_1 \,\}\; 1 \;\{\, \mathtt{true}, (X+1) \,\mathtt{mod}\, 2 \,\}}\,\mathrm{ii} \qquad (21)}{(19)}\,\mathrm{xii}
$$

and:

$$
\dfrac{\dfrac{\dfrac{(19)}{\{\, p \,\}\; \mathtt{self}.m'(x) \;\{\, \mathtt{true}, (X+1) \,\mathtt{mod}\, 2 \,\}}\,\mathrm{vi}}{\{\, p_4 \,\}\; c_1 \;\{\, \mathtt{true}, (X+1) \,\mathtt{mod}\, 2 \,\}}\,\mathbf{vii},\mathrm{ii} \qquad (22)}{(20)}\,\mathrm{xx}
$$

and:

$$
\dfrac{(24) \qquad \dfrac{p_1 \Rightarrow p_3 \qquad \dfrac{\{\, p_3 \,\}\; 0 \;\{\, p_3, 0 \,\} \quad p_3 \Rightarrow X \,\mathtt{mod}\, 2 \doteq 0}{\{\, p_3 \,\}\; 0 \;\{\, p_3, X \,\mathtt{mod}\, 2 \,\}}\,\mathrm{xvii} \qquad p_3 \Rightarrow \mathtt{true}}{\{\, p_1 \,\}\; 0 \;\{\, \mathtt{true}, X \,\mathtt{mod}\, 2 \,\}}\,\mathrm{ii} \qquad (20)}{(18)}\,\mathrm{xii}
$$

Chapter 6

Soundness

This chapter proves the soundness of **P1** with respect to validity in interpretation $\widehat{\mathbf{I}}$. That is, it proves that if a correctness formula $\{\,p\,\}\,c\,\{\,q,t\,\}$ — or $\{\,p\,\}\,c\,\{\,q\,\}$ —, where p,q are formulas and t is a term of **L1** and c is a command of **A1** occurring (in a given program) in an object definition associated to ϱ, is provable in **P1** from a set W of formulas and/or correctness formulas or **L1** for which $\widehat{\mathbf{I}}$ is a model (see below), then this correctness formula is valid with respect to ϱ and interpretation $\widehat{\mathbf{I}}$.

We consider validity in the given interpretation $\widehat{\mathbf{I}}$, and not in validity in all interpretations.

Sections 6.1 to 6.14 state and prove lemmas of soundness of the inference rules and axiom schemes of **P1**. The last section states and proves the soundness theorem.

To present the soundness theorem we introduce below some definitions and notational conventions. The symbol $\Box$ is used to indicate the end of a proof.

Definition 19 (Truth and validity of formulas) Let p be a formula.

p is true with respect to interpretation $\widehat{\mathbf{I}}$, a state σ, a non-primitive object α and an assignment $\hat{s}_{\sigma,\alpha}$ (for $\widehat{\mathbf{I}}$, relative to that state and that non-primitive object) — denoted by $\hat{s}_{\sigma,\alpha} \models p$ — if $\widehat{\mathbf{I}}[\![p]\!](\sigma,\hat{s}_{\sigma,\alpha}) = \mathit{true}$.[1] Otherwise, p is false with respect to interpretation $\widehat{\mathbf{I}}$, state σ and assignment $\hat{s}_{\sigma,\alpha}$ — denoted by $\hat{s}_{\sigma,\alpha} \not\models p$.

p is valid in interpretation $\widehat{\mathbf{I}}$ — denoted by $\models p$ — if $\hat{s}_{\sigma,\alpha} \models p$ for all states σ, non-primitive objects α and assignments $\hat{s}_{\sigma,\alpha}$. Otherwise, p is not valid in interpretation $\widehat{\mathbf{I}}$ — denoted by $\not\models p$.

Definition 20 (Truth and validity of correctness formulas) Let p,q be formulas, t a term and c a command occurring in an object definition associated to ϱ. Let $w \equiv \{\,p\,\}\,c\,\{\,q,t\,\}$ or $w \equiv \{\,p\,\}\,c\,\{\,q\,\}$.

w is true with respect to γ, ϱ and interpretation $\widehat{\mathbf{I}}$ — denoted by $\gamma \models w$ — if $\widehat{\mathbf{I}}[\![w]\!](\gamma)(\sigma,\hat{s}_{\sigma,\alpha}) = \mathit{true}$, for all states σ, non-primitive objects $\alpha \in A_\varrho \cap \mathtt{dom}(\sigma)$ and assignments $\hat{s}_{\sigma,\alpha}$.

w is valid with respect to ϱ and interpretation $\widehat{\mathbf{I}}$ — denoted by $\models w$ — if $\gamma \models w$, for all function assignments γ.

For clarity, we sometimes explicitly write $\varrho \models w$ instead of simply $\models w$, which assumes that ϱ refers to the object definition in which c occurs (and analogously for $\gamma,\varrho \models w$ and $\gamma \models w$, respectively).

Definition 21 ($\widehat{\mathbf{I}}$ as a Model) Interpretation $\widehat{\mathbf{I}}$ is a model of a set X of formulas or correctness formulas if $\models x$ for every $x \in X$.

Definition 22 (Soundness of inference rules) Let w be a correctness formula and $w_1,\ldots,w_n$ be formulas or correctness formulas. Let $w \equiv \{\,p\,\}\,c\,\{\,q,t\,\}$ — or $w \equiv \{\,p\,\}\,c\,\{\,q\,\}$ —, where c is a command occurring in an object definition ϱ. If w_i is a correctness formula, for any i between 1 and n, let

[1] The notation assumes that the state and the non-primitive object are those relative to which the assignment is defined. They must be explicitly specified (as, for example, with σ and α in $\hat{s}_{\sigma,\alpha}$). Also, the symbol $\models$ is used in this chapter as an abbreviation of $\models_{\widehat{\mathbf{I}}}$, which indicates that truth is considered with respect to interpretation $\widehat{\mathbf{I}}$.

$w_i \equiv \{\, p_i \,\}\; c_i\; \{\, q_i, t_i \,\}$ — or $w_i \equiv \{\, p_i \,\}\; c_i\; \{\, q_i \,\}$ —, where each such c_i occurs in an object definition ϱ' (the same or different from ϱ).

An inference rule r:

$$\frac{w_1 \quad \ldots \quad w_n}{w}$$

is sound with respect to ϱ, ϱ' and interpretation $\widehat{\mathbf{I}}$ — denoted by $\models r$ — if it holds that $\varrho \models w$ on the assumption that, for all i between 1 and n, it holds that $\varrho' \models w_i$, if w_i is a correctness formula, or $\models w_i$, if w_i is a formula.

An inference rule r:[2]

$$\begin{array}{c} [\![w_1']\!] \\ \dfrac{w_1}{w} \end{array}$$

is sound with respect to ϱ and ϱ' in interpretation $\widehat{\mathbf{I}}$ — denoted by $\models r$ — if it holds that: $\varrho \models w$ on the assumption that, for all γ, we have: $\gamma, \varrho' \models w_1$ if $\gamma, \varrho' \models w_1'$.[3]

6.1 Instantiation Rule

We use the following:

Lemma 1 For all $\sigma \in \Sigma$, $\hat{s}_{\sigma,\alpha} \in \widehat{S}$ and $x \in \mathtt{dom}(\hat{s}_{\sigma,\alpha})$, we have:

$$\text{if } \widehat{\mathbf{I}}[\![x]\!](\sigma, \hat{s}_{\sigma,\alpha}) = \widehat{\mathbf{I}}[\![t_0]\!](\sigma, \hat{s}_{\sigma,\alpha}) \text{ then } \widehat{\mathbf{I}}[\![t^x_{t_0}]\!](\sigma, \hat{s}_{\sigma,\alpha}) = \widehat{\mathbf{I}}[\![t]\!](\sigma, \hat{s}_{\sigma,\alpha})$$

Proof: By induction on the structure of t.[4] □

Lemma 2 For all $\sigma \in \Sigma$, $\hat{s}_{\sigma,\alpha} \in \widehat{S}$ and $x \in \mathtt{dom}(\hat{s}_{\sigma,\alpha})$, we have:

$$\text{if } \widehat{\mathbf{I}}[\![x]\!](\sigma, \hat{s}_{\sigma,\alpha}) = \widehat{\mathbf{I}}[\![t_0]\!](\sigma, \hat{s}_{\sigma,\alpha}) \text{ then } \widehat{\mathbf{I}}[\![p^x_{t_0}]\!](\sigma, \hat{s}_{\sigma,\alpha}) = \widehat{\mathbf{I}}[\![p]\!](\sigma, \hat{s}_{\sigma,\alpha})$$

Proof: Direct from Lemma 1, using induction on the structure of p. □

Lemma 3 For all $\sigma \in \Sigma$, $\hat{s}_{\sigma,\alpha} \in \widehat{S}$ and $x \in \mathtt{dom}(\hat{s}_{\sigma,\alpha})$, we have:

$$\widehat{\mathbf{I}}[\![x]\!](\sigma, \hat{s}_{\sigma,\alpha}) = \widehat{\mathbf{I}}[\![t_0]\!](\sigma, \hat{s}_{\sigma,\alpha}) \text{ if and only if } \hat{s}_{\sigma,\alpha} \models x \doteq t_0$$

Proof: Direct from the semantics of formulas (Section 3.2.2). □

We have:

[2] The Recursion Rule (v) has only one premise (and so only one discharged hypothesis). The definition of soundness for the general case (with possibly more than one premise) is analogous.

[3] Informally speaking, the "premise" 'w_1 on the assumption of w_1'' has to be universally quantified over γ, as occurs with the conclusion. The universal closure (over γ) of:

$$\gamma, \varrho \models w \text{ if } \left(\gamma, \varrho' \models w_1 \text{ if } \varrho' \models w_1'\right)$$

is not adequate. This can be seen more easily by considering that $\varrho' \not\models w_1'$ would imply that the "premise" is true, irrespective of w_1.

[4] Appendix A presents an illustrative example of a proof by induction on the structure of terms. Other proofs by induction on the structure of terms used in this thesis are analogous.

Lemma 4 (Soundness of the Instantiation Rule)

$$\models \{\, p^{\nu}_{t_0} \,\}\; c \;\{\, q^{\nu}_{t_0}, t^{\nu}_{t_0} \,\}$$

on the assumptions:

$$\models \{\, p \,\}\; c \;\{\, q, t \,\} \tag{1}$$

$$\models \{\, \nu \doteq t_0 \wedge p \,\}\; c \;\{\, \nu \doteq t_0 \,\} \tag{2}$$

Proof: Assuming

$$\hat{s}_{\sigma,\alpha} \models p^{\nu}_{t_0} \tag{3}$$

and

$$\mathbf{I}[\![c]\!](\alpha)(\gamma)(\sigma) = (\sigma', \theta) \tag{4}$$

it needs to be shown that

$$\hat{s}_{\sigma',\alpha} \models q^{\nu}_{t_0}$$

and

$$\widehat{\mathbf{I}}[\![t^{\nu}_{t_0}]\!](\sigma', \hat{s}_{\sigma',\alpha}) = \theta$$

Consider:

- $\nu \notin Fn(t_0)$. Then consider that $\hat{s}_{\sigma,\alpha}$ is such that:

  $$\hat{s}_{\sigma,\alpha}(\nu) = \widehat{\mathbf{I}}[\![t_0]\!](\sigma, \hat{s}_{\sigma,\alpha}) \tag{5}$$

  Then:

$\hat{s}_{\sigma,\alpha} \models p$	(6)	From Assumptions (3) and (5) (and semantics of terms), by Lemma 2
$\hat{s}_{\sigma,\alpha} \models \nu \doteq t_0$	(7)	From Assumption (5) (and semantics of terms), by Lemma 3
$\hat{s}_{\sigma',\alpha} \models \nu \doteq t_0$	(8)	From Assumption (2) and from (7) and (4), by Definitions 20 and 19 and semantics of formulas and correctness formulas
$\widehat{\mathbf{I}}[\![\nu]\!](\sigma', \hat{s}_{\sigma',\alpha}) = \widehat{\mathbf{I}}[\![t_0]\!](\sigma', \hat{s}_{\sigma',\alpha})$	(9)	From (8), by Lemma 3
$\hat{s}_{\sigma',\alpha} \models q$	(10)	From Assumption (1) and from (6) and (4), by Definitions 20 and 19 and semantics of formulas and correctness formulas
$\hat{s}_{\sigma',\alpha} \models q^{\nu}_{t_0}$		From (10) and (9), by Lemma 2
$\widehat{\mathbf{I}}[\![t]\!](\sigma', \hat{s}_{\sigma',\alpha}) = \theta$	(11)	From Assumption (1) and from (6) and (4), by Definitions 20 and 19 and semantics of formulas and correctness formulas
$\widehat{\mathbf{I}}[\![t^{\nu}_{t_0}]\!](\sigma', \hat{s}_{\sigma',\alpha}) = \theta$		From (11) and (9), by Lemma 1

- $\nu \in Fn(t_0)$. Then let $t'_0 \equiv (t_0)^{\nu}_{\nu'}$, where ν' is such that $\nu' \notin Fn(p, q, t, t_0)$. Then, we have from above that

  $$\frac{\{\, p \,\}\; c \;\{\, q, t \,\} \quad \{\, \nu \doteq t'_0 \wedge p \,\}\; c \;\{\, \nu \doteq t'_0 \,\}}{\{\, p^{\nu}_{t'_0} \,\}\; c \;\{\, q^{\nu}_{t'_0}, t^{\nu}_{t'_0} \,\}}$$

is sound. Then, considering that $\models \{\, \nu' \doteq \nu \,\}\, c \,\{\, \nu' \doteq \nu \,\}$, using the same argument as above we get:

$$\frac{\{\, p^{\nu}_{t'_0} \,\}\, c \,\{\, q^{\nu}_{t'_0}, t^{\nu}_{t'_0} \,\} \quad \{\, \nu' \doteq \nu \,\}\, c \,\{\, \nu' \doteq \nu \,\}}{\{\, p^{\nu}_{t_0} \,\}\, c \,\{\, q^{\nu}_{t_0}, t^{\nu}_{t_0} \,\}}$$

□

6.2 Consequence Rule

Lemma 5

$$\models \{\, p \,\}\, c \,\{\, q, t \,\}$$

on the assumptions:

$$\models p \Rightarrow p'$$
$$\models \{\, p' \,\}\, c \,\{\, q', t \,\}$$
$$\models q' \Rightarrow q$$

Proof: Direct from Definitions 19 and 20, and the semantics of correctness formulas and formulas. □

6.3 Expression Axiom Scheme

Lemma 6

$$\models \{\, p \,\}\, e \,\{\, p, e \,\}$$

Proof: Direct from the semantics of commands (page 37) and Definition 20. □

6.4 Assignment Rule

We use the following:

Lemma 7 Let $\sigma \in \Sigma$, $\alpha \in A_\varrho$, for some type ϱ, and $\hat{s}_{\sigma,\alpha} \in \widehat{S}$. Let $x \in \mathtt{dom}(\sigma(\alpha))$ be a program variable of a certain type and $\sigma' = \sigma \dagger \left\{\alpha \mapsto \sigma(\alpha) \dagger \{x \mapsto \widehat{\mathbf{I}}[\![t]\!](\sigma, \hat{s}_{\sigma,\alpha})\}\right\}$. Then for all terms t_0 and all terms t of this type, we have:

$$\widehat{\mathbf{I}}[\![t_0[t/x]]\!](\sigma, \hat{s}_{\sigma,\alpha}) = \widehat{\mathbf{I}}[\![t_0]\!](\sigma', \hat{s}_{\sigma',\alpha})$$

Proof: By induction on the structure of t_0.[5] □

Lemma 8 Let x be a variable of a certain type. Then for all formulas p, terms t of this type, states σ and assignments $\hat{s}_{\sigma,\alpha}$, we have, where σ' is as defined in Lemma 7:

$$\hat{s}_{\sigma',\alpha} \models p \text{ if and only if } \hat{s}_{\sigma,\alpha} \models p[t/x]$$

[5] The complete proof is given in Appendix A.

