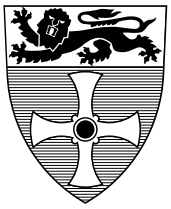
UNIVERSITY OF
NEWCASTLE



University of Newcastle upon Tyne

COMPUTING SCIENCE

Understanding programming language concepts via Operational Semantics

C. B. Jones

TECHNICAL REPORT SERIES

No. CS-TR-1046 August, 2007

TECHNICAL REPORT SERIES

No. CS-TR-1046 August, 2007

Understanding programming language concepts via Operational Semantics

Cliff B. Jones

Abstract

The origins of \formal methods" lie partly in language description (although applications of methods like VDM, RAISE or B to areas other than programming languages are probably more widely known). This paper revisits the language description task but uses operational (rather than denotational) semantics to illustrate that the crucial idea is thinking about an abstract model of something that one is trying to understand or design. A \story" is told which links together some of the more important concepts in programming languages and thus illustrates how formal semantics deepens our understanding.

Printed and published by the University of Newcastle upon Tyne,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

JONES, C. B.

Understanding programming language concepts via Operational Semantics
[By] C. B. Jones.

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Science, 2007.

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-1046)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Science. Technical Report Series. CS-TR-1046

Abstract

The origins of \formal methods" lie partly in language description (although applications of methods like VDM, RAISE or B to areas other than programming languages are probably more widely known). This paper revisits the language description task but uses operational (rather than denotational) semantics to illustrate that the crucial idea is thinking about an abstract model of something that one is trying to understand or design. A \story" is told which links together some of the more important concepts in programming languages and thus illustrates how formal semantics deepens our understanding.

About the author

Cliff Jones is currently Professor of Computing Science at Newcastle. He has spent more of his career in industry than academia. Fifteen years in IBM saw, among other things, the creation with colleagues of the Vienna Development Method. He went on to build the Formal Methods Group at Manchester University, which among other projects created the "mural" theorem proving assistant. A Senior Fellowship focused on formal (compositional) development methods for concurrent systems. In 1996 he moved to Harlequin directing some 50 developers on Information Management projects and finally became overall Technical Director before leaving to re-join academia in 1999. Cliff's interests in formal methods have now broadened to reflect wider issues of dependability. Cliff is a Fellow of the Royal Academy of Engineering, the ACM, BCS and IEE.

Suggested keywords

OPERATIONAL SEMANTICS

Understanding programming language concepts via operational semantics

Cliff B. Jones

School of Computing Science, Newcastle University, NE1 7RU, UK

Abstract. The origins of "formal methods" lie partly in language description (although applications of methods like VDM, RAISE or B to areas other than programming languages are probably more widely known). This paper revisits the language description task but uses operational (rather than denotational) semantics to illustrate that the crucial idea is thinking about an abstract model of something that one is trying to understand or design. A "story" is told which links together some of the more important concepts in programming languages and thus illustrates how formal semantics deepens our understanding.

This report is a preprint of a paper that will appear in an LNCS volume — please cite that publication [Jon07].

1 Introduction

One objective of this paper is to show how the concept of "abstract modelling" of any computer system applies to programming languages. The position taken is that a description of the semantics of such a language can not only aid understanding but can also be used to design a language which is likely to satisfy the needs of both users and compiler writers.

Computers are normally programmed in "high-level" (programming) languages (HLLs). Two important problems are

1. the correctness of programs (i.e. whether or not a program satisfies its specification) written in such a language; and
2. the correctness of the compiler that translates "source programs" into "object programs" (in "machine code").

At the root of both of these problems is the far more important issue of the design of the high-level language itself.

The designer of a programming language faces several engineering challenges — one balance that must be sought is the "level" of the language: too low a level of language increases the work of every programmer who writes programs in that language; too high a level (far from the realities of the machines on which the object programs will run) and not only is the compiler writer's task harder but it is also likely that any compiler will produce less efficient code than a programmer with closer access to the machine. A badly designed language will impair the effectiveness of both user and implementer.

The essence of what this paper addresses is the *modelling* of *concepts* in programming languages based on the firm conviction that most language issues can –and should– be thought out in terms of a semantic model long before the task of designing a complier is undertaken.

A thread –partly historical– through some of the more important concepts of programming languages is created. Little time is spent on details of the (concrete) syntax of how a concept is expressed in one or another programming language. The interest here is in concepts like "strong typing", the need for –and support of– abstract types, ways of composing programs and documenting interfaces to components, modes of parameter passing, constraining "side-effects", deciding where to allow non-determinacy, the need for files/databases, the (lack of) integration of database access languages into programming languages, and the role of objects. A particular emphasis is on issues relating to concurrency.

The main focus is not, however, on the specific features selected for study; it is the modelling concept which is promoted. Modelling tools like "environments", choosing a "small state" and (above all) *abstraction* are taught by application. The interest here is in *modelling* — not in theory for its own sake. The methods can be applied to almost any language. They are not limited to the specific language features discussed here — but an attempt has been made to provide a "thread" through those features chosen. Indeed, another objective of the paper is to show relationships between language concepts that are often treated separately.

This paper uses VDM notation for objects/functions etc. — the choice is not important and the material could be presented in other notations. The story of the move from VDL [LW69] to VDM [BBH+74,BJ78,BJ82] is told in [Jon01b]. This author's decision to move back to operational semantics is justified in [Jon03b].

1.1 Natural vs. artificial languages

The distinction between natural and formal languages is important. All humans use languages when they speak and write "prose" (or poetry). These "natural" languages have evolved and each generation of humans pushes the evolution (if not "development") further. The natural language in which this paper is written incorporates ideas and words from the languages of the many invaders of "England".[1]

In contrast to the evolving natural languages, humans have designed formal or artificial languages to communicate with computers. Different though the history –and objectives– of natural and formal languages are, there are ideas in common in the way one can study languages in either class.

The languages that are spoken by human beings were not designed by committee; they just evolved[2] — and they continue to change. The evolution process is all too obvious from the irregularities in natural languages. The task of describing natural languages is therefore very challenging but, because they have been around longer, it is precisely with natural languages that one first finds a systematic study. Charles Sanders Peirce (1839-1914) used the term "Semiotics". Crucially, he divided the study the study of languages into:

- syntax: structure
- semantics: meaning
- pragmatics: intention

Heinz Zemanek applied the terminology to programming languages in [Zem66].

It is not difficult to write syntax rules for parts of natural languages but because of their irregularity, a complete syntax is probably not a sensible objective.

It is far harder to give a semantics to a language (than to write syntactic rules). If one knows a language, another language might be explained by translating it into the known language (although nuances and beauty might be lost).

Within one language, a dictionary is used to give the meanings of words. But there is clearly a danger of an infinite loop here.

1.2 Formal languages

People designed "formal" (or artificial) languages long before computers existed (a relevant example is Boole's logic [Boo54]); but the main focus here is on

[1] No slight to the rest of the United Kingdom – just to make the link to "English" which is the name of our common language.

[2] Of course, there are a small number of exceptions like Volapük and Esperanto.

languages used to communicate with (mostly – program) computers. These languages are "formal" because they are designed (often by committees); the term artificial is used to emphasize the distinction from the evolutionary process that gives us "natural languages".

When digital computers were first programmed, it was by writing instructions in the code of the machine (indeed, in the world's first "stored program electronic computer" –the Manchester "Baby"– (1948), the program was inserted in binary via switches). Assembler programs gave some level of convenience/abstraction (e.g. offering names for instructions; later, allocating addresses for variables).

FORTRAN ("formula translator") is generally credited as the first successful programming language. It was conceived in the early 1950s and offered a number of conveniences (and some confusions) to people wanting to have a computer perform numerical computations.

The creation of FORTRAN led to the need for a translator (or compiler). The first such was built by an IBM team led by John Backus (1924–2007) in 1954–57.

There is an enormous number of high-level programming languages.[3] Jean Sammet wrote a book on "500 Programming Languages" but gave up trying to update it; a very useful history of the main languages is [Wex81]. I take the position that existing languages are *mostly poor and sometimes disastrous!* Just think for a minute about how many billions of pounds have been wasted by programs that can index outside the limits of a "stack". It is clear that these imperative[4] languages are *crucial* in the effective use of computers.[5]

- few encourage clear expression of ideas
- almost all offer gratuitous traps for the unwary
- almost none maximize the cases where a static process (compilation) can detect errors in programs

Apart from programming languages, there are many other classes of artificial languages associated with computers including those for databases. So one can see readily that there is an enormous advantage in designing good languages.