Proof: Direct from Lemma 7, using induction on the structure of p. □

Lemma 9 (Soundness of the Assignment Rule)

$$\models \{\, p \,\}\; x \leftarrow c \;\{\, q, x \,\}$$

on the assumption:

$$\models \{\, p \,\}\; c \;\{\, q[t/x], t \,\} \tag{12}$$

Proof: Assuming

$$\hat{s}_{\sigma,\alpha} \models p \tag{13}$$

and

$$\mathbf{I}[\![x \leftarrow c]\!](\alpha)(\gamma)(\sigma) = (\sigma'', \theta) \tag{14}$$

it needs to be shown that

$$\hat{s}_{\sigma'',\alpha} \models q$$

and

$$\widehat{\mathbf{I}}[\![x]\!](\sigma'', \hat{s}_{\sigma'',\alpha}) = \theta$$

We have (for some $\sigma' \in \Sigma$):

$\mathbf{I}[\![c]\!](\alpha)(\gamma)(\sigma) = (\sigma', \theta)$	(15)	From Assumption (14), by the semantics of commands (page 37)
$\sigma'' = \sigma' \dagger \{\alpha \mapsto \sigma'(\alpha) \dagger \{x \mapsto \theta\}\}$	(16)	By the semantics of commands (page 37)
$\hat{s}_{\sigma',\alpha} \models q[t/x]$	(17)	From Assumptions (12) and (13) and from (15), by Definition 20
$\hat{s}_{\sigma'',\alpha} \models q$		From (17) and (16), by Lemma 8
$\widehat{\mathbf{I}}[\![x]\!](s_{\sigma'',\alpha}, \sigma'') = \theta$		From (16), by the semantics of terms □

6.5 Message Sending

In this section we prove the soundness of the inference rules relative to the message sending command. Consider the following (cf. Section 4.5):

- ϱ a non-primitive type such that $v_1, \ldots, v_k$ is the list of variables of ϱ and m is a method name in the set of method names of ϱ;
- c_0 is the body of m and $fp_1, \ldots, fp_n$ the (list of) formal parameters of m;
- $c \equiv \texttt{self}.m(fp_1, \ldots, fp_n)$, where `self` has type ϱ.

6.5.1 Recursion Rule

Lemma 10

$$\varrho \models \{\, p \,\}\; c_0 \;\{\, q, t \,\}$$

on the assumption that, for all γ:

$$\gamma, \varrho \models \{\, p \,\}\; c_0 \;\{\, q, t \,\} \text{ if } \gamma, \varrho \models \{\, p \,\}\; c \;\{\, q, t \,\} \tag{18}$$

Proof: Consider a function assignment, say γ_1, such that:

$$\gamma_1, \varrho \not\models \{\, p \,\}\, c_0 \,\{\, q, t \,\} \tag{19}$$

Then consider γ_1' such that:

$$\gamma_1' = \gamma_1 \dagger \{(\varrho, m) \mapsto \bot_G\} \tag{20}$$

We have:

$\gamma_1' \models \{\, p \,\}\, c \,\{\, q, t \,\}$	From (20), by Definition 20 and the semantics of commands (page 38)
$\gamma_1' \not\models \{\, p \,\}\, c_0 \,\{\, q, t \,\}$	From (19) and (20), by a straightforward induction on the structure of c_0

That is, the negation of the conjecture implies the negation of the assumption. □

6.5.2 Invocation Rule

Lemma 11

$$\models \{\, p \,\}\, c \,\{\, q \,\}$$

on the assumption:

$$\models \{\, p \,\}\, c_0 \,\{\, q \,\}$$

Proof: Direct from Definition 20 and the fact that, by the semantics of commands, $\mathbf{I}[\![c]\!](\alpha)(\gamma)(\sigma) = \mathbf{I}[\![c_0]\!](\alpha)(\gamma)(\sigma)$, for all function assignments γ such that $\gamma(\varrho, m) \neq \bot_G$, all states σ and all non-primitive objects $\alpha \in A_\varrho \cap \mathtt{dom}(\sigma)$. □

6.5.3 Substitution Rule

Consider that:

- ϱ' is such that $e_0.m(e_1, \ldots, e_n)$ occurs in an object definition associated with it,
- ϱ is the type of e_0,
- $v_1', \ldots, v_r'$ is the list of variables of ϱ',
- $\sigma \in \Sigma$ and
- $\alpha' \in A_{\varrho'} \cap \mathtt{dom}(\sigma)$.

We use the following:[6]

Lemma 12 Let $\mathbf{I}[\![e_0]\!](\sigma(\alpha')) = \alpha$ and:

$$\sigma' = \sigma \dagger \left\{\alpha \mapsto \sigma(\alpha) \dagger \{fp_1 \mapsto \mathbf{I}[\![e_1]\!](\sigma(\alpha')), \ldots, fp_n \mapsto \mathbf{I}[\![e_n]\!](\sigma(\alpha'))\}\right\}$$

Then, for all terms t, all states σ, σ' and all assignments $\hat{s}_{\sigma,\alpha'}, \hat{s}_{\sigma',\alpha}$, we have:

[6] Lemma 12 characterizes states, namely σ and σ', and defines assignments relative to these states, namely $\hat{s}_{\sigma,\alpha'}$ and $\hat{s}_{\sigma',\alpha}$, with respect to which interpretations are equal, respectively, of those terms occurring in the precondition of the conclusion and in the precondition of the premise of the Substitution Rule. Lemma 12 is used in Lemma 13, which states that if the precondition of the conclusion is valid with respect to σ and $\hat{s}_{\sigma,\alpha'}$, then the precondition of the premise is valid with respect to σ' and $\hat{s}_{\sigma',\alpha}$. Lemma 14 and 15 proceed similarly with respect to the final state and relevant assignments relative to this state.

$$\widehat{\mathbf{I}}[\![t]\!](\sigma', \hat{s}_{\sigma',\alpha}) = \widehat{\mathbf{I}}[\![\bar{t}\,]\!](\sigma, \hat{s}_{\sigma,\alpha'})$$

where: $\bar{t} \equiv t^{\mathtt{self},\overline{fp},\bar{v}}_{e_0,\bar{e},\bar{e}_0}$
$\bar{e} \equiv e_1, \ldots, e_n$
$\bar{e}_0 \equiv e_0[1], \ldots, e_0[\underline{k}]$
$\overline{fp} \equiv fp_1, \ldots, fp_n$
$\bar{v} \equiv v_1, \ldots, v_k$

Proof: By induction on the structure of t.[7] □

Lemma 13 Let q be a formula, $\mathbf{I}[\![e_0]\!](\sigma(\alpha')) = \alpha$ and σ' as defined in Lemma 12. Then for all formulas p and all assignments $\hat{s}_{\sigma,\alpha'}, \hat{s}_{\sigma',\alpha}$, we have:

$$\text{if } \hat{s}_{\sigma,\alpha'} \models \bar{p} \text{ then } \hat{s}_{\sigma',\alpha} \models p$$

where $\bar{p} \equiv p_n$ and, for all $i = 1, \ldots, n$, we have:

$$p_i \equiv \begin{cases} p_{i-1} & \text{if } p_{i-1} \equiv e_i \doteq \nu_i \wedge p'', \text{ for some } p'', \nu_i, \text{ or} \\ & Fn(q) \cap \{fp_i\} = \emptyset, \text{ or } Fn(e_i) \cap \{v'_1, \ldots, v'_r\} = \emptyset \\ p_{i-1} \wedge e_i \doteq \nu_i & \text{otherwise, where } \nu_i \notin Fn(p_{i-1}) \end{cases}$$

$$p_0 \equiv \begin{cases} p' & \text{if } p' \equiv e_0 \doteq \nu' \wedge p'', \text{ for some } p'', \nu', \text{ or} \\ & Fn(q) \cap \{\mathtt{self}\} = \emptyset, \text{ or } Fn(e_0) \cap \{v'_1, \ldots, v'_r\} = \emptyset \\ p' \wedge e_0 \doteq \nu' & \text{otherwise, where } \nu' \notin Fn(p') \end{cases}$$

$$\begin{aligned} p' &\equiv p^{\mathtt{self},\overline{fp},\bar{v}}_{e_0,\bar{e},\bar{e}_0} \\ \bar{e} &\equiv e_1, \ldots, e_n \\ \bar{e}_0 &\equiv e_0[1], \ldots, e_0[\underline{k}] \\ \overline{fp} &\equiv fp_1, \ldots, fp_n \\ \bar{v} &\equiv v_1, \ldots, v_k \end{aligned}$$

Proof: By induction on the structure of p, using Lemma 12. □

Lemma 14 Let q be a formula and let $\hat{s}_{\sigma,\alpha}$ and $\hat{s}_{\sigma,\alpha'}$ be assignments such that $\widehat{\mathbf{I}}[\![e'_0]\!](\sigma, \hat{s}_{\sigma,\alpha'}) = \alpha$ and $\widehat{\mathbf{I}}[\![e'_i]\!](\sigma, \hat{s}_{\sigma,\alpha'}) = \widehat{\mathbf{I}}[\![fp_i]\!](\sigma, \hat{s}_{\sigma,\alpha})$, for all $i \in \{1, \ldots, n\}$, where:

$$e'_0 \equiv \begin{cases} e_0 & \text{if } Fn(e_0) \cap \{v'_1, \ldots, v'_r\} = \emptyset \\ \nu' & \text{otherwise, where } \nu' \text{ is such that } \widehat{\mathbf{I}}[\![\nu']\!](\sigma, \hat{s}_{\sigma,\alpha'}) = \widehat{\mathbf{I}}[\![e_0]\!](\sigma, \hat{s}_{\sigma,\alpha'}) \end{cases}$$

For all $i = 1, \ldots, n$, we have:

$$e'_i \equiv \begin{cases} e_i & \text{if } Fn(e_i) \cap \{v'_1, \ldots, v'_r\} = \emptyset \\ \nu_i & \text{otherwise, where } \nu_i \text{ is such that } \widehat{\mathbf{I}}[\![\nu_i]\!](\sigma, \hat{s}_{\sigma,\alpha'}) = \widehat{\mathbf{I}}[\![e_i]\!](\sigma, \hat{s}_{\sigma,\alpha'}) \end{cases}$$

Then for all terms t, we have:

$$\widehat{\mathbf{I}}[\![t]\!](\sigma, \hat{s}_{\sigma,\alpha}) = \widehat{\mathbf{I}}[\![\bar{t}\,]\!](\sigma, \hat{s}_{\sigma,\alpha'})$$

where:

$$\begin{aligned} \bar{t} &\equiv t^{\mathtt{self},\overline{fp},\bar{v}}_{e'_0,\bar{e}',\bar{e}'_0} \\ \bar{e}'_0 &\equiv e'_0[1], \ldots, e'_0[\underline{k}] \\ \bar{e}' &\equiv e'_1, \ldots, e'_n \end{aligned}$$

[7] Cf. footnote 4 on page 66.

Proof: By induction on the structure of t.[8] □

Lemma 15 Let q be a formula and $\hat{s}_{\sigma,\alpha}$, $\hat{s}_{\sigma,\alpha'}$ assignments as in Lemma 14. Then we have:

$$\text{if } \hat{s}_{\sigma,\alpha} \models q \text{ then } \hat{s}_{\sigma,\alpha'} \models \bar{q}$$

where:

$$\bar{q} \equiv q^{\mathtt{self},\overline{fp},\bar{v}}_{e_0,\bar{e},\bar{e}_0}$$

Proof: By induction on the structure of q, using Lemma 14. □

Lemma 16 (Soundness of the Substitution Rule)

$$\varrho' \models \{\,\bar{p}\,\}\, e_0.m(e_1,\ldots,e_n)\,\{\,\bar{q},\bar{t}\,\}$$

on the assumption:

$$\varrho \models \{\,p\,\}\, c\,\{\,q,t\,\} \tag{21}$$

where $\bar{p}$ is as in Lemma 13, $\bar{t}$ as in Lemma 14 and $\bar{q}$ as in Lemma 15.

Proof: Assuming

$$\hat{s}_{\sigma,\alpha'} \models \bar{p} \tag{22}$$

and

$$\mathbf{I}[\![e_0.m(e_1,\ldots,e_n)]\!](\alpha')(\gamma)(\sigma) = (\sigma'',\theta) \tag{23}$$

it needs to be shown that

$$\hat{s}_{\sigma'',\alpha'} \models \bar{q}$$

and

$$\widehat{\mathbf{I}}[\![\,\bar{t}\,]\!](\sigma'',\hat{s}_{\sigma'',\alpha'}) = \theta$$

We have, where $\alpha = \mathbf{I}[\![e_0]\!](\sigma(\alpha'))$ and σ' is as defined in Lemma 12:

$\hat{s}_{\sigma',\alpha} \models p$	(24)	From Assumption (22), by Lemma 13
$\mathbf{I}[\![e_0.m(e_1,\ldots,e_n)]\!](\alpha')(\gamma)(\sigma) =$ $\mathbf{I}[\![c]\!](\alpha)(\gamma)(\sigma')$	(25)	By the semantics of commands
$\hat{s}_{\sigma'',\alpha} \models q$	(26)	From Assumption (21) and from (24) and (25), by Definition 20
$\widehat{\mathbf{I}}[\![e_0']\!](\sigma'',\hat{s}_{\sigma'',\alpha'}) = \alpha$	(27)	From the following: • if $e_0' \equiv \nu'$ then $\widehat{\mathbf{I}}[\![\nu']\!](\sigma'',\hat{s}_{\sigma'',\alpha'}) = \widehat{\mathbf{I}}[\![\nu']\!](\sigma,\hat{s}_{\sigma,\alpha'}) = \mathbf{I}[\![e_0]\!](\sigma(\alpha')) = \alpha$. • if $e_0' \equiv e_0$ then $\widehat{\mathbf{I}}[\![e_0]\!](\sigma'',\hat{s}_{\sigma'',\alpha'}) = \mathbf{I}[\![e_0]\!](\sigma(\alpha')) = \alpha$.

For all $i \in \{1,\ldots,n\}$:

$\widehat{\mathbf{I}}[\![e_i']\!](\sigma'',\hat{s}_{\sigma'',\alpha'}) = \widehat{\mathbf{I}}[\![fp_i]\!](\sigma'',\hat{s}_{\sigma'',\alpha'})$	(28)	From the following, for all $i \in \{1,\ldots,n\}$: • if $e_i' \equiv \nu_i$ then $\widehat{\mathbf{I}}[\![\nu_i]\!](\sigma'',\hat{s}_{\sigma'',\alpha'}) = \widehat{\mathbf{I}}[\![\nu_i]\!](\sigma,\hat{s}_{\sigma,\alpha'}) = \widehat{\mathbf{I}}[\![e_i]\!](\sigma,\hat{s}_{\sigma,\alpha'}) = \widehat{\mathbf{I}}[\![fp_i]\!](\sigma',\hat{s}_{\sigma',\alpha'}) = \widehat{\mathbf{I}}[\![fp_i]\!](\sigma'',\hat{s}_{\sigma'',\alpha'})$. • if $e_i' \equiv e_i$ then $\widehat{\mathbf{I}}[\![e_i]\!](\sigma'',\hat{s}_{\sigma'',\alpha'}) = \widehat{\mathbf{I}}[\![e_i]\!](\sigma,\hat{s}_{\sigma,\alpha'}) = \widehat{\mathbf{I}}[\![fp_i]\!](\sigma',\hat{s}_{\sigma',\alpha'}) = \widehat{\mathbf{I}}[\![fp_i]\!](\sigma'',\hat{s}_{\sigma'',\alpha'})$.

[8] Cf. footnote 4 on page 66.

$\hat{s}_{\sigma'',\alpha'} \models \bar{q}$	(29)	From (26), (27) and (28), by Lemma 15
$\widehat{\mathbf{I}}[\![t]\!](s_{\sigma'',\alpha},\sigma'') = \theta$	(30)	From Assumption (21) and from (24) and (25), by Definition 20

$\widehat{\mathbf{I}}[\![t]\!](s_{\sigma'',\alpha},\sigma'') = \theta$ From (26), (27) and (28), by Lemma 14

□

6.5.4 Invariance Rule

Lemma 17 Consider that:

- $e_0.m(e_1,\ldots,e_n)$ occurs in an object definition associated to ϱ,
- e_0 has type ϱ and
- p is a formula such that $Fn(p) \subset Lv \cup \{\texttt{self}\}$

Then:

$$\models \{\, p \,\}\, e_0.m(e_1,\ldots,e_n)\, \{\, p \,\}$$

on the assumption:

$$\models \{\, p \,\}\, c\, \{\, p \,\} \tag{31}$$

Proof: Let $\sigma \in \Sigma$ and $\alpha' \in A_\varrho \cap \texttt{dom}(\sigma)$. Assuming

$$\hat{s}_{\sigma,\alpha'} \models p \tag{32}$$

and

$$\mathbf{I}[\![e_0.m(e_1,\ldots,e_n)]\!](\alpha')(\gamma)(\sigma) = (\sigma'',\theta) \tag{33}$$

it needs to be shown that

$$\hat{s}_{\sigma'',\alpha'} \models p$$

Let $\alpha = \mathbf{I}[\![e_0]\!](\sigma(\alpha'))$ and σ' as defined in Lemma 12.

Let, for any value $\hat{d}$ in the semantic domain:

$$\delta(\hat{d}) = \begin{cases} \alpha' & \text{if } \hat{d} = \alpha \\ \hat{d} & \text{otherwise} \end{cases}$$

Let, for any assignment $\hat{s}$, $\delta_1(\hat{s})$ denote the assignment obtained by considering, for all $x \in \texttt{dom}(\hat{s})$, $\delta_1(\hat{s})(x) = \delta\big(\hat{s}(x)\big)$.

Let, for any state σ, $\delta_2(\sigma)$ denote the state obtained by considering, for all $a \in \texttt{dom}(\sigma)$, $\delta_2(\sigma)(a) = \delta_1\big(\sigma(a)\big)$.

Let $\sigma_1 = \delta_2(\sigma')$ and $\sigma_2 = \delta_2(\sigma'')$.

We have:

For all terms t in p:

$\widehat{\mathbf{I}}[\![t]\!](\sigma,\hat{s}_{\sigma,\alpha'}) = \widehat{\mathbf{I}}[\![t]\!](\sigma_1',\hat{s}_{\sigma_1',\alpha})$	(34)	By induction on the structure of terms
$\hat{s}_{\sigma_1,\alpha} \models p$	(35)	From Assumption (32) and from (34), by the semantics of formulas
$\mathbf{I}[\![e_0.m(e_1,\ldots,e_n)]\!](\alpha')(\gamma)(\sigma) =$ $\mathbf{I}[\![c]\!](\alpha)(\gamma)(\sigma')$	(36)	By the semantics of commands

$$\mathbf{I}[\![c]\!](\alpha')(\gamma)(\sigma_1) = \big(\sigma_2, \delta(\theta)\big) \quad (37) \text{ From (36), by induction on the structure of commands (i.e. the body of } m)$$

$$\hat{s}_{\sigma_2,\alpha} \models p \quad (38) \text{ From Assumption (31) and from (35) and (37), by Definition 20}$$

For all terms t in p:

$$\widehat{\mathbf{I}}[\![t]\!](\sigma_2, \hat{s}_{\sigma_2,\alpha}) = \widehat{\mathbf{I}}[\![t]\!]\big(\sigma'', \hat{s}_{\sigma'',\alpha'}\big) \quad (39) \text{ Analogously as for (34)}$$

$$\hat{s}_{\sigma'',\alpha'} \models p \quad \text{From (38) and from (39), by the semantics of formulas}$$

□

6.6 Object Creation

In this section we prove the soundness of the axiom schemes for object creation.

6.6.1 First Axiom Scheme

Lemma 18

$$\models \{\, p_0 \,\}\ x \leftarrow odn!\mathtt{new}(e_1, \ldots, e_n)\ \{\, p \wedge x[\mathtt{1}] \doteq (e_1)^x_{t_0} \wedge \ldots \wedge x[\underline{n}] \doteq (e_n)^x_{t_0}, x \,\}$$

where: t_0 is a term such that $x \notin Fn(t_0)$, and

$$p_0 \equiv \begin{cases} \forall \nu : p[\nu/x] & \text{if } x \notin Fn(e_1, \ldots, e_n) \\ x \doteq t_0 \wedge \forall \nu : p[\nu/x] & \text{otherwise} \end{cases}$$

Proof: Assuming

$$\hat{s}_{\sigma,\alpha} \models p_0 \tag{40}$$

and

$$\mathbf{I}[\![x \leftarrow odn!\mathtt{new}(e_1, \ldots, e_n)]\!](\alpha)(\gamma)(\sigma) = (\sigma', \theta) \tag{41}$$

it needs to be shown that

$$\hat{s}_{\sigma',\alpha} \models p \wedge x[\mathtt{1}] \doteq (e_1)^x_{t_0} \wedge \ldots \wedge x[\underline{n}] \doteq (e_n)^x_{t_0} \tag{42}$$

and

$$\widehat{\mathbf{I}}[\![x]\!](\sigma', \hat{s}_{\sigma',\alpha}) = \theta$$

By the semantics of commands (page 38), we have:

$$\begin{aligned} \sigma' = \sigma\ &\dagger \big\{a \mapsto \{\mathtt{self} \mapsto a, v_1 \mapsto \mathbf{I}[\![e_1]\!](\sigma(\alpha)), \ldots, v_n \mapsto \mathbf{I}[\![e_n]\!](\sigma(\alpha))\}\big\} \\ &\dagger \big\{\alpha \mapsto \sigma(\alpha) \dagger \{x \mapsto a\}\big\} \end{aligned} \tag{43}$$

and $\theta = a$.

Thus, $\widehat{\mathbf{I}}[\![x]\!](\sigma', \hat{s}_{\sigma',\alpha}) = \theta$.

We now prove (42).

Consider:

- $x \notin Fn(e_1, \ldots, e_n)$; then:

$$p_0 \equiv \forall \nu : p[\nu/x] \tag{44}$$

Consider $\nu_0 \in Lv$ such that:

$$\hat{s}_{\sigma,\alpha}(\nu_0) = \hat{s}_{\sigma',\alpha}(x) \tag{45}$$

We have:

$\hat{s}_{\sigma,\alpha} \models p[\nu_0/x]$ (46) From Assumption (40) and from (44), by the semantics of formulas and Definition 19

$\widehat{\mathbf{I}}[\![t[\nu_0/x]]\!](\sigma, \hat{s}_{\sigma,\alpha}) = \widehat{\mathbf{I}}[\![t]\!](\sigma', \hat{s}_{\sigma',\alpha})$ (47) From (43) and (45), by induction on the structure of t

$\hat{s}_{\sigma',\alpha} \models p$ (48) From (46) and (47), by induction on the structure of p

For all $i \in \{1, \ldots, n\}$:

$(e_i)^x_{\nu_0} \equiv e_i$ (49) From $x \notin Fn(e_i)$, for all $i \in \{1, \ldots, n\}$, and from the definition of textual substitution (Definition 13), by induction on the structure of e_i

For all $i \in \{1, \ldots, n\}$:

$\hat{s}_{\sigma',\alpha} \models x[\underline{i}] \doteq (e_i)^x_{\nu_0}$ (50) From (49) and (43), by the semantics of terms and formulas and Definition 19

(42) From (50) and (48), by the semantics of formulas and Definition 19

- $x \in Fn(e_1, \ldots, e_n)$; then:

$$p_0 \equiv x \doteq t_0 \wedge \forall \nu : p[\nu/x] \tag{51}$$

We have:

$\hat{s}_{\sigma',\alpha} \models p$ (52) By the same proof as in the case above

$\widehat{\mathbf{I}}[\![x]\!](\sigma, \hat{s}_{\sigma,\alpha}) = \widehat{\mathbf{I}}[\![t_0]\!](\sigma, \hat{s}_{\sigma,\alpha})$ (53) From Assumption (40) and from (51), by the semantics of formulas and Definition 19

$\widehat{\mathbf{I}}[\![t_0]\!](\sigma, \hat{s}_{\sigma,\alpha}) = \widehat{\mathbf{I}}[\![t_0]\!](\sigma, \hat{s}_{\sigma',\alpha})$ (54) From (43) and the fact that $x \notin Fn(t_0)$ (and the fact that $\hat{s}_{\sigma,\alpha}(\nu) = \hat{s}_{\sigma',\alpha}(\nu)$, for all $\nu \in Lv$), by induction on the structure of t_0

For all $i \in \{1, \ldots, n\}$:

$\widehat{\mathbf{I}}[\![(e_i)^x_{t_0}]\!](\sigma, \hat{s}_{\sigma,\alpha}) = \widehat{\mathbf{I}}[\![(e_i)^x_{t_0}]\!](\sigma', \hat{s}_{\sigma',\alpha})$
(55) From (54), by induction on the structure of e_i

For all $i \in \{1, \ldots, n\}$:

$\widehat{\mathbf{I}}[\![(e_i)^x_{t_0}]\!](\sigma, \hat{s}_{\sigma,\alpha}) = \widehat{\mathbf{I}}[\![e_i]\!](\sigma, \hat{s}_{\sigma,\alpha})$ (56) From (53), by Lemma 1

For all $i \in \{1, \ldots, n\}$:

$\hat{s}_{\sigma',\alpha} \models x[\underline{i}] \doteq (e_i)^x_{\nu_0}$ (57) From (56), (55) and (43), by the semantics of terms and formulas and Definition 19

(42) From (57) and (52), by the semantics of formulas and Definition 19

□

6.6.2 Second Axiom Scheme

Lemma 19

$$\models \{\, \forall \nu : p[\nu/x] \,\}\; x \leftarrow odn!\mathtt{new}(e_1, \ldots, e_n)\; \{\, p \wedge t \not\equiv x, x \,\}$$

where t is a term such that $x \notin Fn(t)$ and $Fn(t) \cap Lv = \emptyset$

Proof: Assuming

$$\hat{s}_{\sigma,\alpha} \models \forall \nu : p[\nu/x]$$

and

$$\mathbf{I}[\![x \leftarrow odn!\mathtt{new}(e_1, \ldots, e_n)]\!](\alpha)(\gamma)(\sigma) = (\sigma', \theta)$$

it needs to be shown that

$$\hat{s}_{\sigma',\alpha} \models p \wedge t \not\equiv x$$

and

$$\widehat{\mathbf{I}}[\![x]\!](\sigma', \hat{s}_{\sigma',\alpha}) = \theta$$

As in Lemma 18 above, by the semantics of commands (page 38) we have (43) and $\theta = a$, and thus $\widehat{\mathbf{I}}[\![x]\!](\sigma', \hat{s}_{\sigma',\alpha}) = \theta$.

We have:

$\hat{s}_{\sigma',\alpha} \models p$	(58)	By the same proof as in Lemma 18
$t \not\equiv x$	(59)	From (43) and the fact that $a \notin \mathtt{dom}(\sigma)$, by induction on the structure of t
$\hat{s}_{\sigma',\alpha} \models p \wedge t \not\equiv x$		From (58) and (59), by the semantics of formulas and Definition 19

6.7 Rule for Sequential Composition

Lemma 20

$$\models \{\, p \,\}\; c_1; c_2\; \{\, q, t \,\}$$

on the assumptions that:

$$\models \{\, p \,\}\; c_1\; \{\, p' \,\}$$

and

$$\models \{\, p' \,\}\; c_2\; \{\, q, t \,\}$$

Proof: Direct from Definitions 20 and 19, and the semantics of commands (page 38). □

6.8 Rule for If-Then-Else

Lemma 21

$$\models \{\, p \,\}\; \mathtt{if}\; c_0\; \mathtt{then}\; c_1\; \mathtt{else}\; c_2\; \{\, q \,\}$$

on the assumptions that:

$$\models \{\, p \,\}\; c_0\; \{\, p', t \,\}$$

and

$$\models \{\, p' \wedge\; t \,\}\; c_1\; \{\, q \,\}$$

and

$$\models \{\, p' \wedge\; \neg t \,\}\; c_2\; \{\, q \,\}$$

Proof: Straightforward, by Definitions 20 and 19, the semantics of commands (page 38) and the semantics of formulas and correctness formulas. □

6.9 Rule for Blocks

Lemma 22

$$\models \{\, p \,\}\; tp_1 : \tau_1, \ldots, tp_n : \tau_n \; c \,\{\, q, t \,\}$$

on the assumption that:

$$\models \{\, p \wedge tp_1 \doteq \texttt{nil} \wedge \ldots \wedge tp_n \doteq \texttt{nil} \,\}\; c \,\{\, q, t \,\}$$

Proof: Straightforward, by Definitions 20 and 19, the semantics of commands (page 38) and blocks (Definition 5), and the semantics of formulas and correctness formulas. □

6.10 Invariance Axiom scheme

Lemma 23

$$\{\, p \,\}\; c \,\{\, p \,\}$$

where p is such that:

1. c occurs in an object definition ϱ such that, for all $x \in Fn(p)$, $(\varrho, x, V) \cap Chg(c) = \emptyset$, if $x \in V$, and $(\varrho, x, Tp) \cap Chg(c) = \emptyset$, if $x \in Tp$.
2. p has no indexed terms, i.e. for no terms t and i, $t[i]$ occurs in p;
3. no term $f(t_1, \ldots, t_n)$ occurs in p where $n \geq 1$ and for some i between 1 and n, t_i has a non-primitive type.

Proof: Straightforward, by Definitions 20 and 19, the semantics of commands (page 38) and blocks (Definition 5), and the semantics of formulas and correctness formulas. □

6.11 Conjunction Rule

Lemma 24

$$\models \{\, p \wedge p' \,\}\; c \,\{\, q \wedge q', t \,\}$$

on the assumptions that:

$$\models \{\, p \,\}\; c \,\{\, q, t \,\}$$

and

$$\models \{\, p' \,\}\; c \,\{\, q' \,\}$$

Proof: Straightforward, by Definitions 20 and 19, and the semantics of formulas and correctness formulas. □

6.12 Rule of Result Elimination

Lemma 25

$$\models \{\, p \,\}\; c \,\{\, q \,\}$$

on the assumption that:

$$\{\, p \,\}\; c \,\{\, q, t \,\}$$

Proof: Direct from Definitions 20 and 19 and the semantics of formulas and correctness formulas. □

6.13 Rule of Result Substitution

Lemma 26

$$\models \{ p \} \, c \, \{ q, t' \}$$

on the assumptions that:

$$\{ p \} \, c \, \{ q, t \}$$

and

$$\models q \Rightarrow t \doteq t'$$

Proof: Straightforward, by Definitions 20 and 19 and the semantics of formulas and correctness formulas. □

6.14 Rule of Logical Variable Elimination

Lemma 27

$$\models \{ p^{\nu}_{t'} \} \, c \, \{ q, t \}$$

on the assumption:

$$\models \{ p \} \, c \, \{ q, t \} \tag{60}$$

Proof: Straightforward, from Definitions 20 and 19 and the semantics of formulas and correctness formulas, by considering an assignment $\hat{s}_{\sigma,\alpha}$, relative to an initial state σ and a non-primitive object α, such that $\hat{s}_{\sigma,\alpha}(\nu) = \widehat{\mathbf{I}}[\![t']\!](\sigma, \hat{s}_{\sigma,\alpha})$. □

6.15 Soundness Theorem

We thus have the following:

Theorem 2 (Soundness of P1) Let X be a set of formulas or correctness formulas valid in $\widehat{\mathbf{I}}$. Then, for all correctness formulas $w \equiv \{ p \} \, c \, \{ q, t \}$, we have:

$$\text{if } X \vdash \{ p \} \, c \, \{ q, t \} \text{ then } \models \{ p \} \, c \, \{ q, t \}$$

Proof: Straightforward, from Lemmas 6.1 to 6.14, by induction on the length of the construction sequence for w. □

Chapter 7

Conclusions and Further Work

Object-based programs in particular, and programs that dynamically change data structures in general, construct data structures with certain (dynamic) properties; furthermore, they have parts that assume that the manipulated data structures have those properties. (A typical example is that of requiring a data structure to be acyclic.) We think that this forms the origin of significant additional difficulties in this area, with respect to Hoare-style proof systems. The ease with which we can work with a proof system is basically due to the ease with which the specification language for that proof system can be used to express such properties. We think that indexing, together with quantification over indices, as in **L1**, provides a natural and relatively simple way of expressing those properties.

Another significant source of difficulties for proof systems in this area is due to name clashing: i.e. the fact that we can have the same name for distinct variables (in the case of **P1**, the same name for variables associated with different objects). Here **P1**'s solution is based, in general, on the reverse situation of aliasing: i.e. on having distinct names for the same variable. This allows, for example, "renaming" the object variable x to `self`$[i]$, where i is the index of x in the list of variables of the type of `self`.

The proof system presented in this thesis has been applied to show the correctness of object-based programs, which are of significant complexity when comparing with examples given in proofs used in connection with imperative programming languages supporting procedure calls[1], despite difficulties specific to the area (as mentioned above). The examples contain elements that illustrate complexities involved in general in proofs of larger programs.

The proof system has introduced original modifications of, or adaptations to, rules used in more traditional Hoare-style proof systems, which have enabled their use and/or helped to simplify formal correctness proofs, such as modifications and adaptations involving aspects related to procedure calls, values returned by commands, instantiation of logical variables, invariance and blocks.

There is much work still to be done and, we expect, innovations to come. We comment next on extending **P1** to prove total correctness, and also on completeness proofs and concurrency.

7.1 Total Correctness

We think that an extension of **P1** to prove total correctness of programs, which additionally guarantees (with respect to partial correctness) termination of programs, is facilitated by the fact that the recursion rule is separated from the substitution, invariance and invocation rules in **P1**, but necessitates awareness of complications due to the interaction of the recursion rule and the rules which formalise reasoning about invariance properties.[2]

Sokołowski[Sok77] presented in 1977 a system, and in particular a rule, for proving total correctness of programs with recursive parameterless procedure calls. Apt[Apt81] showed that such a system did

[1] This is particularly true when comparing with languages supporting recursive procedure calls and parameters.

[2] Cf. [AdB90b]. (See also [Fra92, Section 6.3.1].)

not allow the proof of certain invariance properties of procedure calls[3]. America and de Boer[AdB90b] showed that the extended system proposed by Apt (for total correctness of parameterless procedures) is not sound (one can derive from it correctness formulas which are not valid). They proposed a modified system with restrictions defined on the use of logical variables which are free in the assumption of the recursion rule (i.e. a system with no substitution or elimination of those variables).[4] We think that similar restrictions can be used in an extension of **P1** for proving total correctness, although this is an area for which alternative new developments are particularly encouraged.[5]

7.2 Completeness

We comment in this section about further work regarding the question of completeness, beginning with some general comments on the notion of completeness and including a brief description of a technique used to give completeness proofs of Hoare-style proof systems.