1.3 Goals of this paper

Engineers use models to understand things *before* they build them. The essence of a model is that it abstracts away from detail that might be irrelevant and facilitates focus on some specific aspect of the system under discussion.

[3] Some interesting points along the history of "imperative" languages are FORTRAN, COMTRAN, COBOL, ALGOL 60, ALGOL W, PL/I, Pascal, Simula, CPL, Modula (1, 2 and 3), BCPL, C, C++, Eiffel, Java and Csharp.

[4] As the qualification "imperative" suggests, there are other classes of HLLs. This paper mostly ignores "functional" and "logic" programming languages here.

[5] This author was on a panel on the history of languages and semantics in CMU during 2004; Vaughan Pratt asked: "(i) how much money have high-level programming languages saved the world? (ii) is there a Nobel prize in economics for answering part (i)?".

It is possible to design a language by writing a compiler for that language but it is likely –even for a small language– to be a wasteful exercise because the writer of a compiler has to work at a level of detail that prevents seeing "the wood for the trees". It might be *slightly* easier to start by writing an interpreter for the language being designed but this still requires a mass of detail to be addressed and actually introduces a specific technical danger (lack of static checking) that is discussed in more detail in Section 2.2.

In fact, many languages have been designed by a mixture of writing down example programs and sketching how they might be translated (it is made clear above that language design requires that engineering trade-offs are made between ease of expression and ease of translation). But in many cases, the first formal manifestation of a language has been its (first) compiler or interpreter.

The idea of writing a formal (syntax or) semantics is to model the language without getting involved in the detail of translation to some specific machine code. With a short document one can experiment with options. If that document is in some useful sense "formal" one can reason about the consequences of a language design.

A repeated "leitmotiv" of this paper is the need to abstract. In fact, the simplest semantic descriptions will be just *abstract interpreters*: the first descriptions in Section 3 are interpreters that are made easier to write because they abstract from much irrelevant detail. Even the syntax descriptions chosen in Section 2 are abstract in the sense that they (constructively) ignore many details of parsing etc.

1.4 A little history

The history of research on program verification is outlined in [Jon03a] (a slightly extended TR version is also available [Jon01a]) but that paper barely mentions the equally interesting story about research on the semantics of programming languages.[6] Some material which relates directly to the current paper can be found in [Jon01b,Plo04a].

There are different approaches to describing the semantics of programming languages. It is common practice to list three approaches but there are actually four and it is, perhaps, useful to split the approaches into two groups:

- model oriented
- implicit

Model-oriented approaches build a more-or-less explicit model of the state of the program; these approaches can be further divided into

- operational
- denotational

[6] There is a wealth of source material all the way from [Ste66] to recent events organized under the aegis of the (UK) "Computer Conservation Society" — see http://vmoc.museophile.org/pvs01 and http://vmoc.museophile.org/pvs04

This paper emphasizes the operational semantics approach. Any operational approach can be compared with the task of interpreting the language in question. This comparison is very clear in Section 3 below but is also the essence of the generalization explained in Section 3.1 (and applied to more language features in Sections 4–4.5). The denotational approach is akin to translating one language into another and is discussed further in Section 7.

Many authors equate "implicit" language descriptions with "axiomatic semantics" but it is actually worth also dividing this class of descriptions into

- axiomatic
- equivalences

Axiomatic descriptions provide a way of reasoning about programs written *in* a language; these approaches –and the link to model-oriented semantics– are discussed in Section 7. The idea of characterizing a programming language by equivalence laws goes back to the 1960s but is achieving more notice again in recent years.

McCarthy's paper [McC66] at the 1964 Baden-bei-Wien conference was a key step in the development of ideas on describing a language by way of an "abstract interpreter". His paper was one of the major influences on VDL [LW69]. Interestingly, McCarthy also introduced a notion of "abstract syntax" in that same paper.

There are many useful texts [Gor79,Gor88,Hen90,NN92,Win93]; books which also look at implementation aspects include [Rey98,Sco00,Wat04].

2 Delimiting the language to be defined

The ultimate interest here is in describing (or designing) the semantics of programming languages. Before one can describe a language, one needs to know what are its valid "texts". The concern with the content (as opposed to the meaning) of a language is termed syntax. In general, there will be an infinite number of possible "utterances" in any interesting language so they cannot be simply enumerated. Consequently a syntactic description technique must be capable of showing how any text in the infinite class can be generated.

Section 2.1 discusses how a *concrete syntax* can be used to define the strings of symbols which are plausible programs in a language. A concrete syntax can also tell us something about the structure the strings are trying to represent. A carefully designed concrete syntax can also be used in parsing.

Most authors define semantics in terms of concrete representations of programs but experience with defining larger languages (e.g. PL/I or Ada) –or languages with many ways of expressing same thing (e.g. C or Java)– makes clear that this becomes messy and brings gratuitous difficulties into the part of the description (the semantics) where one wants to focus on deeper questions. Therefore, *abstract syntax* descriptions are used because these focus on structure and remove the need to worry about those symbols that are only inserted in order to make parsing possible.

This is not quite the end of the story since both concrete and abstract syntax descriptions allow too many possibilities and Section 2.2 explains how to cut down the set of "valid" programs before attempting to give their semantics.

2.1 Syntax

There are many variants of notation for describing the *Concrete Syntax* of a language. It is not difficult to devise ways of specifying valid strings and most of the techniques are equivalent to Chomsky "context free" syntax notation. Most publications follow the Algol 60 report [BBG$^+$63] and use the notation which is known as "Backus Normal Form" (also known as "Backus Naur Form").[7] Such a grammar can be used to *generate* or *recognize* sentences in the language. *Parsing* also uses the grammar to associate a tree structure with the recognized sentences.

A concrete syntax gives both a way of producing the texts of programs and of parsing programs. But even for this rather simple language the concrete syntax is "fussy" in that it is concerned with those details which make it possible to parse strings (e.g. the commas, semicolons, keywords and –most notably– those things that serve to bracket strings that occur in recursive definitions). For a programming language like C or Java where there are many options, the concrete syntax becomes tedious to write; the task has to be done but, since the syntactic variants have nothing to do with semantics, basing the semantics on a concrete syntax complicates it in an unnecessary way. The first big dose of abstraction is deployed and all of the subsequent work is based on an "abstract syntax".

An abstract syntax defines a class of *objects*. In most cases, such objects are tree-like in that they are (nested) VDM composite objects. But to achieve the abstraction objective, sets, sequences and maps are used whenever appropriate.

This section builds up the Abstract Syntax of "Base" (see Appendix A where the description of the "Base" Language is presented in full).

A simple language ("Base") can be built where a program is rather like a single Algol block (with no nested blocks). A *Program* contains declarations of *Id*s as (scalar[8]) variables and a sequence of *Stmt*s which are to be executed.[9] The declarations of the variables (*vars*) maps the identifiers to their types.

$$\begin{array}{lll} Program :: & vars & : Id \xrightarrow{m} ScalarType \\ & body & : Stmt^* \end{array}$$

Notice that has –at a stroke– removed worries about the (irrelevant) order of declarations; this also ignores the delimiters between identifiers and statements since they are not an issue in the semantic description. The parsability of a language has to be sorted out; but it is not a semantic question. Here and

[7] In passing, it is hard to believe that so many programming language books today are published without a full concrete syntax for the language!

[8] The adjective "scalar" appears superfluous for now but compound variables like arrays are discussed below.

[9] Section 4 introduces *Block*s and a program can then be said to contain a single *Block* – but this is not yet needed.

elsewhere it is preferable to deal with issues separately and get things out of the way rather than complicate the semantic description. The abstraction in an abstract syntax does precisely this.

According to this abstract syntax, the smallest possible *Program* declares no variables and contains no statements, thus

$$mk\text{-}Program(\{\,\},[\,]) \in Program$$

Not much more useful is a program which declares one variable but still has no statements.

$$mk\text{-}Program(\{i \mapsto \textsc{IntTp}\},[\,]) \in Program$$

More interesting programs are given below.

There are exactly two types in this base language:

$$ScalarType = \textsc{IntTp} \mid \textsc{BoolTp}$$

Three forms of statement will serve to introduce most concepts

$$Stmt = Assign \mid If \mid While$$

$$\begin{array}{lll} Assign :: & lhs & : Id \\ & rhs & : Expr \end{array}$$

Thus, if $e \in Expr; i \in Id$

$$mk\text{-}Assign(i,e) \in Assign$$

One of the major advantages of the VDM record notation is that the *mk-Record* constructors make the sets disjoint.