Informally speaking, one would usually state that a proof system is *complete* if every valid correctness formula is provable in that system. ("Valid" is used here as meaning "true with respect to all interpretations".) This definition is not adequate for Hoare-style proof systems, as we see below.

As an initial consideration, recall that Hoare-style proof systems generally present the Consequence Rule but do not include proof rules for proving implications (used in the premise of this rule).[6] Let now $X \models p$ abbreviate "$\models_{\mathbf{I}} p$ for every model $\mathbf{I}$ of X" (in this case, p is called a *logical consequence* of X). More significant than our initial consideration is the fact that for all interpretations which are models of the theory[7] of (Peano) arithmetic (as a consequence of Gödel's Incompleteness Theorem[8]) there is no extension of a Hoare-style proof system which is complete according to the above definition.

Let $Th_{\mathbf{I}}$ be the set of all formulas that are valid in interpretation $\mathbf{I}$. ("Valid" is used here as meaning "true with respect to all assignments".) The notion of completeness is defined as follows.

Definition 23 (Completeness) We call a Hoare-style proof system **P** *complete* with respect to a specification language **S**, programming language **L** and interpretation **I** if $\models_{\mathbf{I}} \{\,p\,\}\,c\,\{\,q\,\}$ implies $Th_{\mathbf{I}} \vdash \{\,p\,\}\,c\,\{\,q\,\}$, for all formulas p, q of **S** and commands c of **L**.

In the case of arithmetical interpretations, this definition amounts to having an oracle for truth, since (by Gödel's Incompleteness Theorem) there is no proof system such that the set of its theorems is equal to the set of valid formulas of (Peano) arithmetic (the latter set is not recursively enumerable). However, in this way concerns about programs can be separated from concerns about the incompleteness of proof systems for arithmetic.[9]

A notion of expressiveness is often used in proofs of completeness of Hoare-style proof systems. This notion is defined in terms of the weakest liberal precondition (or, alternatively, of the strongest postcondition) of a given command and a given predicate (see Section 1.2.4, page 7).

[3] I.e. properties establishing that the initial value of a variable that is used in the procedure equals its final value.

[4] Informally speaking, these free variables give, in the recursion rule, an upper bound to the number of nested calls necessary for the termination of the procedure.

[5] Note also that the invariance axiom scheme is not valid for total correctness, but can be substituted for the following invariance rule:

$$\frac{\{\,p'\,\}\,c\,\{\,q, t\,\}}{\{\,p' \wedge p\,\}\,c\,\{\,q \wedge p\,\}}$$

where p is as in the invariance axiom scheme of **P1**.

[6] Thus, for example, $\{\,\texttt{true}\,\}\,x \leftarrow \texttt{1}\,\{\,x \doteq \texttt{1}\,\}$ is true in all interpretations but not provable, since there are no rules for proving $\texttt{true} \Rightarrow \texttt{1} \doteq \texttt{1}$.

[7] A *theory* is a non-empty subset X of formulas for which there exists at least one model and which is closed under logical consequence (i.e. for which if $X \models p$, then $p \in X$).

[8] See e.g. [Win93, Chapter 7][End72].

[9] We could use Definition 23 and define a proof system **P** to be complete, with respect to given programming and specification languages, if it is complete with respect to those programming and specification languages and all interpretations. However, consider now interpretations that are not rich enough, for which the set of valid formulas is a recursive set (such as the usual interpretation for Presburger arithmetic, i.e. Peano arithmetic without multiplication). For these interpretations, we have again (by the undecidability of the halting problem) that there can be no Hoare-style proof system such that the set of its theorems is equal to the set of valid formulas.

Definition 24 A specification language **S** is said to be *expressive* with respect to a programming language **L** and an interpretation **I** if for all commands c (of **L**) and all formulas q (of **S**) there is a formula p (of **S**) such that its meaning $\mathbf{I}[\![p]\!]$ is the weakest liberal precondition of c and $\mathbf{I}[\![q]\!]$. In this case $\mathbf{I}[\![p]\!]$ is said to be expressible in **S**.

The standard technique for proving expressibility is based on coding of a finite sequence of natural numbers by a pair of natural numbers (called Gödelization). This is used to express the weakest liberal precondition as a formula of the specification language. The weakest liberal precondition is obtained, in the case of iterative constructs, by "unwinding" the iterations. The Gödelization is used to encode the existential quantification over the number of such iterations.

We think that this technique can be applied for proving expressibility of **L1** with respect to **A1** and interpretation $\widehat{\mathbf{I}}$. It is expected that this will require the use of more involved predicates and codings[10].

The standard technique for proving completeness with respect to a specification language **S**, programming language **L** and interpretation **I** uses the fact that the specification language is expressive with respect to **L** and **I** and, roughly speaking, uses a correctness formula $w_0 \equiv \{\, p_0 \,\}\, c_0 \,\{\, q_0 \,\}$, called *most general correctness formula*, such that:

1. p_0 is such that it "freezes" the initial state (usually this is done simply by equating program variables occurring in c_0 to logical variables);[11]
2. q_0 is such that $\mathbf{I}[\![q_0]\!]$ is the strongest postcondition of c and $\mathbf{I}[\![p]\!]$.

The completeness proof is done in two steps, using the most general correctness formula w_0:

1. If $\models_{\mathbf{I}} \{\, p \,\}\, c \,\{\, q \,\}$, then $Th_{\mathbf{I}} \cup \{w_0\} \vdash \{\, p \,\}\, c \,\{\, q \,\}$;
2. $Th_{\mathbf{I}} \vdash w_0$.

The first is proved by induction on the structure of c and the second is immediate from the first and the recursion rule.

We think that a proof of completeness of **P1** with respect to a **L1**, **A1** and interpretation $\widehat{\mathbf{I}}$ can be based on this technique. The proofs are in general expected to be involved, but use the same general ideas of proofs used in the case of imperative languages with procedures.[12]

7.3 Concurrency

The study and development of proof systems for concurrent object-based languages and, moreover, the study and development of practical development methods for concurrent object-based programs are major, difficult tasks which are outside the scope of this thesis.

However, we would like here to suggest some further work related to a program development method possible for a small class of concurrent object-based programs, i.e. those which are free of interference, based on the transformation of (provably correct) sequential ones.[13]

The extension of **A1** with a return command, to allow the result of a method call to be given before the end of that call[14], is proposed as an initial step for the introduction of concurrency. For simplicity, allowing only one active thread of execution for non-primitive objects at a time is suggested for this initial step.[15]

[10] See [AdB90a, Section 6.2, Appendix B].

[11] It is clear that the definition of precondition p_0, in a proof system for an object-based language, is more complicated than in the case of simple imperative languages, due to name clashing, i.e. due to the fact that we can have the same name for distinct variables (variables associated with distinct objects).

[12] See [Apt81, AdB90a].

[13] The following has been inspired by the work on concurrent object-oriented languages in general (see Section 1.3.3) and in particular by the work of Cliff Jones on a design method for concurrent object-based programs[Jon93b].

[14] Cf. item 3, page 18.

[15] Unfortunately, as a consequence of this restriction not all correct sequential programs are correct concurrent ones, since a message sent by an object to itself causes more than one thread of execution for that object to be active at a time. (This occurs for most concurrent object-oriented languages.)

The denotational semantics of such a language, called **A2**, can be defined by the method of semantics definition outlined in Appendix C. This appendix includes a denotational semantics for a concurrent object-based language with such a return command and synchronous and asynchronous message sending commands.

The semantics of **A2** is intended to be such that for a sequential program pg in **A1** for which it is assumed that there exist no messages sent by an object to itself, pg is also a program in **A2**,[16] and the meanings given by the two semantic definitions are the same (in particular, although parallelism is involved in the semantics definition of **A2**, the semantics of **A2** is intended to be such that the execution of pg is deterministic).

Consider now a command C and an expression e of **A1** such that the following correctness formulas are provable in **P1**:

$$\{\, e \doteq E \,\}\, c \,\{\, e \doteq E \,\} \tag{1}$$

and:

$$\{\, t \doteq X \,\}\, c \,\{\, t \neq X \Rightarrow (\exists! k : \mathtt{self}[k] \doteq t) \,\} \tag{2}$$

where E, X, k are logical variables, t is a program variable of non-primitive type or an indexed term of non-primitive type whose index is a constant term and $\exists!$ expresses unique existence[17].

Informally, equation 1 establishes that c does not change e and equation 2 that c only changes "private" variables (i.e. program variables of non-primitive type denoting values that are not denoted by other program variables).

Using the semantics of **A2** and the way it is related to the semantics of **A1** and properties inferred from equations 1 and 2 by the semantics of correctness formulas and of **L1**, it is expected that it can be shown that the meaning of an **A2** program pg, obtained by a transformation of a sequential **A1** program as mentioned above, is not changed by the following transformation:

$$c;\ \mathtt{return}\, e \ \rightsquigarrow\ \mathtt{return}\, e;\ c$$

We think this study can provide good foundations for further extensions and similar transformation rules, such as those related to the transformation of a synchronous message sending command into an asynchronous one. It is also expected that this work can provide new insights on the study of semantics definitions for concurrent programming languages, and also on more systematic developments of concurrent programs which allow some form of interference.

It is also expected that this work would be relevant for comparisons between the use of different formalisms as a basis for reasoning about the semantics of programs, specifically for comparisons between denotational semantics and the approach of giving the semantics by translation to a process calculus, currently being used by Cliff Jones for the semantics of $\pi o \beta \lambda$ and for reasoning about the above mentioned transformation.[18]

[16] Or a program in **A2** can be derived from pg where each command c; e which is the body of a method in pg is replaced by c; $\mathtt{return}\, e$.

[17] I.e. $\exists! k : p(k)$ is an abbreviation for $\exists k : p(k) \wedge \forall k' : p(k') \Rightarrow k' \doteq k$, where for any x, $p(x)$ denotes a formula such that $fn(p) = \{x\}$.

[18] See [Jon93b, Jon93a].

Appendix A

Proof by Induction on the Structure of Terms

This appendix presents an illustrative example of a proof by induction on the structure of terms of **L1**. Other proofs by induction on the structure of terms used in this thesis are analogous.

We re-state and prove:

Lemma 7 Let $\sigma \in \Sigma$ and $\alpha \in A_\varrho$, for some type ϱ. Let $x \in \mathtt{dom}\big(\sigma(\alpha)\big)$ be a program variable of a certain type and $\sigma' = \sigma \dagger \big\{\alpha \mapsto \sigma(\alpha) \dagger \{x \mapsto \widehat{\mathbf{I}}[\![t]\!](\sigma, \hat{s}_{\sigma,\alpha})\}\big\}$. Then for all terms t_0 and all terms t of this type, we have:

$$\widehat{\mathbf{I}}[\![t_0[t/x]]\!](\sigma, \hat{s}_{\sigma,\alpha}) = \widehat{\mathbf{I}}[\![t_0]\!](\sigma', \hat{s}_{\sigma',\alpha}) \tag{1}$$

Proof: The proof is by induction on the structure of t_0.

In the case that the variable to be substituted for (namely x) is an object variable, substitution is defined by means of substitution in arguments, which is also defined by induction on the structure of terms; therefore we prove simultaneously (when x is an object variable) that:

$$\widehat{\mathbf{I}}_{\mathbf{x}}[\![t_0[x \rightsquigarrow t]]\!](\sigma, \hat{s}_{\sigma,\alpha}) = \widehat{\mathbf{I}}_{\mathbf{x}}[\![t_0]\!](\sigma', \hat{s}_{\sigma',\alpha}) \tag{2}$$

In this case, we let x be the j-th variable in the list of variables of (say) ϱ.

In the following we write, for example:

$$\begin{aligned} X &= Y && \text{From } \ldots, \text{ by } \ldots \\ &= Z && \text{From } \ldots, \text{ by } \ldots \end{aligned}$$

as an abbreviation of:

$$\begin{aligned} X &= Y && \text{From } \ldots, \text{ by } \ldots \\ Y &= Z && \text{From } \ldots, \text{ by } \ldots \end{aligned}$$

Let also $u = (\sigma, \hat{s}_{\sigma,\alpha})$ and $u' = (\sigma', \hat{s}_{\sigma',\alpha})$.

1. $t_0 \equiv x$; then:

   - (1), by:

$$\begin{aligned} \widehat{\mathbf{I}}[\![t_0[t/x]]\!](u) &= \widehat{\mathbf{I}}[\![t]\!](u) && \text{By Definition 15} \\ &= \widehat{\mathbf{I}}[\![x]\!](u') && \text{By the semantics of terms and the definition of } \hat{s}_{\sigma',\alpha} \end{aligned}$$

- (2), by considering, where ϱ_0 is the type of x:
  - $typ(\varrho_0) \supseteq \{\varrho\}$; then:

$$\begin{aligned}\widehat{\mathbf{I}}[\![t_0[t/x]]\!](u) &= \widehat{\mathbf{I}}[\![t_0[\mathtt{self}, \underline{j}, t]]\!](u) && \text{By Definition 16}\\ &= \widehat{\mathbf{I}}[\![t_0]\!](u') && \text{By the semantics of terms}\end{aligned}$$

  - otherwise:

$$\begin{aligned}\widehat{\mathbf{I}}[\![t_0[t/x]]\!](u) &= \widehat{\mathbf{I}}[\![t]\!](u) && \text{By Definition 16}\\ &= \widehat{\mathbf{I}}[\![x]\!](u') && \text{From } typ(\varrho_0) \not\supseteq \{\varrho\}\text{, by the semantics of terms}\end{aligned}$$

2. $t_0 \equiv y$ ($y \not\equiv x$, y a variable, formal parameter or `self`); then:

- (1), by:

$$\begin{aligned}\widehat{\mathbf{I}}[\![t_0[t/x]]\!](u) &= \widehat{\mathbf{I}}[\![y]\!](u) && \text{By Definition 15}\\ &= \widehat{\mathbf{I}}[\![y]\!](u') && \text{By the semantics of terms and the definition of } \hat{s}_{\sigma',\alpha}\end{aligned}$$

- (2), by considering, where ϱ_0 is the type of y:
  - $typ(\varrho_0) \supseteq \{\varrho\}$; then the proof is the same as for $t_0 \equiv x$;
  - otherwise:

$$\begin{aligned}\widehat{\mathbf{I}}_{\mathbf{x}}[\![t_0[x \rightsquigarrow t]]\!](u) &= \widehat{\mathbf{I}}_{\mathbf{x}}[\![y]\!](u) && \text{By Definition 16}\\ &= \widehat{\mathbf{I}}_{\mathbf{x}}[\![y]\!](u') && \text{From } typ(\varrho_0) \not\supseteq \{\varrho\}\text{, by the semantics of terms and definitions of } \sigma' \text{ and } \hat{s}_{\sigma',\alpha}\end{aligned}$$

3. t_0 is either `true`, `false`, `nil` or an integer constant; then:

- (1), by:

$$\begin{aligned}\widehat{\mathbf{I}}[\![t_0[t/x]]\!](u) &= \widehat{\mathbf{I}}[\![t_0]\!](u) && \text{By Definition 15}\\ &= \mathcal{I}(t_0) && (3)\ \text{By the semantics of terms}\\ &= \widehat{\mathbf{I}}[\![t_0]\!](u') && (4)\ \text{By the semantics of terms}\end{aligned}$$

- (2), by:

$$\begin{aligned}\widehat{\mathbf{I}}_{\mathbf{x}}[\![t_0[x \rightsquigarrow t]]\!](u) &= \widehat{\mathbf{I}}_{\mathbf{x}}[\![t_0]\!](u) && \text{By Definition 16}\\ &= \widehat{\mathbf{I}}_{\mathbf{x}}[\![t_0]\!](u') && \text{From (3), (4) and } \mathcal{I}(t_0) \notin A\text{, by definition of } \widehat{\mathbf{I}}_{\mathbf{x}}[\![t_0]\!]\end{aligned}$$

4. $t_0 \equiv e_1 \doteq e_2$; then:

- (1), by considering (where $t' \equiv e_1[t/x] \doteq e_2[t/x]$):
  - $\widehat{\mathbf{I}}[\![e_i[t/x]]\!](u) \neq \varepsilon$, for $i = 1, 2$; then:

$$\widehat{\mathbf{I}}[\![e_i]\!](u') \neq \varepsilon \qquad (5)\ \text{For } i = 1,2\text{, from } \widehat{\mathbf{I}}[\![e_i[t/x]]\!](u) \neq \varepsilon \text{ for such } i\text{, by induction hypothesis}$$

$\widehat{\mathbf{I}}[\![t_0[t/x]]\!](u) = \widehat{\mathbf{I}}[\![t']\!](u)$	By Definition 15
$= \big(\widehat{\mathbf{I}}[\![e_1[t/x]]\!](u) = \widehat{\mathbf{I}}[\![e_2[t/x]]\!](u)\big)$	From $\widehat{\mathbf{I}}[\![e_i[t/x]]\!](u) \neq \varepsilon$, for $i = 1, 2$, by the semantics of terms (and expressions)
$= \big(\widehat{\mathbf{I}}[\![e_1]\!](u') = \widehat{\mathbf{I}}[\![e_2]\!](u')\big)$	By induction hypothesis
$= \widehat{\mathbf{I}}[\![e_1 \doteq e_2]\!](u')$	From (5), by the semantics of terms (and expressions)

– $\widehat{\mathbf{I}}[\![e_i[t/x]]\!](u) = \varepsilon$, for some $i \in \{1, 2\}$; then:

$\widehat{\mathbf{I}}[\![e_i]\!](u') = \varepsilon$	(6)	For some $i \in \{1, 2\}$, from $\widehat{\mathbf{I}}[\![e_i[t/x]]\!](u) = \varepsilon$ for such i, by induction hypothesis

$\widehat{\mathbf{I}}[\![t_0[t/x]]\!](u) = \widehat{\mathbf{I}}[\![t']\!](u)$	By Definition 15
$= \varepsilon$	From $\widehat{\mathbf{I}}[\![e_i[t/x]]\!](u) = \varepsilon$, for some $i \in \{1, 2\}$, by the semantics of terms (and expressions)
$= \widehat{\mathbf{I}}[\![e_1 \doteq e_2]\!](u')$	From (6), by the semantics of terms (and expressions)

- (2), analogously (by considering, respectively, $\widehat{\mathbf{I}}_{\mathbf{x}}[\![e_i[x \rightsquigarrow t]]\!](u) \neq \varepsilon$, for $i = 1, 2$, and $\widehat{\mathbf{I}}_{\mathbf{x}}[\![e_i[x \rightsquigarrow t]]\!](u) = \varepsilon$, for some $i \in \{1, 2\}$).