Conditional execution can be defined in an *If* statement.

$$\begin{array}{lll} If :: & test & : Expr \\ & th & : Stmt^* \\ & el & : Stmt^* \end{array}$$

Thus,

$$mk\text{-}If(e,[\,],[\,]) \in If$$

A common form of repetitive execution is achieved by a *While* statement.

$$\begin{array}{lll} While :: & test & : Expr \\ & body & : Stmt^* \end{array}$$

Thus,

$$mk\text{-}While(e,[\,]) \in While$$

It is possible to illustrate the key semantic points with rather simple expressions.

$$Expr = ArithExpr \mid RelExpr \mid Id \mid ScalarValue$$

$$\begin{array}{lll} ArithExpr :: & opd1 & : Expr \\ & operator & : \textsc{Plus} \mid \textsc{Minus} \\ & opd2 & : Expr \end{array}$$

$$\begin{array}{lll} \mathit{RelExpr} :: & \mathit{opd}1 & : \mathit{Expr} \\ & \mathit{operator} & : \textsc{Equals} \mid \textsc{NotEquals} \\ & \mathit{opd}2 & : \mathit{Expr} \end{array}$$

$$\mathit{ScalarValue} = \mathbb{Z} \mid \mathbb{B}$$

No definition is provided for *Id* since one can abstract from such (concrete) details.

Thus,

$1 \in \mathit{ScalarValue}$
$i \in \mathit{Id}$
$\mathit{mk\text{-}ArithExpr}(i, \textsc{Minus}, 1) \in \mathit{Expr}$
$\mathit{mk\text{-}RelExpr}(i, \textsc{NotEquals}, 1) \in \mathit{Expr}$

And then with

$$\begin{array}{l} s_1 = \mathit{mk\text{-}If}(\mathit{mk\text{-}RelExpr}(i, \textsc{NotEquals}, 1), \\ \qquad [\mathit{mk\text{-}Assign}(i, \mathit{mk\text{-}ArithExpr}(i, \textsc{Minus}, 1))], \\ \qquad [\mathit{mk\text{-}Assign}(i, \mathit{mk\text{-}ArithExpr}(i, \textsc{Plus}, 1))]) \end{array}$$

$s_1 \in \mathit{Stmt}$

And, finally

$\mathit{mk\text{-}Program}(\{i \mapsto \textsc{IntTp}\}, [s_1]) \in \mathit{Program}$

It it worth noting that the supposed distinction between arithmetic and relational expressions is not policeable at this point; this gets sorted out in the next section when type information is used.

2.2 Eliminating invalid programs

Before moving to look at semantics, it is worth eliminating as many invalid programs as possible. For example, it is easy to recognize that

$$\mathit{mk\text{-}ArithExpr}(i, \textsc{Minus}, \mathbf{true})$$

contains an error of types. This is easy to check because it requires no "context" but in a program which (like the final one in the preceding section) declares only the identifier i, one would say that

$$\mathit{mk\text{-}Assign}(j, \mathit{mk\text{-}ArithExpr}(i, \textsc{Minus}, 1))$$

has no meaning because it uses an undeclared variable name. Similarly, uses of variables should match their declarations and, if i is declared to be an integer,

$$\mathit{mk\text{-}Assign}(i, \mathbf{true})$$

makes no sense.

A function is required which "sorts the sheep from the goats": a program which is type correct is said to be "well formed". A function which delivers

either **true** or **false** is a predicate. It is not difficult to define a predicate which delivers **true** if type information is respected and **false** otherwise.

In order to bring the type information down from the declarations, the signature of the inner predicates must be

$$wf\text{-}Stmt : Stmt \times TypeMap \to \mathbb{B}$$
$$wf\text{-}Stmt(s, tpm) \triangleq \cdots$$

(all of these predicates are given names starting $wf\text{-}\cdots$ as a reminder that they are concerned with well-formedness) with the following "auxiliary objects"

$$TypeMap = Id \xrightarrow{m} ScalarType$$

The top-level predicate is defined

$$wf\text{-}Program : Program \to \mathbb{B}$$
$$wf\text{-}Program(mk\text{-}Program(vars, body)) \triangleq wf\text{-}StmtList(body, vars)$$

All that remains to be done is to define the subsidiary predicates. Those for *Stmt* (and for *Expr*) have to be recursive because the objects themselves are recursive. Thus

$$wf\text{-}StmtList : (Stmt^*) \times TypeMap \to \mathbb{B}$$
$$wf\text{-}StmtList(sl, tpm) \triangleq \forall i \in \mathbf{inds}\ sl \cdot wf\text{-}Stmt(sl(i), tpm)$$

Then:

$$wf\text{-}Stmt : Stmt \times TypeMap \to \mathbb{B}$$
$$wf\text{-}Stmt(s, tpm) \triangleq \cdots$$

is most easily given by cases below[10]

$$wf\text{-}Stmt(mk\text{-}Assign(lhs, rhs), tpm) \triangleq$$
$$\quad lhs \in \mathbf{dom}\ tpm \wedge$$
$$\quad c\text{-}tp(rhs, tpm) = tpm(lhs)$$

$$wf\text{-}Stmt(mk\text{-}If(test, th, el), tpm) \triangleq$$
$$\quad c\text{-}tp(test, tpm) = \textsc{BoolTp} \wedge$$
$$\quad wf\text{-}StmtList(th, tpm) \wedge wf\text{-}StmtList(el, tpm)$$

$$wf\text{-}Stmt(mk\text{-}While(test, body), tpm) \triangleq$$
$$\quad c\text{-}tp(test, tpm) = \textsc{BoolTp} \wedge$$
$$\quad wf\text{-}StmtList(body, tpm)$$

[10] This is using the VDM pattern matching trick of writing a "constructor" in a parameter list.

The auxiliary function to compute the type of an expression (*c-tp* used above) is defined as follows

$$\textit{c-tp} : \textit{Expr} \times \textit{TypeMap} \to (\textsc{IntTp} \mid \textsc{BoolTp} \mid \textsc{Error})$$

$$\textit{c-tp}(e, \textit{tpm}) \triangleq \text{given by cases below}$$

$$\textit{c-tp}(\textit{mk-ArithExpr}(e1, \textit{opt}, e2), \textit{tpm}) \triangleq$$
if $\textit{c-tp}(e1, \textit{tpm}) = \textsc{IntTp} \wedge \textit{c-tp}(e2, \textit{tpm}) = \textsc{IntTp}$
then IntTp
else Error

$$\textit{c-tp}(\textit{mk-RelExpr}(e1, \textit{opt}, e2), \textit{tpm}) \triangleq$$
if $\textit{c-tp}(e1, \textit{tpm}) = \textsc{IntTp} \wedge \textit{c-tp}(e2, \textit{tpm}) = \textsc{IntTp}$
then BoolTp
else Error

For the base cases:

$$e \in \textit{Id} \;\Rightarrow\; \textit{c-tp}(e, \textit{tpm}) = \textit{tpm}(e)$$

$$e \in \mathbb{Z} \;\Rightarrow\; \textit{c-tp}(e, \textit{tpm}) = \textsc{IntTp}$$

$$e \in \mathbb{B} \;\Rightarrow\; \textit{c-tp}(e, \textit{tpm}) = \textsc{BoolTp}$$

Because they are dealing with the type information in the context of single statements and expressions, such a collection of predicates and functions are referred to as the "context conditions" of a language. They correspond to the type checking done in a compiler. Just as there, it is not always so clear how far to go with static checking (e.g. would one say that a program which included an infinite loop had no meaning?)

The issue of whether or not a language is "strongly typed" is important and this issue recurs repeatedly.

3 Semantics and abstract interpreters

This section explains the essential idea of presenting semantics via an abstract interpreter. This is first done in terms of functions. The need –and notation– for generalizing this to relations follows.

3.1 Presenting operational semantics by rules

The essence of any imperative language is that it changes some form of "state": programs have an effect. For a procedural programming language the state notion normally contains an association (sometimes indirect) between variable names

and their values (a "store"). In the simple language considered in this section, the main "semantic object" is

$$\Sigma = Id \xrightarrow{m} ScVal$$

Thus $\sigma \in \Sigma$ is a single "state"; Σ is the set of all "States".

The fundamental idea is that executing a statement will transform the state — this can be most obviously modelled as a function:

$$exec : Stmt \times \Sigma \to \Sigma$$
$$exec(s, \sigma) \triangleq \ldots$$

Such a function can be presented one case at a time by using the VDM constructors as pattern matching parameters.

Expression evaluation has the type

$$eval : Expr \times \Sigma \to ScVal$$
$$eval(e, \sigma) \triangleq \ldots$$

and can again be defined one case at a time.

Remember that only *Program*s that satisfy the context conditions are to be given semantics. This restriction implies that types of arguments match the operators "during execution"; similarly, the type of the expression of the value evaluated in *rhs* must match the *lhs* variable and the type of any value in a conditional or while statement must be Boolean.

The recursive function style is intuitive but there is a serious limitation: it does not handle non-determinism! Non-determinism can arise in many ways in programming languages:

- order of expression evaluation is a nasty example
- specific non-deterministic constructs
- parallelism

It is really the third of these which is most interesting and is the reason for facing non-determinism from the beginning.

The issues can be illustrated with a tiny example language. Later sections (notably, Sections 5.4, and 6) make clear that key concepts are establishing threads (of computation), the atomicity with which threads can merge and explicit synchronization between threads. In the simple language that follows, a *Program* consists of exactly two threads, assignment statements are (unrealistically) assumed to be atomic and no explicit thread synchronization mechanisms are offered. This gives rise to the following abstract syntax.

$$\begin{array}{lll} Program :: & left & : Assign^* \\ & right & : Assign^* \end{array}$$

In order to give a semantics for any such language, one needs to accept that it is necessary to think in terms of relations: one starting state can legitimately give rise to (under the same program) different final states. Moreover, the semantic

relation has to be between "configurations" which record, as well as the store, the program which remains to be executed. Thus:

$$\xrightarrow{p} : \mathcal{P}((Program \times \Sigma) \times (Program \times \Sigma))$$

The two semantic rules which follow show exactly how nondeterminism arises because a program which has non-empty statement lists in both branches will match the hypotheses of both rules.

$$\frac{(s, \sigma) \xrightarrow{s} \sigma'}{mk\text{-}Program([s] \frown restl, r), \sigma) \xrightarrow{p} mk\text{-}Program(restl, r), \sigma')}$$

$$\frac{(s, \sigma) \xrightarrow{s} \sigma'}{(mk\text{-}Program(l, [s] \frown restr), \sigma) \xrightarrow{p} (mk\text{-}Program(l, restr), \sigma')}$$

Finally, a program is complete when both of its branches have terminated.

$$\frac{}{mk\text{-}Program([\,], [\,]), \sigma) \longrightarrow \sigma}$$

The interleaving of the two threads is achieved by the semantics being "small step": the whole configuration is subject to matching of the rules after each step.

Alternatively, one might want to add to the language an explicit construct with which a programmer can determine the level of atomicity:

$$\begin{array}{lrcl} Program & :: & left & : (Assign \mid Atomic)^* \\ & & right & : (Assign \mid Atomic)^* \end{array}$$

$$Atomic :: Assign^*$$

The relevant semantic rule becomes:

$$\frac{(sl, \sigma) \xrightarrow{sl} \sigma'}{(mk\text{-}Program([mk\text{-}Atomic(sl)] \frown restl, r), \sigma) \xrightarrow{p} (mk\text{-}Program(restl, r), \sigma')}$$

The semantic transition relation for expressions could be given as:

$$\xrightarrow{e} : \mathcal{P}((Expr \times \Sigma) \times ScVal)$$

Strictly, there are no constructs in this language that make expression evaluation non-deterministic so it would be possible to stick with functions.

It would also be possible to illustrate how it is necessary to extend the idea of a functional semantics (to one using relations) by looking at a specific non-deterministic construct such as Dijkstra's "guarded command" [Dij76]. Just as with the concurrent assignment threads, an explicit relation $((Stmt \times \Sigma) \times \Sigma)$ has to be used.

This way of presenting operational semantics rules follows Plotkin's [Plo81][11] and is often referred to as "Plotkin rules". Using this style for an entirely deterministic language gives the first full semantics in this paper: the attentive reader ought be able to understand the definition in Appendix A.

[11] Republished as [Plo04b] — see also [Plo04a,Jon03b].

3.2 Ways of understanding the rules

There are several different views of the operational semantics rules used in the previous section. For a given starting store and program text, the rules can be used to construct a diagram whose root is that initial program and store. Each rule that matches a particular configuration can then be used to define successor configurations. Because more than one rule might match, the inherent non-determinism is seen where there is more than one outgoing arc from a particular configuration.

For our purposes, it is more interesting to view the rules as providing an inductive definition of the $\xrightarrow{s}$ relation. This leads on naturally to the use of such rules in proofs. At the specific level one might write:

	from $\sigma_0 = \{x \mapsto 9, y \mapsto 1\};\ \sigma_1 = \{x \mapsto 3, y \mapsto 1\}$	
1	$(3, \sigma_0) \xrightarrow{e} 3$	$\xrightarrow{e}$
2	$(mk\text{-}Assn(x, 3), \sigma_0) \xrightarrow{s} \sigma_1$	1, $\xrightarrow{s}$
3	$(x, \sigma_1) \xrightarrow{e} 3$	$\xrightarrow{e}$
4	$(mk\text{-}Assn(y, x), \sigma_1) \xrightarrow{s} \{x \mapsto 3, y \mapsto 3\}$	3, $\xrightarrow{s}$
	infer $([mk\text{-}Assn(x, 3), mk\text{-}Assn(y, x)], \sigma_0) \xrightarrow{sl} \{x \mapsto 3, y \mapsto 3\}$	2, 4, $\xrightarrow{sl}$

but it is more interesting to produce general proofs of the form:

	from $pre\text{-}prog(\sigma_0); (prog, \sigma_0) \xrightarrow{p} \sigma_f$
n	$\vdots$
	infer $post\text{-}prog(\sigma_0, \sigma_f)$

where the intermediate steps are justified either by a semantic rule or by rules of the underlying logic. This is essentially what is done in [CM92,KNvO$^+$02] and very clearly in [CJ07].

3.3 Developments from this base

There are many ways to extend the language description in Appendix A that do not require any further modelling concepts — they would just constitute applications of the ideas above to cover other programming language concepts. There are a few topics, however, which deserve mention before moving on to Section 4.

Looking back at the semantic rule for $(mk\text{-}Program(vars, body)) \xrightarrow{p}$ DONE in Appendix A it could be observed that there is no point in running a program! Execution leaves no visible trace in the world outside the program because the block structure "pops all of the variables off the stack" at the end of execution. Adding input/output is however a simple exercise. For the abstract syntax:

$Stmt = \ldots \mid Write$

$$\mathit{Write} :: \mathit{value} : \mathit{Expr}$$

The relevant context condition is:

$$\mathit{wf\text{-}Stmt}(\mathit{mk\text{-}Write}(\mathit{value}), \mathit{tps}) \triangleq \mathit{tp}(\mathit{value}, \mathit{tps}) = \textsc{IntTp}$$

The essence of modelling an imperative language is to put in the "state" those things that can be changed. So for output the state needs to be a composite object that embeds the "store" in "state" but adds an abstraction of an output file.

$$\begin{array}{lll} \Sigma :: & \mathit{vars} & : \mathit{Id} \xrightarrow{m} \mathit{ScVal} \\ & \mathit{out} & : \mathbb{Z}^* \end{array}$$

The semantic rule for the new statement is:

$$\frac{(\mathit{value}, \sigma) \xrightarrow{e} v}{(\mathit{mk\text{-}Write}(\mathit{value}), \sigma) \xrightarrow{s} \mathit{mk}\text{-}\Sigma(\sigma.\mathit{vars}, \sigma.\mathit{out} \frown [v])}$$

Unfortunately, some other rules have to be revised because of the change in Σ — but only a few (routine) changes.

$$\frac{(\mathit{rhs}, \sigma) \xrightarrow{e} v}{(\mathit{mk\text{-}Assign}(\mathit{lhs}, \mathit{rhs}), \sigma) \xrightarrow{s} \mathit{mk}\text{-}\Sigma(\sigma.\mathit{vars} \dagger \{\mathit{lhs} \mapsto v\}, \sigma.\mathit{out})}$$

$$\frac{e \in \mathit{Id}}{(e, \sigma) \xrightarrow{e} \sigma.\mathit{vars}(e)}$$

Covering input statements should be obvious and an extension to linking programs to more complex file stores –or even databases– not difficult. (The whole question of why programming languages do not directly embed database concepts is interesting. It is certainly not difficult to view relations as datatypes and concepts of typing could be clarified thereby. The most interesting aspects concern the different views of concurrency and locking: this is briefly touched on in Section 5.4.)

Another topic that offers interesting language design trade-offs is statements for repetition. Whilst it is true that **while** statements suffice in that they make a language "Turing complete", many other forms of repetitive statement are found in programming languages.