5. $t_0 \equiv \mathtt{if}\, e\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2$; the proof is similar to that given in item 4 above;

6. $t_0 \equiv e_0.m(e_1, \ldots, e_n)$; again, the proof is similar to that given in item 4 above;

7. $t_0 \equiv t_1[k]$; if x is not an object variable, then for any formula p and any term t, the substitution $p[t/x]$ is defined as equivalent to the textual substitution p^x_t, and the proof is direct, from the semantics of terms and the induction hypothesis; if x is an object variable, we have:

   - (1), by considering:
     – $t_1 \equiv \mathtt{nil}$; then:

$\widehat{\mathbf{I}}[\![t_0[t/x]]\!](u) = \widehat{\mathbf{I}}[\![\mathtt{nil}[k]]\!](u)$	By Definition 15
$= \mathit{error}$	By the semantics of terms (expressions)
$= \widehat{\mathbf{I}}[\![\mathtt{nil}[k]]\!](u')$	By the semantics of terms (expressions)

     – otherwise, let τ_1 the type of t_1, n the length of the list of variables of τ_1, $t'_1 \equiv t_1[t/x]$, $k' \equiv k[t/x]$ and consider:
       * k is a constant term; then, letting $k \equiv \underline{i}$, we have:
       (a) if $i > n_0$, then, letting $t' \equiv (t_1[k\, \mathtt{div}\, \underline{n}_0])[k\, \mathtt{mod}\, \underline{n}_0]$, we have:

$\widehat{\mathbf{I}}[\![t_0[t/x]]\!](u) = \widehat{\mathbf{I}}[\![t'[t/x]]\!](u)$	By Definition 15

$\widehat{\mathbf{I}}[\![t']\!](u')$ From the fact that the substitution continues until all indices are less than n_0, by the induction hypothesis and the proofs below (when k is a constant term)

$= \widehat{\mathbf{I}}[\![t_1[k]]\!](u')$ By the semantics of terms

(b) if $\tau_1 \equiv \varrho$ and $i \equiv j$, then, letting

$$t' \equiv \ \texttt{if}\ t'_1 \doteq \texttt{self}\ \texttt{then}\ t\ \texttt{else}\ t'_1[k]$$

and

$$t'' \equiv \ \texttt{if}\ t_1 \doteq \texttt{self}\ \texttt{then}\ x\ \texttt{else}\ t_1[k]$$

we have:

$\widehat{\mathbf{I}}[\![t_0[t/x]]\!](u) = \widehat{\mathbf{I}}[\![t']\!](u)$ By Definition 15

$= \widehat{\mathbf{I}}[\![t'']\!](u')$ By the semantics of terms and the induction hypothesis

$= \widehat{\mathbf{I}}[\![t_1[k]]\!](u')$ By the semantics of terms

(c) otherwise (i.e. $\tau_1 \not\equiv \varrho$ or $i \not\equiv j$), we have:

$\widehat{\mathbf{I}}[\![t_0[t/x]]\!](u) = \widehat{\mathbf{I}}[\![t'_1[k]]\!](u)$ By Definition 15

$= \widehat{\mathbf{I}}[\![t_1[k]]\!](u')$ From $\tau_1 \not\equiv \varrho$ or $i \not\equiv j$, by the semantics of terms and the induction hypothesis

$*$ otherwise (i.e. k is not a constant term), we have:

(a) if $typ(\tau_1) \supseteq \{\varrho\}$, then:

$\widehat{\mathbf{I}}[\![t_0[t/x]]\!](u) = \widehat{\mathbf{I}}[\![t_0[\texttt{self}, \underline{j}, t]]\!](u)$ By Definition 15

$= \widehat{\mathbf{I}}[\![t_0]\!](u')$ By the semantics of terms

(b) otherwise, we have:

$\widehat{\mathbf{I}}[\![t_0[t/x]]\!](u) = \widehat{\mathbf{I}}[\![t'_1[k']]\!](u)$ By Definition 15

$= \widehat{\mathbf{I}}[\![t_1[k]]\!](u')$ From $typ(\tau_1) \not\supseteq \{\varrho\}$, by the semantics of terms and the induction hypothesis

- (2), analogously.

8. $t_0 \equiv <t' : t_1, \ldots, t_n>$; (1) and (2) are obtained straightforwardly, from the semantics of terms and the induction hypothesis.

9. $t_0 \equiv f(t_1, \ldots, t_n)$, then:

- (1), by the following (where $t'_i \equiv t_i[x \rightsquigarrow t]$, for all i between 1 and n):

$$
\begin{aligned}
\widehat{\mathbf{I}}[\![t_0[t/x]]\!](u) &= \widehat{\mathbf{I}}[\![f(t'_1,\ldots,t'_n)]\!](u) && \text{By Definition 15}\\
&= \mathcal{I}(f)\big(\widehat{\mathbf{I}}_{\mathbf{x}}[\![t'_1]\!](u),\ldots,\widehat{\mathbf{I}}_{\mathbf{x}}[\![t'_n]\!](u)\big) && \text{By the semantics of terms}\\
&= \mathcal{I}(f)\big(\widehat{\mathbf{I}}_{\mathbf{x}}[\![t_1]\!](u'),\ldots,\widehat{\mathbf{I}}_{\mathbf{x}}[\![t_n]\!](u')\big) && \text{By the induction hypothesis}\\
&= \widehat{\mathbf{I}}[\![f(t_1,\ldots,t_n]\!](u') && \text{By the semantics of terms}
\end{aligned}
$$

- (2), analogously, from $\widehat{\mathbf{I}}_{\mathbf{x}}[\![f(t_1,\ldots,t_n)]\!](u) = \widehat{\mathbf{I}}[\![f(t_1,\ldots,t_n)]\!](u)$, for all function symbols f, terms $t_1,\ldots,t_n$ and $u \in U$.

10. $t_0 \equiv t_1[t_2, \underline{i}, t']$; then (1) and (2) are direct, from the semantics of terms.

$\square$

Appendix B

Proof of Continuity of the Functional $\Phi[\![pg]\!]$

This appendix shows, using propositions given in Section 1.2.6, that the functional $\Phi[\![pg]\!]$ defined in Section 2.2.2 is continuous.

Let then pg be a program, $\{odn_1, \ldots, odn_n\}$ be the set of object definition names of pg, $n \geq 1$. For all $i = 1, \ldots, n$, let $\{m_{i1}, \ldots, m_{ik_i}\}$ be the set of method names of odn_i, $k_i \geq 1$, and for all $j = 1, \ldots, k_i$ let c_{ij} be the body of m_{ij} and $fp_{ij1}, \ldots, fp_{ijk_{ij}}$ the formal parameters of m_{ij}.

The functional $\Phi[\![pg]\!] : (G_\varepsilon)^k \to (G_\varepsilon)^k$, where $k = k_1 + \cdots + k_n$ is given by:

$$\begin{aligned}\Phi[\![pg]\!](g) = \ & \Big(\lambda\alpha.\lambda\bar{\theta}_{11}.\lambda\sigma.\mathbf{I}[\![c_{11}]\!](\alpha)(\gamma)(\sigma_{11}), \ldots, \\ & \lambda\alpha.\lambda\bar{\theta}_{1k_1}.\lambda\sigma.\mathbf{I}[\![c_{1k_1}]\!](\alpha)(\gamma)(\sigma_{1k_1}), \ldots, \\ & \lambda\alpha.\lambda\bar{\theta}_{n1}.\lambda\sigma.\mathbf{I}[\![c_{n1}]\!](\alpha)(\gamma)(\sigma_{n1}), \ldots, \\ & \lambda\alpha.\lambda\bar{\theta}_{nk_n}.\lambda\sigma.\mathbf{I}[\![c_{nk_n}]\!](\alpha)(\gamma)(\sigma_{nk_n})\Big)\end{aligned}$$

where we have, for all i, j:

1. γ is the function assignment given by:

   $$\gamma = \{(odn_i, m_{ij}) \mapsto g_{ij}\}$$

   for $g = (g_{11}, \ldots, g_{1k_1}, \ldots, g_{n1}, \ldots, g_{nk_n})$

2. $\bar{\theta}_{ij} = (\theta_{ij1}, \ldots, \theta_{ijk_{ij}})$

3. $\sigma_{ij} = \sigma \dagger \big\{\alpha \mapsto \sigma(\alpha) \dagger \{fp_{ij1} \mapsto \bar{\theta}_{ij1}, \ldots, fp_{ijk_{ij}} \mapsto \bar{\theta}_{ijk_{ij}}\}\big\}$

By Proposition 1, G_ε is a cpo. By Proposition 2, $(G_\varepsilon)^k$ is a cpo ($k \geq 1$).

By Proposition 8, $\Phi[\![pg]\!]$ is continuous if $pr_q \circ \Phi[\![pg]\!]$ is continuous for all $q = 1, \ldots, k$.

For an arbitrary $q \in \{1, \ldots, k\}$, we have, since $k = k_1 + \cdots + k_n$, that $q = k_1 + \cdots + k_{i-1} + j$, for some i, j such that $1 \leq i \leq n$ and $1 \leq j \leq k_i$.

In this chapter we let (for brevity) $Y_{ij} = A \times \Theta^{k_{ij}} \times \Sigma$ and $X_{ij} = (G_\varepsilon)^k \times Y_{ij}$.

By Proposition 14 and the definition of $\Phi[\![pg]\!]$, $pr_q \circ \Phi[\![pg]\!]$ is continuous if the function

$$h_q^{c_{ij}}[\![pg]\!] : X_{ij} \to (\Sigma \times \Theta)_\varepsilon$$

given by

$$h_q^{c_{ij}}[\![pg]\!](g, \alpha, \bar{\theta}_{ij}, \sigma) = \mathbf{I}[\![c_{ij}]\!](\alpha)(\gamma)(\sigma_{ij})$$

is continuous.

We show that $h_q^{c_{ij}}[\![pg]\!]$ is continuous by induction on the structure of c_{ij}. We have:

1. $c_{ij} \equiv e$; then $h_q^{c_{ij}}[\![pg]\!]$ is continuous, by Proposition 9, since it is continuous in its second, third and fourth arguments, by Proposition 5, and it is continuous in its first argument, by Proposition 7 (since it is constant).

2. $c_{ij} \equiv x \leftarrow c$; then $h_q^{c_{ij}}[\![pg]\!]$ is continuous, by Proposition 9, since it is continuous in its second, third and fourth arguments, by Proposition 5, and it is continuous in its first argument, by Proposition 13, since $h_q^{c_{ij}}[\![pg]\!][\![\alpha, \bar{\theta}_{ij}, \sigma]\!]$ can be written as:

$$h_q^{c_{ij}}[\![pg]\!][\![\alpha, \bar{\theta}_{ij}, \sigma]\!](g)\ \ \texttt{if-then-else}\big(f_1(g), f_2(g), f_3(g)\big)$$

where:

(a) $f_1 : (G_\varepsilon)^k \to \mathbf{B}_\perp$ is given by:

$$f_1(g) = f_=\big(h_q^{c_1}[\![pg]\!](g, \alpha, \bar{\theta}_{ij}, \sigma), \perp_{\Sigma\times\Theta}\big)$$

where $f_=$ is the strict extension of the equality function on $\Sigma \times \Theta$.
f_1 is continuous, by Proposition 13, since $f_=$ is continuous, by Proposition 6, $h_q^{c_1}$ is continuous, by the induction hypothesis, and constant functions are continuous.

(b) $f_2 : (G_\varepsilon)^k \to (\Sigma \times \Theta)_\varepsilon$ is given by:

$$f_2(g) = \perp_{\Sigma\times\Theta}$$

which is constant, thus continuous.

(c) $f_3 : (G_\varepsilon)^k \to (\Sigma \times \Theta)_\varepsilon$ is given by:

$$f_3(g) = \texttt{if-then-else}\big(f_4(g), f_5(g), f_6(g)\big)$$

where:

- $f_4 : (G_\varepsilon)^k \to \mathbf{B}_\perp$ is given by:

$$f_4(g) = f_=\big(h_q^{c_1}[\![pg]\!](g, \alpha, \bar{\theta}_{ij}, \sigma), \mathit{error}\big)$$

which is continuous, analogously as f_1.

- $f_5 : (G_\varepsilon)^k \to (\Sigma \times \Theta)_\varepsilon$ is given by:

$$f_5(g) = \mathit{error}$$

which is constant, thus continuous.

- $f_6 : (G_\varepsilon)^k \to (\Sigma \times \Theta)_\varepsilon$ is given by:

$$f_6(g) = \big(\sigma' \dagger \big\{\alpha \mapsto \sigma'(\alpha) \dagger \{x \mapsto \theta\}\big\}, \theta\big)$$

which is constant, thus continuous.

3. $c_{ij} \equiv e_0.m_{rs}(e_1, \ldots, e_t)$, where $t \geq 0$ and r, s are such that $1 \leq r \leq n$ and $1 \leq s \leq k_r$; let $u = k_1 + \cdots + k_{r-1} + s$, and let, for brevity, $\mathbf{I}[\![e_i]\!](\sigma_{ij}(\alpha)) = d_i$, for all $i = 0, 1, \ldots, t$; the function $h_q^{c_{ij}}[\![pg]\!]$ can be written as:

$$\begin{aligned} h_q^{c_{ij}}[\![pg]\!](x) = \ & \texttt{if-then-else}\Big(d_0 = \perp_{\Sigma\times\Theta}, \perp_{\Sigma\times\Theta}, \\ & \texttt{if-then-else}\Big(d_0 = \mathit{error}, \mathit{error}, \\ & \texttt{if-then-else}\Big(d_0 = \texttt{nil}, \mathit{error}, \\ & \texttt{if-then-else}\Big(d_1 = \perp_{\Sigma\times\Theta}, \perp_{\Sigma\times\Theta}, \\ & \texttt{if-then-else}\Big(d_1 = \mathit{error}, \mathit{error}, \\ & \vdots \\ & \texttt{if-then-else}\Big(d_t = \perp_{\Sigma\times\Theta}, \perp_{\Sigma\times\Theta}, \\ & \texttt{if-then-else}\big(d_t = \mathit{error}, \mathit{error}, \\ & \mathit{apply}(f_u(x), g_u(x))\big)\Big)\ldots\Big)\Big)\Big)\Big)\Big) \end{aligned}$$

where $f_u : X_{rs} \to \big(Y_{rs} \to (\Sigma \times \Theta)_\varepsilon\big)$ is given by:

$$f_u(x) = apply(pr_u, g)$$

and $g_u : X_{rs} \to Y_{rs}$ is given by:

$$g_u(x) = \big(d_0, (d_1, \ldots, d_t), \sigma_{ij}\big)$$

f_u is continuous, by Proposition 9, since:

- it is continuous in its first argument, by Proposition 13, since *apply*, identity and projection functions are continuous, and
- it is continuous in the other arguments, by Proposition 5.

g_u is continuous, by Proposition 9 since it is continuous in its first argument, by Proposition 7 (since it is constant) and continuous in the other arguments, by Proposition 5.

Thus, since *apply* is continuous, and `if-then-else` and constant functions are continuous, it holds that $h_q^{c_{ij}}[\![pg]\!]$ is continuous, by (repeated applications of) Proposition 13.

4. $c_{ij} \equiv x \leftarrow odn!\mathtt{new}(e_1, \ldots, e_t)$, where $t \geq 0$; then, analogously as in item 1, $h_q^{c_{ij}}[\![pg]\!]$ is continuous.