The intuition here is that programmers want to deal with regular collections of data in analogous ways. Thus, it would be useful to study **for** statements in connection with arrays. (But the more interesting possibilities for the semantics of arrays come after parameter passing by location (aka by reference) has been covered in Section 4.) A simple statement might be:

$$\mathit{Stmt} = \cdots \mid \mathit{For}$$

$$\begin{array}{lll} \mathit{For} :: & \mathit{control} & : \mathit{Id} \\ & \mathit{limit} & : \mathit{Expr} \\ & \mathit{body} & : \mathit{Stmt}^* \end{array}$$

$$\frac{(limit, \sigma) \xrightarrow{e} limitv \quad ((control, limitv, body), \sigma \dagger \{control \mapsto 1\}) \xrightarrow{i} \sigma'}{(mk\text{-}For(control, limit, body), \sigma) \xrightarrow{s} \sigma'}$$

Where, the auxiliary concept of iteration is defined as follows:

$$\xrightarrow{i} : \mathcal{P}(((Id \times \mathbb{Z} \times (Stmt^*)) \times \Sigma) \times \Sigma)$$

$$\frac{\sigma(control) > limitv}{((control, limitv, body), \sigma) \xrightarrow{i} \sigma}$$

$$\frac{\sigma(control) \leq limitv \quad (body, \sigma) \xrightarrow{sl} \sigma' \quad ((control, limitv, body), \sigma' \dagger \{control \mapsto \sigma'(control) + 1\}) \xrightarrow{i} \sigma''}{((control, limitv, body), \sigma) \xrightarrow{s} \sigma''}$$

Not only are there many alternative forms of **for** construct to be investigated but they also point to both questions of scoping (e.g. should the control variable be considered to be a local declaration) and interesting semantic issues of equivalences a programmer might expect to hold between different forms. It is also tempting to look at parallel forms because –from their earliest manifestation in FORTRAN– **for** statements have frequently over-specified irrelevant sequential constraints.

4 Scopes and parameter passing

This section considers the problem of variables having scope and some different ways in which parameters can be passed to functions. The language ("Blocks") used is still in the ALGOL/Pascal family because it is here that these problems were first thought out. A firm understanding of these concepts makes for greater appreciation of what is going on in an object-oriented language (see Section 6). A bonus of this study is that one of the key modelling techniques is explained (in the setting where is first arose).

4.1 Nested blocks

Sections 4.2–4.5 cover the modelling of functions and procedures which facilitate the use of the same piece of code from different places in a program containing them. Before addressing this directly, nested blocks can be added to the language of Section 3. Here the pragmatics of the feature are the ability to use the same identifier with different meaning within the same program (a nested block might for example be designed by a different programmer than the one who wrote the containing text; it would be tedious and error prone to have to change all identifiers to be unique throughout a program).

In the examples throughout Section 4, a concrete syntax is used that surrounds the declarations and statements with **begin** $\cdots$ **end**. Consider the example (concrete) program in Figure 1. The inner block defines its own scope and the a declared (to be of type **integer**) there is distinct from the variable of the same name (declared to be of type **bool**) in the outer block. Although the two assignments to the name a imply that the type is different, both are correct because two different variables are in play. One could insist that programmers avoid the reuse of names in this way but this would not be a kind restriction.

```
program
  begin
  bool a; int i; int j;
  if i = j then
    begin
    int a;
    a := 1
    end
  fi
  a := false
  end
end
```

Fig. 1. Scope for *Block*

It is easy to add an option to *Stmt* that allows such nesting

$Stmt = \cdots \mid Block$

One might then expect to say that a *Program* is a *Block* but allowing it to be a single *Stmt* leaves that possibility open and adds a slight generalization. Thus the abstract syntax might change (from Section 3) in the following ways:

$Program = Stmt$

$Stmt = Assign \mid If \mid While \mid Block$

$$\begin{array}{lll} Block & :: & vars : Id \xrightarrow{m} ScalarType \\ & & body : Stmt^* \end{array}$$

The context conditions only change as follows:

$wf\text{-}Program : Program \to \mathbb{B}$

$wf\text{-}Program(s) \triangleq wf\text{-}Stmt(s, \{\})$

$wf\text{-}Block : Block \times TypeMap \to \mathbb{B}$

$wf\text{-}Block(mk\text{-}Block(vars, body), tpm) \triangleq wf\text{-}StmtList(body, tpm \dagger vars)$

Notice that it is obvious from this that all statements in a list are checked (for type correctness) against the same *TypeMap*: even if the ith statement is a block, the $i+1$st statement has the same potential set of variables as the $i-1$st statement.

The semantics for a *Block* have to show that local variables are distinct from those of the surrounding text. On entry to *mk-Block*(*vars*, *body*) this just requires that each identifier in the domain of *vars* gets initialized. Leaving the block is actually more interesting. After the execution of *body* has transformed σ_i into σ'_i, the state after execution of the whole block contains the values of the (not re-declared) local variables from σ'_i but it is also necessary to recover the values from σ of variables which were masked by the local names.

$$\{id \mapsto \sigma'_i(id) \mid id \in \mathbf{dom}\,\sigma \wedge id \notin \mathbf{dom}\,vars\} \cup \\ \{id \mapsto \sigma(id) \mid id \in \mathbf{dom}\,\sigma \wedge id \in \mathbf{dom}\,vars)\}$$

Thus (using the VDM map restriction operator):

$$\sigma_i = \sigma \dagger (\{id \mapsto 0 \mid id \in \mathbf{dom}\,vars \wedge vars(id) = \textsc{IntTp}\} \cup \\ \{id \mapsto \mathbf{true} \mid id \in \mathbf{dom}\,vars \wedge vars(id) = \textsc{BoolTp}\})$$
$$\frac{(body, \sigma_i) \xrightarrow{sl} \sigma'_i}{(\textit{mk-Block}(vars, body), \sigma) \xrightarrow{s} ((\mathbf{dom}\,\sigma - \mathbf{dom}\,vars) \lhd \sigma'_i) \cup (\mathbf{dom}\,vars \lhd \sigma)}$$

Notice that $\mathbf{dom}\,\sigma$ is unchanged by any *Stmt* (even a *Block*).

4.2 Avoiding non-deterministic side effects

The pragmatics for adding functions (or procedures) to a language are for re-use of code: one can pull out a piece of algorithm to be used frequently — not so much for space since a compiler might anyway "in-line" it — but as a way of making sure that it is modified everywhere at once if it has to be changed.

Functions again bring a form of local naming. Unless a language designer is careful, they can also bring a very messy side effect problem. If a function can reference non-local variables, a call to the function can give rise to side effects. In cases where there is more than one function call in an expression, the order in which the function calls occur can influence the final result of the program.[12]

An obvious way to avoid the non-determinism caused by functions referring to non-local variables is to set up the context conditions to ban global access: the only identifiers to which a function can refer are either names of the parameters or those of newly defined local variables.

The form of *Function* here has an explicit result clause[13] at the end of its text:

[12] Pascal's rule that such a program would be in error is the worst of all worlds for the language specifier: one has to show the non-determinism in order to ban it!

[13] This avoids a goto-like jump out of phrase structure.

```
function f(int a) int
    a := 1;
    result(a + 7)
end
```

A small program to illustrate scope definition in functions might be as in Figure 2. As in Figure 1, there are two distinct uses of the name a and the arguments against asking the programmer to take the strain are stronger since functions might well be taken from another source.

```
program
  begin
  bool a;
  function f(int a) int
    a := 1; ...
    result(7)
   end
  ...
  a := true
  end
end
```

Fig. 2. Scope for *Fun*

One might build such a definition around an abstract syntax:

$$\begin{array}{lll} \mathit{Block} :: & \mathit{vars} & : \mathit{Id} \xrightarrow{m} \mathit{ScalarType} \\ & \mathit{fns} & : \mathit{Id} \xrightarrow{m} \mathit{FnDefn} \\ & \mathit{body} & : \mathit{Stmt}^* \end{array}$$

$$\begin{array}{lll} \mathit{FnDefn} :: & \mathit{type} & : \mathit{ScalarType} \\ & \mathit{parml} & : \mathit{ParmInfo}^* \\ & \mathit{body} & : \mathit{Stmt}^* \\ & \mathit{result} & : \mathit{Expr} \end{array}$$

$$\begin{array}{lll} \mathit{ParmInfo} :: & \mathit{name} & : \mathit{Id} \\ & \mathit{type} & : \mathit{ScalarType} \end{array}$$

$$\mathit{Expr} = \cdots \mid \mathit{FnCall}$$

$$\begin{array}{lll} \mathit{FnCall} :: & \mathit{fn} & : \mathit{Id} \\ & \mathit{argl} & : \mathit{Expr}^* \end{array}$$

There are interesting semantic issues even with such a restrictive form of function call. First, there is the language decision about how to return a value from a function: different languages introduce a statement **return**(e); or assign to the name of the function as in $f \leftarrow e$; or just allow that an expression is written in place of a statement. Notice that all of these approaches only support

the return of a single value. One can mitigate the impact of this restriction with the introduction of "records" (see Section 5.1). Another way around the restriction is by using an appropriate parameter passing mechanism (e.g. pass by location) — see Section 4.3.

There is a whole collection of questions around modes of parameter passing, but these are deferred to Sections 4.3–4.5.

Turning next to the description of such a language, the result of banning side effects is that the semantic transition relation for expressions remains:

$$\xrightarrow{e} : \mathcal{P}((Expr \times \Sigma) \times ScVal)$$

(rather than also having a Σ on the right of the main relation).

When writing a semantics for any feature that can use a named piece of text from many places, there is the question of how that text is located when it is "called". Here, it is stored in some form of environment (*Env* in Section 4.3); a slightly different approach is illustrated for the object-oriented language in Section 6.

It has however been made clear by the title of this section that a major language design issue around functions is finding ways to avoid unpredictable side effects. An alternative way of avoiding the non-determinism from side-effects is to have "procedure calls" (rather than functions which can be referenced in an expressions).

4.3 Parameter passing

In Section 3.1, variables were declared for the whole *Program*; Section 4.1 introduced nested blocks and Section 4.2 looked at functions with no external references. It is now time to move on to begin a look at various forms of parameter passing — call-by-location (call by reference) is covered first followed by a look at other modes in Section 4.5.

There are real engineering trade-offs here. Passing parameters by location offers a way to change values in the calling environment.[14] This is particularly useful for programs which manipulate and reshape tree structures. But passing parameters by location introduces aliasing problems which complicate formal reasoning and debugging alike.

So the Abstract Syntax for Function definitions might be

$$\begin{array}{lll} Fun :: & returns & : ScalarType \\ & params & : Id^* \\ & paramtps & : Id \xrightarrow{m} ScalarType \\ & body & : Stmt^* \\ & result & : Expr \end{array}$$

[14] In passing, it is worth noting that this facilitates returning more than one value from a single function call.

and the relevant context condition[15]

$$wf\text{-}Fun : Fun \times Types \to \mathbb{B}$$

$$\begin{array}{l} wf\text{-}Fun(mk\text{-}Fun(returns, params, paramtps, body, result), tps) \quad \triangleq \\ \quad uniquel(params) \wedge \\ \quad \mathbf{elems}\, params = \mathbf{dom}\, paramtps \wedge \\ \quad tp(result) = returns \wedge \\ \quad wf\text{-}StmtList(body, tps \dagger paramtps) \end{array}$$

To define the way that a *Block* builds the extended *Types*

$$Types = Id \xrightarrow{m} Type$$

$$Type = ScalarType \mid FunType$$

one needs

$$\begin{array}{lcl} FunType :: & returns & : ScalarType \\ & paramtpl & : ScalarType^* \end{array}$$

The Abstract Syntax for *Block* is

$$\begin{array}{lcl} Block :: & vars & : Id \xrightarrow{m} ScalarType \\ & funs & : Id \xrightarrow{m} Fun \\ & body & : Stmt^* \end{array}$$

and the resulting Context Condition is[16]

---

[15] The function *uniquel* can be defined:

$$uniquel : (X^*) \to \mathbb{B}$$

$$uniquel(l) \quad \triangleq \quad \forall i, j \in \mathbf{inds}\, l \cdot i \neq j \;\Rightarrow\; l(i) \neq l(j)$$

[16] The auxiliary function *apply* is defined:

$$apply : (X^*) \times (X \xrightarrow{m} Y) \to (Y^*)$$

$$\begin{array}{l} apply(l, m) \quad \triangleq \\ \quad \mathbf{if}\ l = [\,] \\ \quad \mathbf{then}\ [\,] \\ \quad \mathbf{else}\ [m(\mathbf{hd}\, l)] \frown apply(\mathbf{tl}\, l, m) \end{array}$$

$$
\begin{array}{l}
\mathit{wf\text{-}Stmt} : \mathit{Block} \times \mathit{Types} \to \mathbb{B} \\
\\
\mathit{wf\text{-}Stmt}(\mathit{mk\text{-}Block}(\mathit{vars}, \mathit{funs}, \mathit{body}), \mathit{tps}) \quad \triangleq \\
\quad \textbf{dom}\ \mathit{vars} \cap \textbf{dom}\ \mathit{funs} = \{\,\} \wedge \\
\quad \textbf{let}\ \mathit{var\text{-}tps} = \mathit{tps} \dagger \mathit{vars}\ \textbf{in} \\
\quad \textbf{let}\ \mathit{fun\text{-}tps} = \\
\quad\quad \{f \mapsto \mathit{mk\text{-}FunType}(\mathit{funs}(f).\mathit{returns}, \\
\quad\quad\quad\quad \mathit{apply}(\mathit{funs}(f).\mathit{params}, \mathit{funs}(f).\mathit{paramtps})) \mid \\
\quad\quad\quad f \in \textbf{dom}\ \mathit{funs}\}\ \textbf{in} \\
\quad \forall f \in \textbf{dom}\ \mathit{funs} \cdot \mathit{wf\text{-}Fun}(\mathit{funs}(f), \mathit{var\text{-}tps}) \\
\quad \mathit{wf\text{-}StmtList}(\mathit{body}, \mathit{var\text{-}tps} \dagger \mathit{fun\text{-}tps})
\end{array}
$$

The next task is to look closely at parameter passing. As indicated at the beginning of this section, this is done in an ALGOL (or Pascal) framework.

A small program which illustrates the way functions are handled in the "Blocks" language is given in Figure 3. Functions are declared to have a type; their definition text contains a body which is a sequence of statements to be executed; the text ends with an explicit result expression.

Here, the non-determinism discussed in Section 4.2 can be avoided by limiting functions to be called only in a specific way.

$$v := f(i)$$

So the Abstract Syntax for a *Call* statement is:

$$
\begin{array}{lll}
\mathit{Call} :: \mathit{lhs} & : & \mathit{Id} \\
\quad\quad \mathit{fun} & : & \mathit{Id} \\
\quad\quad \mathit{args} & : & \mathit{Id}^*
\end{array}
$$

The Context Condition is

$$
\begin{array}{l}
\mathit{wf\text{-}Stmt}(\mathit{mk\text{-}Call}(\mathit{lhs}, \mathit{fun}, \mathit{args}), \mathit{tps}) \quad \triangleq \\
\quad \mathit{lhs} \in \textbf{dom}\ \mathit{tps} \wedge \\
\quad \mathit{fun} \in \textbf{dom}\ \mathit{tps} \wedge \\
\quad \mathit{tps}(\mathit{fun}) \in \mathit{FunType} \wedge \\
\quad \mathit{tps}(\mathit{lhs}) = (\mathit{tps}(\mathit{fun})).\mathit{returns} \wedge \\
\quad \textbf{len}\ \mathit{args} = \textbf{len}\ (\mathit{tps}(\mathit{fun})).\mathit{paramtpl} \wedge \\
\quad \forall i \in \textbf{inds}\ \mathit{args} \cdot \mathit{tp}(\mathit{args}(i), \mathit{tps}) = ((\mathit{tps}(\mathit{fun})).\mathit{paramtpl})(i)
\end{array}
$$

In Figure 3, within f, both x, i refer to the same "location". Changing the call to be as in Figure 4 results in the situation, within f, that all of x, y, i refer to the same "location". Notice that $j := f(i + j, 3)$ cannot be allowed for "by location".