5. $c_{ij} \equiv c_1; c_2$; then let, for brevity:

$$\sigma' = pr_1 \circ h_q^{c_1}[\![pg]\!](g, \alpha, \bar{\theta}_{ij}, \sigma)$$

$$\mathbf{I}[\![fp_{ij1}]\!](\sigma'(\alpha)) = \theta'_{ij1}, \ldots,$$
$$\mathbf{I}[\![fp_{ijk_{ij}}]\!](\sigma'(\alpha)) = \theta'_{ijk_{ij}}$$

and

$$\bar{\theta}'_{ij} \equiv (\theta'_{ij1}, \ldots, \theta'_{ijk_{ij}})$$

$h_q^{c_{ij}}[\![pg]\!]$ can be written as:

$$\begin{aligned} h_q^{c_{ij}}[\![pg]\!](g, \alpha, \bar{\theta}_{ij}, \sigma) = \; & \mathtt{if\text{-}then\text{-}else}\Big(f_=(\sigma', \bot_{\Sigma\times\Theta}), \bot_{\Sigma\times\Theta}, \\ & \mathtt{if\text{-}then\text{-}else}\big(f_=(\sigma', error), error, \\ & f(g, \alpha, \bar{\theta}_{ij}, \sigma)\big)\Big) \end{aligned}$$

where

$$f(g, \alpha, \bar{\theta}_{ij}, \sigma) = h_q^{c_2}[\![pg]\!](g, \alpha, \bar{\theta}'_{ij}, \sigma')$$

The projection function (pr_1) is continuous, by Proposition 7, and $h_q^{c_1}[\![pg]\!]$ is continuous, by the induction hypothesis. Then their composition is continuous, by Proposition 12.

f is continuous, by Proposition 13, since it can be written as $f(g, \alpha, \bar{\theta}_{ij}, \sigma) = apply\big(h_q^{c_2}[\![pg]\!], f_1(g)\big)$ where

$$f_1(g, \alpha, \bar{\theta}_{ij}, \sigma) = (g, \alpha, \bar{\theta}'_{ij}, \sigma')$$

and f_1 is continuous, by Proposition 8 (using Propositions 9 and 7), *apply* is continuous, and $h_q^{c_2}[\![pg]\!]$ is continuous (by the induction hypothesis).

Therefore, and since also constant functions, $f_=$ and `if-then-else` are continuous, $h_q^{c_{ij}}[\![pg]\!]$ is continuous, by Proposition 13.

6. $c_{ij} \equiv \mathtt{if}\; c_0 \;\mathtt{then}\; c_1 \;\mathtt{else}\; c_2$; similarly as above.

7. $c \equiv bk$; similarly as above.

Appendix C

The Denotational Semantics of Ajour

This chapter presents a denotational semantics for a language based on a concurrent object-based model of computation, with synchronous and asynchronous message passing, named **Ajour**. It develops a simple formal semantics definition method that is compositional in the presence of parallelism. The method used here has been inspired mainly by the work of Pierre America on a denotational semantics for POOL[AR92], but it's hoped to offer simpler definitions. The definition in this chapter needs further work related to the justification of well-definedness of the semantic functions and also further work associated with possible extensions concerning the inclusion of recursive method calls and related aspects of non-termination of programs.

An important concept of the semantics definition method used here is that of an *external event space* — or *interaction space* — of possibly functional values, which enables the definition of a function to merge parallel actions. The method is based on the (so-called) method of *resumption semantics* developed by Plotkin[Sch86, Section 12.3][Plo76].

C.1 Notation

The semantics definitions in this chapter use the following notational conventions, additionally to those given in Section 1.2.1:

- finite partial functions: $X \xrightarrow{m} Y$ denotes the set of all finite partial functions from X to Y;
- variant functions: given $f : X \to Y$, $x_1, \ldots, x_n \in X$ and $y_1, \ldots, y_n \in Y$, then $f \dagger \{x_i \mapsto y_i\}_{i=1}^{n}$ is used as an abbreviation for:

$$f \dagger \{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$$

C.2 Introduction

This section gives an introduction to the method by using it to define the semantics of two concurrent "baby" languages.

C.2.1 L2

L2 is a concurrent language simplified to the point of being completely deterministic.

The execution of programs in **L2** is as follows. The execution of a program p, in which its commands, say c_1 and c_2, are executed concurrently, terminates always and terminates with a result that is either *error* or a pair (n_1, n_2) of integer values given by the execution of c_1 and c_2, respectively. The value n_i

($i = 1$ or $i = 2$) is given by the first (sub-)command $\uparrow e$ of c_i executed. The set of variables referenced in these two commands is disjoint (i.e. there are no shared variables). There are simple commands for sending and receiving integer values between the concurrent processes associated with c_1 and c_2. The command for sending values is asynchronous.

Syntax

Consider the following sets and corresponding arbitrary elements: $pg \in Pg, c \in C, x \in Var, n \in Num, e \in E, op \in \{+, -, *\}$.

Let the syntax of L2 be defined by:

$$
\begin{array}{lcl}
pg & ::= & c_1 \mid\mid c_2 \\
c & ::= & x := e \mid \; c_1; c_2 \mid !e \mid ?x \mid \uparrow e \\
e & ::= & n \mid \; x \mid \; e_1 \, op \, e_2
\end{array}
$$

Semantics

We define the following sets:

$$\Sigma = Var \rightarrow (Int \cup \{\emptyset\})$$

$$CC = \Sigma \rightarrow R$$

R can be informally interpreted as an external event space, or an interaction space (for parallel processes). In this space are output the values and functions that are required to define below the interaction function ι, which specifies the result of the interaction of occurred events. The non-internal event space is here defined by the following recursive domain equation:

$$R = (Int \times R) \cup (Int \rightarrow R) \cup \{r_0\}$$

r_0 is a distinguished element intended to represent a completed computation.

CC is the so-called set of *continuations*.[1] Here continuations allow the separation of the local effects of a command from those needed for the interaction between its process and other processes executing in parallel.

The use of continuations together with a non-internal event space simplifies the definition of the semantic rules. The semantic rule of any command c specifies:

- the meaning of the computation following c (corresponding to its sequential process);
- the values and functions of the event space required for the interaction between its process and parallel ones.

The meaning functions μ_{pg}, μ_c and μ_e are defined inductively in Figure C.1; $\tilde{n}$ denotes the integer number corresponding to the numeral n, and operations $\widetilde{op}$ corresponding to the syntactic elements given by op are the usual arithmetic operations.

C.2.2 Ln

Ln is a natural extension of **L2**. It has n commands executing concurrently (instead of just two as in **L2**). Non-determinism is introduced (due to matching between receiving a value and values sent being possible for more than one pair of concurrent processes) and thus the meaning of a program is no longer defined as an integer tuple, but as a set of tuples representing the set of possible results of the execution of that program, or $error$.

[1]Cf. Section 1.2.5.

$$
\begin{array}{l}
\mu_{pg} : Pg \to \Sigma \to (Int^2 \cup \{error\}) \\
\mu_{pg}[\![c_1 \mid\mid c_2]\!](\sigma) = \iota(\mu_c[\![c_1]\!](finis)(\sigma), \mu_c[\![c_2]\!](finis)(\sigma)) \\
\text{where:} \\
\quad finis = \lambda\sigma.r_0 \\
\\
\quad \iota : R^2 \to (Int^2 \cup \{error\}) \text{ is recursively defined by:} \\
\quad \iota(r_1, r_2) = \begin{cases} \iota(r_1', (\lambda v.r_2')v_1) & \text{if } r_1 = (v_1, r_1') \text{ and } r_2 = \lambda v.r_2' \\ \iota((\lambda v.r_1')v_2, r_2') & \text{if } r_1 = \lambda v.r_1' \text{ and } r_2 = (v_2, r_2') \\ (v_1, v_2) & \text{if } r_1 = (v_1, r_0) \text{ and } r_2 = (v_2, r_0) \\ error & \text{otherwise} \end{cases} \\
\\
\mu_c : C \to CC \to \Sigma \to R \\
\mu_c[\![x := e]\!](cc)(\sigma) = cc(\sigma \dagger \{x \mapsto \mu_e[\![e]\!](\sigma)\}) \\
\mu_c[\![c_1; c_2]\!](cc) = \mu_c[\![c_1]\!](\mu_c[\![c_2]\!](cc)) \\
\mu_c[\![!e]\!](cc)(\sigma) = (\mu_e[\![e]\!](\sigma), cc(\sigma)) \\
\mu_c[\![?x]\!](cc)(\sigma) = \lambda v.cc(\sigma \dagger \{x \mapsto v\}) \\
\mu_c[\![\uparrow e]\!](cc)(\sigma) = (\mu_e[\![e]\!](\sigma), r_0) \\
\\
\mu_e : E \to \Sigma \to Int \\
\mu_e[\![n]\!](\sigma) = \tilde{n} \\
\mu_e[\![x]\!](\sigma) = \sigma(x) \\
\mu_e[\![e_1 \; op \; e_2]\!](\sigma) = (\mu_e[\![e_1]\!](\sigma)) \; \widetilde{op} \; (\mu_e[\![e_2]\!](\sigma))
\end{array}
$$

Figure C.1: Semantics of $L2$

The syntax is changed by just one rule:

$$pg \quad ::= \quad c_1 \mid\mid \ \ldots \ \mid\mid c_n$$

Given a cartesian product $X = X_1 \times \ldots \times X_n$, let $x = (x_1, \ldots, x_n)$ and consider the family of functions $\psi_{X,i} : X \times X_i \to X$ defined as:

$$\psi_{X,i}(x, x_i') = (x_1, \ldots, x_{i-1}, x_i', x_{i+1}, \ldots, x_n)$$

The notation $\xi(x, x_i/x_i')$ is used instead of $\psi_{X,i}(x, x_i')$, with no risk of confusion provided that each x_i is denoted by a distinct name. We extend this notation to allow more than one element of a cartesian product to be changed:

$$\xi(x, x_i/x_i', \ldots, x_j/x_j') = \xi(\xi(\ldots\xi(x, x_i/x_i'), \ldots), x_j/x_j')$$

No modification is needed to the defined sets and domain equation. The function μ_{pg} is changed as shown in Figure C.2.

C.3 Ajour

This section defines the formal semantics of a concurrent object-based language, named **Ajour**, for which:

- all objects are potential units of concurrent execution;
- there is only one thread of execution for non-primitive objects;

$$
\begin{array}{l}
\mu_{pg} : Pg \rightarrow \Sigma \rightarrow (\wp(Int^n) \cup error) \\
\mu_{pg}[\![c_1 \mid\mid \ \ldots \ \mid\mid \ c_n]\!](\sigma) = \iota(\mu_c[\![c_1]\!](finis)(\sigma), \ldots, \mu_c[\![c_n]\!](finis)(\sigma)) \\
\text{where:} \\
\ finis = \lambda\sigma.r_0
\end{array}
$$

$\iota : R^n \rightarrow (\wp(Int^n) \cup \{error\})$ is recursively defined by:

$$
\iota(r_1, \ldots, r_n) = \begin{cases}
\bigcup\{\iota(\xi(r, r_i/r'_i, r_j/r'_j))\} & \text{if } \exists i, j \in \{1, \ldots, n\} \text{ such that} \\
 & r_i = (v_i, r'_i) \text{ and } r_j = \lambda v.r''_j, \\
 & \text{ and where } r'_j = (\lambda v.r''_j)v_i \\
(v_1, \ldots, v_n) & \text{if } r_1 = (v_1, r_0), \ldots, r_n = (v_n, r_0) \\
error & \text{otherwise}
\end{cases}
$$

Figure C.2: Semantics of *Ln*

- a return command $\uparrow c$ can be used to specify the result of a method call before the end of that call, allowing (after the execution of $\uparrow c$) the executions associated with the sender and the receiver objects to proceed concurrently;
- message sending commands can be synchronous or asynchronous;[2]
- objects are created by (asynchronous) object creation procedure calls;
- that are commands that specify that execution continues after the execution of one of a given set of methods (answer commands) and commands that specify the set of possible methods that can be executed after that of the current method (become commands).

C.3.1 Syntax

The syntax given is an *abstract syntax*, but terminal symbols are used to increase readability.

For simplicity, the following notation used in the abstract (and thus semantically biased) syntax, instead of sequences, when the order of appearance of the elements is (semantically) irrelevant. When the order of the elements $x_1, \ldots, x_n$ is irrelevant, the notation $(x_i)_{i=1}^n$ is used instead of the list $x_1, \ldots, x_n$. Similarly, instead of using, for example,

$$pg ::= odn_1 : od_1; \cdots; odn_n : od_n$$

the notation

$$pg ::= (odn_i : od_i)_{i=1}^n$$

is used.

We assume the existence of:

- A set Π of primitive[3] object names and a set $\mathcal{E}$ of *external* object names, disjoint from Π. The set $\mathcal{U}$ is defined as $\Pi \cup \mathcal{E}$.
- A set $\mathcal{F}_1$ of predefined functions, which forms an algebra whose carriers are subsets of Π_1 (e.g. `Int`, `Bool` etc.).
- A set $\mathcal{F}_2$ of external functions, which forms an algebra whose carriers are subsets of $\mathcal{U}$.

[2] Synchronous and asynchronous message sending commands are indicated respectively by the use of one and two dots (cf. syntax and semantics below).

[3] Also called predefined, or standard.

$$
\begin{array}{lcl}
pg & ::= & (odn_i : od_i)_{i=1}^n \\
od & ::= & (v_i)_{i=1}^k ; (prn_i : pr_i)_{i=1}^n \\
pr & ::= & (t_1, \ldots\ t_n) ; c \\
c & ::= & x \ \mid\ u \ \mid\ \mathbf{self} \ \mid\ \mathbf{nil} \\
 & \mid & x \leftarrow\ c \\
 & \mid & \uparrow c \\
 & \mid & c.m(c_1, \ldots, c_n) \\
 & \mid & c..m(c_1, \ldots, c_n) \\
 & \mid & odn!q(c_1, \ldots, c_n) \\
 & \mid & \mathbf{answer}\ \beta \\
 & \mid & \mathbf{become}\ \beta \\
 & \mid & b \\
 & \mid & c_1 ; c_2 \\
 & \mid & c_1 = c_2 \\
b & ::= & (t_i)_{i=1}^n ; c
\end{array}
$$

Figure C.3: Abstract Syntax of Ajour

We use the following syntactic sets and typical elements:

$pg \in\ Pg$	program
$odn \in Odn$	object definition name
$od \in\ Od$	object definition
$prn \in Prn$	procedure name ($Prn = Q \cup M$)
$pr \in Pr$	procedure
$m \in\ M$	method (external procedure) name
$q \in\ Q$	creation procedure name
$v \in\ V$	object variable (name)
$t \in\ T$	temporary variable (name)
$x \in\ X$	variable (name) ($X = V \cup\ T$)
$c \in\ C$	command
$\beta \in \wp(M)$	finite set of method names
$b \in\ B$	block

The abstract syntax rules are given in Figure C.3.

C.3.2 Semantics

Consider the following disjoint sets:

A The set of names of *active* objects

$\mathcal{Z}$ The set of names of *future* objects

Active objects are those dynamically created during a program execution, as a result of an object creation command. *Future* objects are created as a result of asynchronous message sending commands.

In the semantic definition we denote by:

$$\widetilde{\Phi} \text{ the set } A \cup \mathcal{U}$$

$$\Phi \text{ the set } \widetilde{\Phi} \cup \{\emptyset\}$$

$$\Theta \text{ the set } \Phi \cup \mathcal{Z}$$

$$W \text{ the set } A \cup \mathcal{Z} \cup \{\emptyset\}$$

The syntactic element **nil** is a command that keeps the local state unchanged and returns $\emptyset$, indicating the absence of an object.

Informally, a given $w \in W$ denotes an object associated with the object given as the result of a given message (this association is formally defined below by the interaction function ι). The empty set element of W is used when more than one return command is executed for a given method activation and to model events corresponding to synchronous messages with corresponding and already occurred method activation events.

Let K_1 and K_2 ($K_1 = K_2 = N$) denote the set of creation counters for active and future objects, respectively. The meaning functions are defined so that:

- any active object name a is a triple (odn, ϕ, k), where odn is the object definition name associated with a (i.e. from which a is created), ϕ denotes the object that created a and $k \in K_1$;
- any future object name z is a pair (a, k), where a denotes the active object that created the object denoted by z and $k \in K_2$.

For each active object a, let X_a be a given finite set of variables and consider the following sets:

$$\Sigma_a = X_a \stackrel{m}{\rightarrow} \Theta$$

$$\Delta_a = \Sigma_a \times A \times W \times \wp(M_a) \times K_1 \times K_2$$

The set Σ_a can be referred to as a's *local store* and Δ_a as a's *local state*. The local state of a given object is an n-tuple containing, respectively:

- the local store (of a);
- the object denoted (or returned) by **self**;
- the object associated with the result of a method call[4];
- the set of methods (of a) that may be activated by the next message, and
- the creation counters for active and future objects (associated with a), respectively.