The basic modelling idea is to split Σ of Section 3.1 into two mappings: *Env* and Σ:

$$\mathit{Env} = \mathit{Id} \xrightarrow{m} \mathit{Den}$$

$$\mathit{Den} = \mathit{ScalarLoc} \mid \mathit{FunDen}$$

```
program
  begin
  int i, j, k;
  function f(int x, int y) int
    i := i + 1; x := x + 1; y := y + 1 /* print(x, y) */
    result(7)
  end
  ...
  i := 1; j := 4;
  k := f(i, j) /* print(i, j) */
  end
end
```

Fig. 3. Parameter example (i)

```
program
  begin
  int i, j, k;
  function f(int x, int y) int
    i := i + 1; x := x + 1; y := y + 1 /* print(x, y) */
    result(7)
  end
  ...
  i := 1; j := 4;
  k := f(i, i) /* print(i, j) */
  end
end
```

Fig. 4. Parameter example (ii)

$$\Sigma = ScalarLoc \xrightarrow{m} ScalarValue$$

So now the basic semantic relations become:

$$\xrightarrow{s} : \mathcal{P}((Stmt \times Env \times \Sigma) \times \Sigma)$$

$$\xrightarrow{e} : \mathcal{P}((Expr \times Env \times \Sigma) \times ScalarValue)$$

One can now talk about the left-hand (of an assignment) value of an identifier and separating this out will pay off in Section 4.4 when dealing with references to elements of arrays.Left-hand values occur elsewhere so it is worth having a way of deriving them.

$$\xrightarrow{lhv} : \mathcal{P}((VarRef \times Env \times \Sigma) \times ScalarLoc)$$

$$\frac{e \in Id}{(e, env, \sigma) \xrightarrow{lhv} env(e)}$$

in terms of which, accessing the "right hand value" can be defined:

$$\frac{\begin{array}{l} e \in Id \\ (e, env, \sigma) \xrightarrow{lhv} l \end{array}}{(e, env, \sigma) \xrightarrow{e} \sigma(l)}$$

The left hand value is used to change a value in –for example– assignments:

$$\frac{\begin{array}{l} (lhs, env, \sigma) \xrightarrow{lhv} l \\ (rhs, env, \sigma) \xrightarrow{e} v \end{array}}{(mk\text{-}Assign(lhs, rhs), env, \sigma) \xrightarrow{s} \sigma \dagger \{l \mapsto v\}}$$

Most rules just pass on *env*:

$$\xrightarrow{sl} : \mathcal{P}(((Stmt^*) \times Env \times \Sigma) \times \Sigma)$$

$$\frac{\begin{array}{l} (s, env, \sigma) \xrightarrow{s} \sigma' \\ (rest, env, \sigma') \xrightarrow{sl} \sigma'' \end{array}}{([s] \curvearrowright rest, env, \sigma) \xrightarrow{sl} \sigma''}$$

Similarly

$$\xrightarrow{e} : \mathcal{P}((Expr \times Env \times \Sigma) \times ScalarValue)$$

$$\frac{\begin{array}{l} (e1, env, \sigma) \xrightarrow{e} v1 \\ (e2, env, \sigma) \xrightarrow{e} v2 \end{array}}{(mk\text{-}ArithExpr(e1, \textsc{Plus}, e2), env, \sigma) \xrightarrow{e} v1 + v2}$$

We now look at how to create and modify *env*. Postponing the question of functions for a moment, the overall shape of the meaning of a *Block* is:

$$\frac{\begin{array}{l}(\mathit{varenv}, \sigma') = \text{/* find and initialize free locations */}\\ \mathit{funenv} = \text{/* create function denotations */}\\ \mathit{env}' = \mathit{env} \dagger \mathit{varenv} \dagger \mathit{funenv}\\ (\mathit{body}, \mathit{env}', \sigma') \xrightarrow{sl} \sigma''\end{array}}{(\mathit{mk\text{-}Block}(\mathit{vars}, \mathit{funs}, \mathit{body}), \mathit{env}, \sigma) \xrightarrow{s} (\mathbf{dom}\,\sigma) \lhd \sigma''}$$

The cleaning up of the locations from σ'' might look "fussy" but it pays off in compiler proofs where the designer will probably want to re-use locations in a stack discipline. The first hypothesis of this rule can be completed to

$$(\mathit{varenv}, \sigma') = \mathit{newlocs}(\mathit{vars}, \sigma)$$

where the auxiliary function *newlocs* creates an initial state for the initial values of a sufficient number of locations for each identifier declared in *vars*: each is initialized appropriately (a formal definition is in Appendix B).

Function denotations contain the information about a function which is needed for its execution. There is one final issue here: consider the program in Figure 5. The non-local reference to a within the function f must refer to the lexically embracing variable and not to the one at the point of call. (This is handled by the *FunDen* containing the *Env* from the point of declaration.)

```
program
  begin
  int a;
  function f() int
    a := 2 result(7) end
  ...
  a := 1
    begin
    int a;
    a := 5;
    a := f()
    end
  /* what is the value of a */
  end
end
```

Fig. 5. Function static scoping

$$\begin{array}{lll} \mathit{FunDen} :: \mathit{parms} & : & \mathit{Id}^* \\ \quad\quad\quad\;\; \mathit{body} & : & \mathit{Stmt}^* \\ \quad\quad\quad\;\; \mathit{result} & : & \mathit{Expr} \\ \quad\quad\quad\;\; \mathit{context} & : & \mathit{Env} \end{array}$$

These are easy to build by selecting some components of the declaration of a *Fun*. (The reason for storing the declaring *Env* is explained below.)

$$b\text{-}Fun\text{-}Den : Fun \times Env \to FunDen$$

$$b\text{-}Fun\text{-}Den(mk\text{-}Fun(returns, params, paramtps, body, result), env) \triangleq mk\text{-}FunDen(params, body, result, env)$$

Putting this all together gives:

$$\frac{\begin{array}{l}(varenv, \sigma') = newlocs(vars, \sigma) \\ funenv = \\ \quad \{f \mapsto b\text{-}FunDen(funs(f), env \dagger varenv) \mid f \in \mathbf{dom}\, funs\} \\ env' = env \dagger varenv \dagger funenv \\ (body, env', \sigma') \xrightarrow{sl} \sigma''\end{array}}{(mk\text{-}Block(vars, funs, body), env, \sigma) \xrightarrow{s} (\mathbf{dom}\, \sigma) \lhd \sigma''}$$

The key point in the semantic rule for *Call* statements is the creation of *arglocs* which holds the locations of the arguments:

$$\frac{\begin{array}{l}(lhs, env, \sigma) \xrightarrow{lhv} l \\ mk\text{-}FunDen(parms, body, result, context) = env(f) \\ \mathbf{len}\, arglocs = \mathbf{len}\, args \\ \forall i \in \mathbf{inds}\, arglocs \cdot (args(i), env, \sigma) \xrightarrow{lhv} arglocs(i) \\ parm\text{-}env = \{parms(i) \mapsto arglocs(i) \mid i \in \mathbf{inds}\, parms\} \\ (body, (context \dagger parm\text{-}env), \sigma) \xrightarrow{sl} \sigma' \\ (result, (context \dagger parm\text{-}env), \sigma') \xrightarrow{e} res\end{array}}{(mk\text{-}Call(lhs, f, args), env, \sigma) \xrightarrow{s} (\sigma' \dagger \{l \mapsto res\})}$$

At this point a complete definition of the language so far can be presented — see Appendix B.

If one were to allow side effects in functions, the type of the semantic relation for *Expr*essions would have to reflect this decision.

Both *Env* and "surrogates" like *ScalarLoc* are general modelling tools.

4.4 Modelling arrays

It is interesting to pause for a moment to consider two possible models for adding arrays to the language in Appendix B. Looking firstly at one dimensional arrays (vectors), one might be tempted to use:

$$Env = Id \xrightarrow{m} Loc$$

$$\Sigma = Loc \xrightarrow{m} (ScalarValue \mid ArrayValue)$$

$$ArrayVal = \mathbb{N} \xrightarrow{m} ScalarValue$$

This would make passing of array elements by-location very messy. A far better model is:

$$Env = Id \xrightarrow{m} Den$$

$$Den = ScalarLoc \mid ArrayLoc \mid FunDen$$

$$ArrayLoc = \mathbb{N} \xrightarrow{m} ScalarLoc$$

$$\Sigma = ScalarLoc \xrightarrow{m} ScalarValue$$

Thinking about alternatives for multi-dimensional arrays, symmetry points us at:

$$ArrayLoc = (\mathbb{N}^*) \xrightarrow{m} ScalarLoc$$

Rather than

$$ArrayLoc = \mathbb{N} \xrightarrow{m} (ScalarLoc \mid ArrayLoc)$$

It is possible to add a data type invariant:

$$ArrayLoc = (\mathbb{N}^*) \xrightarrow{m} ScalarLoc$$
$$\textbf{inv}\ (m) \triangleq \exists ubl \in (\mathbb{N}^*) \cdot \textbf{dom}\, m = sscs(ubl)$$

The semantics of Appendix B requires minimal changes.[17] They are sketched here, starting with the abstract syntax:

$$\begin{array}{lll} Assign :: & lhs & : VarRef \\ & rhs & : Expr \end{array}$$

$$VarRef = ScalarRef \mid ArrayElRef$$

$$ScalarRef :: name : Id$$

$$\begin{array}{lll} ArrayElRef :: & array & : Id \\ & sscs & : Expr^* \end{array}$$

$$\begin{array}{lll} Call :: & lhs & : VarRef \\ & fun & : Id \\ & args & : VarRef^* \end{array}$$

$$Expr = ArithExpr \mid RelExpr \mid VarRef \mid ScalarValue$$

The semantics requires a revision to the computation of left hand values.[18]

$$\xrightarrow{lhv} : \mathcal{P}((VarRef \times Env \times \Sigma) \times ScalarLoc)$$

$$\frac{}{(mk\text{-}ScalarRef(id), env, \sigma) \xrightarrow{lhv} env(id)}$$

[17] In fact, the most extensive change is coding up a way to select distinct *ScalarLoc*s for each array element.