Let $Msg = M \times \Theta^* \times W$, and consider the following sets and domain R:

$$\Gamma = A \stackrel{m}{\rightarrow} D$$

$$D = X \cup ((P \cup M) \stackrel{m}{\rightarrow} Proc) \cup (Q \stackrel{m}{\rightarrow} Cproc)$$

$$Proc = \Theta^* \rightarrow CC \rightarrow \Delta \rightarrow R$$

$$Cproc = \Theta^* \rightarrow \mathcal{Z} \rightarrow A \rightarrow R$$

$$CC = \Theta \rightarrow \Delta \rightarrow R$$

Γ is referred to as the (static) *environment* of the program, and D as the set of *denotable values*. We adopt a simple hierarchical version of the environment, in which procedure names are particular to a certain object definition name. Also, given this hierarchical characterization of the environment, for simplicity we do not to distinguish between identifiers (variable names) and the variables themselves. Similarly, no distinction is made between primitive and external object names and the corresponding objects themselves.

Here R is specified by the following equation:

[4] I.e. the "waiting" object.

$$
\begin{aligned}
R = \quad & \{r_0\} \cup (W \times \Theta \times R) \cup \\
& (\Phi \times Msg \times R) \cup \\
& (\Phi \times Msg \times (\Theta \to R)) \cup \\
& (\mathcal{Z} \times (\Phi \to R)) \cup \\
& (A \times \wp(M) \times (Msg \to R)) \cup \\
& (R \times R)
\end{aligned}
$$

Informally, elements of the above set model the non-internal events associated with objects, as follows:

- r_0: a completed computation;
- (w, θ, r): a result setting, with result value θ associated with (either future or active) object w;
- $(\tilde{\phi}, msg, r)$: asynchronous message sending;
- (ϕ, msg, f): synchronous message sending, where the message is given by msg and the target object by ϕ; a function f from object names to R models a space dependent on the value returned as a result of the message;
- (z, f): a function f from non-future objects to R models a space dependent on the value returned as a result of an asynchronous message, associated (in a way precisely specified by the interaction function) with future object z;
- (a, β, f): a function f from messages to R models a space dependent on a message sent to active object a (for messages restricted to β as a result of an answer or become command);
- (r_1, r_2): object creation: the event space of any object includes those of the objects that it creates.

The meaning functions are defined inductively below. The semantics of commands is given first, in Figure C.4. For simplicity, error conditions are not specified.

The semantics of syntactic constructs other than programs are given in Figure C.5.

To define the meaning function of programs, let:

- $Emsg$ be the set of external messages given by:

$$Emsg = \mathcal{E} \times M \times \Theta^*$$

- T be the set of sequences of sequences of external messages; each sequence of external messages is associated with a distinct active object (and corresponds to the external messages it sends):

$$T = (Emsg^*)^*$$

The occurrences of events in the interaction space are given by boolean expressions $E_k, 1 \leq k \leq n_{ev}$ (where $n_{ev} = 6$), corresponding respectively to:

1. the result of a synchronous message;
2. the result of an asynchronous message;
3. a method activation;
4. a synchronous message to a primitive object;
5. an asynchronous message to a primitive object;
6. a synchronous message to an external object.

The functions defined in Figure C.6 are used to specify the boolean expressions $E_k, 1 \leq k \leq n_{ev}$.

The meaning function μ_{pg} is defined in Figure C.7.

The logical expressions $E_k, 1 \leq k \leq n_{ev}$, are given in Figure C.8. For each of the sets T_k, we have: $\neg E_k \Rightarrow T_k = \emptyset$, otherwise T_k is as defined in this figure.

$\mu_c : C \to \Gamma \to CC \to \Delta \to R$
Let in the following equations $\bar{\theta} = (\theta_1, \ldots, \theta_n)$, $\delta_a = (\sigma, a, w, \beta, k_1, k_2)$

$$\mu_c[\![x]\!](\gamma)(cc)(\delta_a) = cc(\sigma(x))(\delta_a)$$
$$\mu_c[\![u]\!](\gamma)(cc)(\delta_a) = cc(u)(\delta_a)$$
$$\mu_c[\![\mathbf{self}]\!](\gamma)(cc)(\delta_a) = cc(a)(\delta_a)$$
$$\mu_c[\![\mathbf{nil}]\!](\gamma)(cc)(\delta_a) = cc(\emptyset)(\delta_a)$$
$$\mu_c[\![x := c]\!](\gamma)(cc) = \mu_c[\![c]\!](\gamma)\big(\lambda\theta.\lambda\delta_a.cc(\theta)(\xi(\delta_a, \sigma_a \dagger \{x \mapsto \theta\}))\big)$$
$$\mu_c[\![c_1; c_2]\!](\gamma)(cc) = \mu_c[\![c_1]\!](\gamma)(\mu_c[\![c_2]\!](\gamma)(cc))$$
$$\mu_c[\![b]\!] = \mu_b[\![b]\!]$$
$$\mu_c[\![\uparrow c]\!](\gamma)(cc) = \mu_c[\![c]\!](\gamma)(\lambda\theta.\lambda\delta_a.r)$$
$$\text{where } r = \begin{cases} cc(\theta)(\delta_a) & \text{if } w = \emptyset \\ \big(w, \theta, cc(\theta)(\xi(\delta_a, w/\emptyset))\big) & \text{otherwise} \end{cases}$$

$$\mu_c[\![c.m(c_1, \ldots, c_n)]\!](\gamma)(cc) =$$
$$\mu_c[\![c]\!](\gamma)\Big(\lambda\theta_0.\lambda\delta_0.\mu_c[\![c_1]\!](\gamma)\big(\lambda\theta_1.\lambda\delta_1.\cdots\mu_c[\![c_n]\!](\gamma)(\lambda\theta_n.\lambda\delta_n.r)\ldots\big)\Big)$$
$$\text{where } r = \begin{cases} \big(\theta_0, m, \bar{\theta}, a, \lambda\theta.cc(\theta)(\delta_n)\big) & \text{if } \theta_0 \in \widetilde{\Phi} \\ \Big(\theta_0, \lambda\phi.\big(\phi, m, \bar{\theta}, a, \lambda\theta.cc(\theta)(\delta_n)\big)\Big) & \text{if } \theta_0 \in \mathcal{Z} \end{cases}$$

$$\mu_c[\![c..m(c_1, \ldots, c_n)]\!](\gamma)(cc) =$$
$$\mu_c[\![c]\!](\gamma)\Big(\lambda\theta_0.\lambda\delta_0.\mu_c[\![c_1]\!](\gamma)\big(\lambda\theta_1.\lambda\delta_1.\cdots\mu_c[\![c_n]\!](\gamma)(\lambda\theta_n.\lambda\delta_n.r)\ldots\big)\Big)$$
$$\text{where } r = \begin{cases} \big(\theta_0, m, \bar{\theta}, z, cc(z)(\xi(\delta_n, k/k+1))\big) & \text{if } \theta_0 \in \widetilde{\Phi} \\ \Big(\theta_0, \lambda\phi.\big(\phi, m, \bar{\theta}, z, cc(z)(\xi(\delta_n, k/k+1))\big)\Big) & \text{if } \theta_0 \in \mathcal{Z} \end{cases}$$
and $z = (k, a), k = \delta_n \downarrow 5$

$$\mu_c[\![odn!q(c_1, \ldots, c_n)]\!](\gamma)(cc) =$$
$$\mu_c[\![c_1]\!](\gamma)\Big(\lambda\theta_1.\lambda\delta_1.\mu_c[\![c_2]\!](\gamma)\big(\lambda\theta_2.\lambda\delta_2.\cdots\mu_c[\![c_n]\!](\gamma)(\lambda\theta_n.\lambda\delta_n.r)\ldots\big)\Big)$$
where $r = \big(\gamma(odn)(q)(\bar{\theta})(z)(a), cc(z)(\xi(\delta_n, k/k+1))\big)$,
and $a_n = \delta_n \downarrow 2, z = (k, a_n), k = \delta_n \downarrow 6, a = (odn, a_n, \delta_n \downarrow 5)$

$$\mu_c[\![c_1 = c_2]\!](\gamma)(cc) = \mu_c[\![c_1]\!](\gamma)\big(\lambda\theta_1.\lambda\delta_1.\mu_c[\![c_2]\!](\gamma)(\lambda\theta_2.\lambda\delta_2.r)\big)$$
$$\text{where } r = \begin{cases} cc(\theta_1 = \theta_2)(\delta_2) & \text{if } \theta_1, \theta_2 \in \Phi \\ \big(\theta_1, \lambda\phi.cc(\phi = \theta_2)(\delta_2)\big) & \text{if } \theta_1 \in \mathcal{Z} \text{ and } \theta_2 \in \Phi \\ \big(\theta_2, \lambda\phi.cc(\phi = \theta_1)(\delta_2)\big) & \text{if } \theta_2 \in \mathcal{Z} \text{ and } \theta_1 \in \Phi \\ \Big(\theta_1, \lambda\phi.\big(\theta_2, \lambda\phi'.cc(\phi = \phi')(\delta_2)\big)\Big) & \text{if } \theta_1, \theta_2 \in \mathcal{Z} \end{cases}$$

$$\mu_c[\![\mathbf{answer}\ \beta_1]\!](\gamma)(cc)(\omega_a) = \big(a, \beta_1, \lambda m.\lambda\bar{\theta}.\lambda w_1.\gamma(a \downarrow 1)(m)(\bar{\theta})(cc_1)(\xi(\delta_a, w/w_1))\big)$$
where $cc_1 = \lambda\theta'.\lambda\delta.cc(a)(\xi(\delta, \delta \downarrow 2/a, \delta \downarrow 3/w))$
$$\mu_c[\![\mathbf{become}\ \beta_1]\!](\gamma)(cc)(\omega_a) = cc(a)(\xi(\delta_a, \beta/\beta_1))$$

Figure C.4: Semantics of Commands

$\mu_b : B \to \Gamma \to CC \to \Omega \to R$
$\mu_b[\![(t_1, \ldots, t_n); c]\!](\gamma)(cc)(\omega_a) = \mu_c[\![c]\!](\gamma)(cc)(\xi(\delta_a, \sigma_a \dagger \{t_i \mapsto \emptyset\}_{i=1}^n)),$

$\mu_{pr} : Pr \to \Gamma \to Proc$
$\mu_{pr}[\![(t_1, \ldots, t_n); c]\!](\gamma) = \lambda\bar{\theta}.\lambda cc.\lambda\delta_a.\mu_c[\![c]\!](\gamma)(cc)((\emptyset, \delta_1))$
where $\delta_1 = \xi(\delta_a, \sigma_a \dagger \{t_i \mapsto \bar{\theta} \downarrow i\}_{i=1}^n),$

$\mu_{cp} : Cpr \to \Gamma \to Cproc$
$\mu_{cp}[\![(t_1, \ldots, t_n); c]\!](\gamma) = \lambda\bar{\theta}.\lambda z.\lambda a'.\mu_c[\![c]\!](\gamma)(cc)((\emptyset, \delta))$
where:
$\delta = (\lambda x.\emptyset \dagger \{t_i \mapsto \bar{\theta} \downarrow i\}_{i=1}^n, a', z, M_{a' \downarrow 1}), x \in X_{a'}$
$cc = \lambda\theta.\lambda\delta_a.\big(a, \beta, \lambda m.\lambda\bar{\theta}'.\lambda w'.\gamma(a \downarrow 1)(m)(\bar{\theta}')(cc)(\xi(\delta_a, a/a', w/w'))\big)$

$\mu_{od} : Od \to D$
Let:
$od_1 \equiv (v_i)_{i=1}^k; (prn_i : pr_i)_{i=1}^n$ and $f = \{prn_i \to pr_i \mid 1 \leq i \leq n\},$
$M_1 \cup Q_1 = \{prn_i \mid 1 \leq i \leq n\}$ and $V_1 = \{v_i \mid 1 \leq i \leq k\}$
$\mu_{od}[\![od_1]\!] = V_1 \cup \{pn \mapsto \mu_{pr}[\![pr_{f(pn)}]\!] \mid pn \in M_1\} \cup \{q \mapsto \mu_{cp}[\![pr_{f(q)}]\!] \mid q \in Q_1\}$

Figure C.5: Semantics of Object Definitions, Procedures and Blocks

$tail : R \to R$ is defined by:

$$tail(r) = \begin{cases} tail(r \downarrow 3) & \text{if } r \in (W \times \Theta \times R) \cup (\Phi \times Msg \times R) \\ r & \text{otherwise} \end{cases}$$

$hd_1 : R \to W \to R$ is defined by:

$$hd_1(r, w) = \begin{cases} hd_1(r \downarrow 3, w) & \text{if } r \in (W - \{w\} \times \Theta \times R) \cup (\Phi \times Msg \times R) \\ r & \text{otherwise} \end{cases}$$

$hd_2 : R \to A \to R$is defined by:

$$hd_2(r, w) = \begin{cases} hd_2(r \downarrow 3, a) & \text{if } r \in (W \times \Theta \times R) \cup (\Phi - \{a\} \times Msg \times R) \\ r & \text{otherwise} \end{cases}$$

Let $R' = \big(\cup(\Phi \times Msg \times R)\big);$
$rdc : R \times R \times R \to R$ is defined by:

$$rdc(r_1, r_2, r_3) = \begin{cases} (r_1 \downarrow 1, r_1 \downarrow 2, rdc(r_1 \downarrow 3, r_2, r_3)) & \text{if } r_1 \in (W \times \Theta \times R) \text{ or} \\ & \quad r_1 \in (\Phi \times Msg \times R), \\ & \text{and } r_1 \downarrow 1 \neq r_2 \downarrow 1, \text{ or} \\ & \quad r_1 \downarrow 2 \neq r_2 \downarrow 2 \\ r_3 & \text{otherwise} \end{cases}$$

Figure C.6: Operations on subdomains

$\mu_{pg} : Pg \to \wp(B) \to \wp(T)$
Let:
$pg_1 \equiv (odn_i : od_i)_{i=1}^{k}$
$\gamma = \{odn_i \mapsto \mu_{od_i}[\![od]\!]\}_{i=1}^{k}$,
$\hat{b} = \{b_i \mid 1 \leq i \leq n\}$
$\mu_{pg}[\![pg_1]\!](\hat{b}) = \iota\big(\mu_b[\![b_1]\!](\gamma)(finis)(\omega_1), \ldots, \mu_b[\![b_n]\!](\gamma)(finis)(\omega_n)\big)$
where:
$finis = \lambda\omega.r_0$,
$\omega_i = \big(\emptyset, (\lambda x.\emptyset, e_i, e_i, \emptyset)\big)$,
$e_i \in \mathcal{E}, \forall i \in \{1, \ldots, n\}$, and

$\iota : R^n \to \wp(T)$ is defined by:

$$\iota(r_1, \ldots, r_n) = \begin{cases} \bigcup_{k=1}^{n_{ev}} T_k & \text{if } \exists k; E_k \\ \epsilon\Big(r_n, \emptyset, \epsilon\big(r_{n-1}, \emptyset, \ldots, \epsilon(r_1, \emptyset, \emptyset) \ldots\big)\Big) & \text{otherwise} \end{cases}$$

where $\epsilon : R \times Emsg^* \times T \to T$ is defined by:

$$\epsilon(r, \overline{emsg}, t) = \begin{cases} \epsilon(r', concat(\overline{emsg}, (e, m, \bar{\theta})), t) & \text{if } r = (e, (m, \bar{\theta}, w), r'), \\ & \quad r \in (\mathcal{E} \times Msg \times R) \\ \epsilon(r \downarrow 1, \emptyset, \epsilon(r \downarrow 2, \overline{emsg}, t)) & \text{if } r \in R \times R \\ \epsilon(r \downarrow 3, \overline{emsg}, t) & \text{if } r \in (W \times \Theta \times R) \\ t & \text{otherwise} \end{cases}$$

Figure C.7: Semantics of programs

Consider $\bar{r} = (r_1, \ldots, r_n)$, $i, j \in \{1, \ldots, n\}$, and let:

$\boxed{\text{Result/Sync Msg}}$

$E_1 = \exists r_i, r_j;\quad tail(r_i) \in A \times Msg \times (\Theta \to R)$,
where we let r denote $tail(r_j)$ and let $r = (a, (m, \bar{\theta}, \emptyset), f)$, and
$hd_1(r_j, a) \in A \times \Theta \times R$ and $hd_1(r_j, a) \downarrow 1 = a$,
where we let r' denote $hd_1(r_j, a)$ and let r'' denote $r' \downarrow 3$

$$T_1 = \iota\Big(\xi\big(\bar{r}, r_i/rdc(r_i, r, f(\theta')), r_j/rdc(r_j, r', r'')\big)\Big)$$

$\boxed{\text{Result/Async Msg}}$

$E_2 = \exists r_i, r_j;\quad tail(r_i) \in \mathcal{Z} \times (\Phi \to R)$,
where we let r denote $tail(r_i)$ and let $r = (z, f)$, and
$hd_2(r_j, z) \in \mathcal{Z} \times \Theta \times R$ and $hd_2(r_j, z) \downarrow 1 = z$,
where we let r' denote $hd_2(r_j, z)$ and let $r' = (z, \theta', r'')$

$$\text{Let } r''' = \begin{cases} f(\theta') & \text{if } \theta' \in \Phi \\ (\theta', f) & \text{otherwise} \end{cases}$$

$$T_2 = \iota\Big(\xi\big(\bar{r}, r_i/rdc(r_i, r, r''')\big)\Big)$$

$\boxed{\text{Method Activation}}$

$E_3 = \exists r_i, r_j;\quad tail(r_i) \in A \times \wp(M) \times (Msg \to R)$,
where we let r denote $tail(r_i)$ and let $r = (a, \beta, f)$, and either
$hd_2(r_j, a) \in A \times Msg \times R$ and $hd_2(r_j, a) \downarrow 1 = a$, or otherwise
$tail(r_j) \in A \times Msg \times (\Phi \to R)$ and $tail(r_j) \downarrow 1 = a$
In the first case we let r' denote $hd_2(r_j, a)$,
and let $r' = (a, (m, \bar{\theta}, w), r'')$
In the second case we let r' denote $tail(r_j)$,
and let $r' = (a, (m, \bar{\theta}, w), f_1)$
In both cases we must have also $w \neq \emptyset$ and $m \in \beta$

$$\text{Let } r''' = \begin{cases} r'' & \text{if } r' = hd_2(r_j, a) \\ (a, (m, \bar{\theta}, \emptyset), f_1) & \text{otherwise} \end{cases}$$

$$T_3 = \iota\Big(\xi\big(\bar{r}, r_i/rdc(r_i, r, f(m, \bar{\theta}, w)), r_j/rdc(r_j, r', r''')\big)\Big)$$

Figure C.8: Events and Interactions

$\boxed{\text{Sync Predef Msg}}$
$E_4 = \exists r_i; tail(r_i) \in \Pi \times Msg \times (\Theta \rightarrow R)$
Let r denote $tail(r_i)$, let $r = (\pi, (m, \bar{\theta}, w), f)$ and $r' = f(\tilde{m}(\bar{\theta}))$, where $\tilde{m} \in \mathcal{F}_1$
$T_4 = \iota\Big(\xi\big(\bar{r}, r_i/rdc(r_i, r, r')\big)\Big)$

$\boxed{\text{ASync Predef Msg}}$
$E_5 = \exists r_i; hd_2(r_i) \in \Pi \times Msg \times R$
Let r denote $hd_2(r_i)$, let $r = (\pi, (m, \bar{\theta}, w), r')$ and $r'' = (w, \tilde{m}(\bar{\theta}), r')$, where $\tilde{m} \in \mathcal{F}_1$
$T_5 = \iota\Big(\xi\big(\bar{r}, r_i/rdc(r_i, r, r'')\big)\Big)$

$\boxed{\text{Sync Extern Msg}}$
$E_6 = \exists r_i; tail(r_i) \in \mathcal{E} \times Msg \times (\Theta \rightarrow R)$
Let r denote $tail(r_i)$, let $r = (e, (m, \bar{\theta}, w), f)$,
and $r' = (e, (m, \bar{\theta}, \emptyset), f(\tilde{m}(\bar{\theta})))$, where $\tilde{m} \in \mathcal{F}_2$
$T_6 = \iota\Big(\xi\big(\bar{r}, r_i/rdc(r_i, r, r')\big)\Big)$

Figure C.8: Events and Interactions (continued)

Bibliography

[AdB90a] Pierre America and Frank de Boer. A proof theory for a sequential version of POOL. Series of notes of the Computing Science Section, Eindhoven Univ. of Technology, Dept. Math. and Computing Science, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, 1990.

[AdB90b] Pierre America and Frank de Boer. Proving Total Correctness of Recursive Procedures. *Information and Computation*, 84(2):129–162, 1990.

[Agh86] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[Ak89] Jean Alain Ah-kee. *Operation Decomposition for Blocks and Procedures.* PhD thesis, University of Manchester, Department of Computer Science, Oxford Road, M13 9PL, Manchester UK, July 1989.

[Ame89] Pierre America. Issues in the Design of a Parallel Object-Oriented Language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[And91] Gregory R. Andrews. *Concurrent Programming: Principles and Practice.* The Benjamin/Cummings Publishing Company, 1991.

[AO91] Krzysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, 1991.

[Apt81] Krzysztof R. Apt. Ten Years of Hoare's Logic: A Survey — Part I. *ACM TOPLAS*, 3:431–483, 1981.

[AR92] Pierre America and Jan Rutten. A layered semantics for a parallel object-oriented language. *Formal Aspects of Computing*, 4:376–408, 1992.

[AS83] Gregory R. Andrews and Fred B. Schneider. Concepts and Notations for Concurrent Programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

[Ast91] Egidio Astesiano. Inductive and Operational Semantics. In E.J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, pages 51–136. Springer-Verlag, 1991.

[AvdL90] Pierre America and Frank van der Linden. A parallel object-oriented language with inheritance and subtyping. In *Proceedings of OOPSLA/ECOOP '90*, pages 161–168, 1990.

[Bac81] Ralph-Johan R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, February 1981.

[Bac88] Ralph-Johan R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(6):593–624, 1988.

[Bar84] H.P. Barendregt. *The Lambda Calculus: its Syntax and Semantics.* North Holland, 1984.

[Bat79] J.L. Bates. *A Logic for Correct Program Development.* PhD thesis, Cornell University, Department of Computer Science, 1979.

[BDMN73] Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *Simula Begin.* Auerbach, Philadelphia PA, 1973.

[BHJL86] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object Struture in the Emerald System. *ACM SIGPLAN Notices*, 21(11):78–86, 1986. Proceedings of OOPSLA '86.

[BJ82] Dines Bjorner and Cliff B. Jones. *Formal Specification and Software Development.* Prentice-Hall, 1982.

[Bor86] A. Borning. Classes versus Prototypes in Object-Oriented Languages. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference*, pages 36–40, Dallas, Texas, November 1986.

[Car47] Rudolf Carnap. *Meaning and Necessity, A Study in Semantics and Modal Logic.* University of Chicago Press, 1947. Enlarged edition: 1956.

[Car48] Rudolf Carnap. *Introduction to Semantics.* Harvard University Press, 1948.

[Car90] Denis Caromel. Concurrency: An Object-Oriented Approach. In *Proceedings of TOOLS90*, pages 183–197, 1990.

[CH74] R.H. Campbell and A.N. Habermann. The Specification of Process Synchronization by Path Expressions. *Lecture Notes in Computer Science*, 16:89–102, 1974.

[CJE83] R.L. Constable, S.D. Johnson, and C.D. Eichenlaub. An Introduction to the PL/CV2 Programming Logic. *Lecture Notes in Computer Science*, 135, 1983.

[Coo90] W. Cook. Inheritance is not Subtyping. In *Proceedings of POPL*, 1990.

[Cox86] Brad J. Cox. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, 1986.

[dB80] Jaco de Bakker. *Mathematical Theory of Program Correctness.* Prentice-Hall, 1980.

[Des84] T. Despeyroux. Executable Specification of Static Semantics. In G. Kahn, D.B. MacQueen, and G.D. Plotkin, editors, *Lecture Notes in Computer Science, 173*, pages 215–234. Springer-Verlag, 1984.

[DH72] Ole-Johan Dahl and C.A.R. Hoare. *Structured Programming*, chapter 3 (Hierarchical Program Structures). Academic Press, 1972.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[DN66] O.J. Dahl and K. Nygaard. Simula, An Algol-based Simulation Language. *Communications of the ACM*, 9:671–678, 1966.

[EC89] Richard L. Epstein and Walter A. Carnielli. *Computability: Computable Functions, Logic, and the Foundations of Mathematics.* Wadsworth & Brooks/Cole Advanced Books & Software, 1989.

[Eds90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1990.

[End72] H.B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[FJ92] L.M.G. Feijs and H.B.M. Jonkers. *Formal Specification and Design.* Cambridge University Press, 1992.

[Flo67] R.W. Floyd. Assigning Meanings to Programs. In *Proceedings of Symposia in Applied Mathematics 19*, pages 19–32, 1967.

[Fra92] Nissim Francez. *Program Verification*. Addison-Wesley, 1992.

[Fre92] Gottlob Frege. *Über Sinn und Bedeutung. (On Sense and Meaning)*, pages 100:25–50. Zeitschrift für Philosophie und philosophische Kritik, 1892. English translations in *Readings in Philosophical Analysis* (H. Feigl and W. Sellars, eds.), pp. 85–102, Appleton-Century-Crofts, NY (1949), and *Translations from the Philosophical Writings of Gottlob Frege* (P.T. Geach and M. Black, eds.), pp. 56–78, Blackwell, Oxford (1960).

[Gal86] Jean H. Gallier. *Logic for Computer Science*. Harper & Row, 1986.

[Goo51] R.L. Goodstein. *Constructive Formalism*. University College, Leicester, 1951.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Gri81] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[Hes93] Wim H. Hesselink. Proof Rules for Recursive Procedures. *Formal Aspects of Comnputing*, 5:554–570, 1993.

[Hew77] C. Hewitt. Viewing control structures as patterns of message passing. *Journal of Artificial Inteligence*, 8(3):323–364, 1977.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12:576–580, 1969.

[HP79] M.C.B. Hennessy and G.D. Plotkin. Full Abstraction for a Simple Programming Language. In *Proceedings of Mathematical Foundations of Computer Science*, pages 108–120. Springer-Verlag, 1979. Lecture Notes in Computer Science 74.

[Jon90] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, second edition, 1990.

[Jon92a] Cliff B. Jones. An object-based design method for concurrent programs. Technical Report UMCS-92-12-1, The University of Manchester, Manchester M12 9PL, England, 1992.

[Jon92b] Cliff B. Jones. The Search for Tractable Ways of Reasoning About Programs. Technical Report UMCS-92-4-4, University of Manchester, Department of Computer Science, Manchester M13 9PL, England, 1992.

[Jon93a] Cliff B. Jones. A pi-calculus semantics for an object-based design notation. *Lecture Notes in Computer Science*, 715:158–172, 1993. Proceedings of CONCUR '93.

[Jon93b] Cliff B. Jones. Constraining interference in an object-based design method. *Lecture Notes in Computer Science*, 668:136–150, 1993. Proceedings of TAPSOFT '93.

[JW74] K. Jensen and N. Wirth. *PASCAL User Manual and Report*. Springer-Verlag, 1974. Second edition.

[Kap85] Simon Mark Kaplan. Specification and Verification of Context Conditions for Programming Languages. Technical Report UIUCDCS-R-85-1232, Department of Computer Science, University of Illinois at Urbana-Champaign, 1985.

[KL89] Dennis G. Kafura and Keung Hae Lee. Inheritance in Actor Based Concurrent Object-Oriented Languages. In *Proceedings of ECOOP'89*, 1989.

[KMV$^+$90] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Rosin, and X. Rousset de Pina. Design and Implementation of an Object-Oriented, Strongly Typed Language for Distributed Applications. *JOOP*, 3(3):11–22, September-October 1990.

[Kol76] Gina Bari Kolata. Mathematical proofs: the genesis of reasonable doubt. *Science*, 192:989–990, 1976.

[Kow79] Tomasz Kowaltowski. Data structures and correctness of programs. *Journal of the ACM*, 26(2):283–301, 1979.

[LG86] Barbara H. Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press/McGraw-Hill, 1986.

[Lie87] Henry Lieberman. Concurrent Object-Oriented Programming in Act1. In Akinori Yonezawa and Mario Tokoro, editors, *Concurrent Object-Oriented Programming*, pages 9–36. MIT Press, 1987.

[Lis81] Barbara Liskov. CLU Reference Manual. *Lecture Notes in Computer Science*, 114, 1981.

[Lis87] Barbara Liskov. Data Abstraction and Hierarchy. In *OOPSLA '87 Conference Proceedings*, pages 17–34, 1987. Addendum to the Proceedings.

[LS87] Jacques Loeckx and Kurt Sieber. *The Foundations of Program Verification*. John Wiley & Sons, 1987.

[LZ74] B. Liskov and S. Zilles. Programming with abstract data types. *SIGPLAN Notices*, 9, 1974. Proceedings of ACM SIGPLAN Conference on Very High Level Languages.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[Mey90] Bertrand Meyer. Sequential and Concurrent Object-Oriented Programming. In *Proceedings of TOOLS90*, pages 17–28, 1990.

[Mey92] Bertrand Meyer. *Eiffel: the Language*. Prentice Hall, 1992.

[Mey93] Bertrand Meyer. Systematic Concurrent Object-Oriented Programming. Technical Report TR–EI–37/SC, Interactive Software Engineering, 270 Storke Road, Suite 7 Goleta CA 93117 USA, 1993.

[MK87] J. Eliot B. Moss and Walter H. Kohler. Concurrency Features for the Trellis/Owl Language. In J. Bézivin, J-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings of ECOOP '87, Lecture Notes in Computer Science, 276*, pages 171–180, Paris, France, June 1987. Springer Verlag.

[Mor82] J.M. Morris. Assignment and Linked Data Structures. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 35–41. D. Reidel Publishing Company, 1982.

[MV92] Caroll Morgan and Trevor Vickers. *On the Refinement Calculus*. Springer-Verlag, 1992.

[N$^+$63] P. Naur et al. Revised Report on the Algorithmic Language Algol 60. *Computer Journal*, 5:349–367, 1963.

[NHWG89] Morgens Nielsen, Klaus Havelund, Kim Ritter Wagner, and Chris George. The RAISE language, method and tools. *Formal Aspects of Computing*, 1(1):85–114, January–March 1989.

[OC75] Derek C. Oppen and Stephen A. Cook. Proving assertions about programs that manipulate data structures. In *Conf. Rec., 7th Ann. ACM Symp. Theory of Computing*, pages 107–116, 1975.

[Pag81] Frank G. Pagan. *Formal Specification of Programming Languages: A Panoramic Primer.* Prentice-Hall, 1981.

[Pap92] Michael Papathomas. *Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming.* PhD thesis, Faculté des Sciences de l'Université de Genève, 1992.

[Par71] D. Parnas. Information Distribution Aspects of Design Methodology. In *Proceedings of IFIP Congress.* North Holland Publishing Co., 1971.

[Par72] D. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12), December 1972.

[Plo76] Gordon D. Plotkin. A Powerdomain Conctruction. *SIAM J. Computing*, 5(3):452–487, Sept. 1976.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN 19, Department of Computer Science, Aarhus University, 1981.

[Pol65] G. Polya. *Mathematical discovery.* John Wiley & Sons, 1965. Two volumes.

[Pra65] Dag Prawitz. *Natural Deduction.* AlmQvist & Wiksell, 1965.

[Sch86] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, 1986.

[Sny86] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *OOPSLA '86 Conference Proceedings*, pages 38–45, 1986.

[Sok77] S. Sokołowski. Total correctness of procedures. In *Proceedings of the 6th Symp. Mathematical Foundations of Computer Science.* Springer-Verlag, 1977. Lecture Notes in Lecture Notes in Computer Science, Vol. 53.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press Series in Computer Science. MIT Press, Cambridge, Mass., 1977.

[Str87] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1987.

[Str88] Bjarne Stroustrup. What is Object-Oriented Programming. *IEEE Software*, pages 10–20, May 1988.

[Tar56] Alfred Tarski. *Logic, Semantics, and Meta-Mathematics.* Oxford University Press, 1956.

[Ten81] R.D. Tennent. *Principles of Programming Languages.* Prentice-Hall, 1981.

[Ten91] R.D. Tennent. *Semantics of Programming Languages.* Prentice Hall, 1991.

[TS89] Chris Tomlinson and V. Singh. Inheritance and Synchronization with Enabled Sets. In *Proceedings of OOPSLA '89, ACM SIGPLAN Notices*, pages 103–112, October 1989.

[Tur49] Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. University Mathematical Laboratory, 1949.

[US87] David Ungar and Randall B. Smith. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*, pages 227–241, Orlando, FL, Dec 1987. SigPlan Notices 22(12).

[vD91] A. van Deursen. An algebraic specification for the static semantics of Pascal. Technical Report CS-R9129, Centre for Mathematics and Computer Science — CWI, 1991.

[Weg87] Peter Wegner. The Object-Oriented Classification Paradigm. *Research Directions in Object-Oriented Programming*, pages 479–560, 1987.

[Win93] Glynn Winskell. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, 1993.

[Wir91] Martin Wirsing. Algebraic Specification: Semantics, Parametrization and Refinement. In E.J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, pages 259–318. Springer-Verlag, 1991.

[YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-Oriented Concurrent Programming in ABCL/1. In *OOPSLA '86 Conference Proceedings*, 1986.

[YT86] Yasuhiko Yokote and Mario Tokoro. The Design and Implementation of ConcurrentSmalltalk. In *OOPSLA '86 Conference Proceedings*, pages 331–340, 1986.