[18] The issue of dynamic errors is here impossible to avoid — see Section 5.5.

$$\frac{\begin{array}{l}\textbf{len}\ sscvl = \textbf{len}\ sscs \\ \forall i \in sscs \cdot (sscs(i), env, \sigma) \xrightarrow{e} sscvl(i) \\ sscvl \in \textbf{dom}\,(env(id))\end{array}}{(mk\text{-}ArrayElRef(id, sscs), env, \sigma) \xrightarrow{lhv} (env(id))(sscvl)}$$

$$\frac{\begin{array}{l}e \in VarRef \\ (e, env, \sigma) \xrightarrow{lhv} l\end{array}}{(e, env, \sigma) \xrightarrow{e} \sigma(l)}$$

One interesting issue that can be considered at this point is array "slicing" (i.e. the ability to define locations for (arbitrary) sub-parts of arrays).

4.5 Other parameter passing mechanisms

Many other parameter passing mechanisms have been devised. Since what happens in object-oriented languages is fairly simple, a full account is not presented here; but a few brief notes might encourage the reader to experiment.

The simplest and most obvious mechanism is probably parameter passing by value. This is modelled as though one were creating a block with initialization via the argument of the newly created locations. Here, of course, arguments in calls can be general expressions.

As pointed out at the beginning of Section 4.3, there are clear dangers in parameter passing by-location. These are tolerated because the other side of the engineering balance is that certain programs are significantly more efficient if addresses are passed without creating new locations and copying values. The other advantage of being able to affect the values in the calling code can, however, be achieved without introducing all of the disadvantages of aliasing. The parameter passing mechanism known as by-value/return copies the values at call time but also copies the values back at the end of the called code. Not surprisingly, the formal model is a hybrid of call-by-name and call-by-value.

These three methods by no means exhaust the possibilities: for example, Algol 60 [BBG+63] offered a general "call-by-name" mechanism which essentially treated the argument like a function (which therefore required evaluation in an appropriate environment). It is important to note that this is not the same as parameter passing "by text" where the raw text is passed and evaluated in the called context.

It is not difficult to see how functions can be passed as arguments. It should be noted that returning functions as results is more delicate because the context in which they were declared might no longer exist after the return. For similar reasons, this author has never accepted arguments about "making functions first class objects" (cf. [vWSM+76]) and adding function variables to a language (they also add confusions in reasoning about programs which are akin to those with goto statements).

5 Modelling more language features

There are many aspects of programming languages that could be explored at this point: here, only to those that relate to our objective of understanding object-oriented languages in Section 6 are considered.

5.1 Records

Algol-W [WH66] provided support for a "record" construct (in other languages sometimes called "structures"). Records are like arrays in that they collect together several values but in the case of records the "fields" need not be of the same type. Reference to individual fields is by name (rather than numerical index) and it is straightforward to offer a "strong typing" approach so that correct reference is a compile time question but this does require a notion of declaring record types if the matching is done by name rather than by shape. (Some languages –including Pascal– somewhat complicated this issue by offering "variant records".)

Having studied arrays in Section 4.4, it is fairly clear how to model structures. Their type checking is straightforward. The semantic model revolves around

$$\mathit{RecordLoc} = \mathit{Id} \xrightarrow{m} \mathit{Loc}$$

Unlike *ArrayLoc*, there is no virtue in providing the symmetrical access to any nested field and one has:

$$\mathit{Loc} = \mathit{ScalrLoc} \mid \mathit{RecordLoc}$$

An interesting scoping extension is the Pascal **with** construct that can be used to open up the naming of the fields of a record.

Extensions to cope with arrays of structures or record elements which are arrays are straightforward.

5.2 Heap storage

The block structured languages up to this point can be implemented with a "stack discipline": that is, the most recently allocated storage is always the next to be released. Making this work for languages of the Algol family is non-trivial but Dijkstra's "display" idea showed that it was possible and there have been subsequent developments (e.g. [HJ71]).

Storage which is allocated and freed by the programmer poses many dangers but heap storage in one form or another is available in all but the most restrictive languages. The need is clear: programs such as those for B-Trees need to allocate and free storage at times that do not match the phrase structure of a program. In fact, forms of dynamic storage manipulation were simulated in arrays from FORTRAN onwards and, of course, LISP was built around pointers. The concept of records made it possible for the programmer to describe structures that contained fields which were pointers (to record types). Pascal offered

a **new** statement which was implemented by keeping a pool of free storage and allocating on request.

Once one has a model of records as in the preceding section, it is not difficult to build a model for heap storage: the set of *ScVal*ues has to include *Pointer*s. In fact, this is an area where the abstract model is perhaps too easy to construct in the sense that the ease hides considerable implementation detail. One can however discuss issues like "garbage collection" and "dangling pointers" in terms of a carefully constructed model.

5.3 Abstract data types

The whole subject of "abstract data types" deserves a history in its own right. For the key contribution made by the "CLU" language, see [Lis96]. Here, it is sufficient to observe that it was realized that programmers needed the ability to change the implementation of a collection of functions and/or procedures by redefining the underlying data structures *without* changing their syntactic or semantic interface. It was thus essential to have language constructs which fixed interfaces but hid internal details.

5.4 More on concurrency

There are many concurrency extensions which can be made to the language developed to this point.[19] Interesting exercises include the addition of a "parallel For statement". As in Section 3.1, one quickly becomes aware of the dangers of interference between concurrent threads of execution. It is argued in Section 6 that one of the advantages of object-oriented languages is that they offer a way to marshal concurrency.

For now, the key questions to be noted are:

- How are threads created?
- How does one synchronize activity between threads?
- What is the level of granularity? (or atomicity)

Each of these questions can be studied and described using operational semantics and the question of atomicity in particular is returned to in Section 7.

A study of the different views of locking taken by the programming language and database communities (cf. [JLRW05]) can also be based on operational semantic descriptions.

5.5 Handling run-time errors

Context conditions are used to rule out programs which can be seen to be incorrect statically: the classic example of such errors is mismatch between type declarations of variables and their use. Many errors can, however, only be detected when a program is executed — at least in general. Access to uninitialized

[19] In fact, it is instructive to model even primitive concepts like "semaphores".

variables (especially those declared to contain pointers) is one class of such errors: obvious cases might be spotted statically, but in general one can only know about control flow issues with the actual values in a state.

A better example –and the one used in this section– might be indexing outside the bounds of an array. As those who have suffered from "stack overflow" attacks know to their cost, this can be an effective way to corrupt a program. It is not difficult to mark the detection of such errors in operational semantic descriptions; the bigger question is what action should be described when run-time errors are detected. In Section 4.4, the rule

$$\frac{\begin{array}{l}\textbf{len}\ sscvl = \textbf{len}\ sscs \\ \forall i \in sscs \cdot (sscs(i), env, \sigma) \xrightarrow{e} sscvl(i) \\ sscvl \in \textbf{dom}\,(env(id))\end{array}}{(\mathit{mk\text{-}ArrayElRef}(id, sscs), env, \sigma) \xrightarrow{lhv} (env(id))(sscvl)}$$

clearly shows in its last hypothesis that access is only defined for valid subscript lists. In effect, there is no rule for invalid subscripts so the computation "stalls". For emphasis, one could add a rule that states an error has occurred but there is a meta-issue about what a language standard has to say about whether such errors must be detected or whether an implementation is free to deliver any result from the time of the error onwards. This latter course might appear to be a denigration of responsibility but one must accept that checking for arbitrary rune time errors can be expensive. This is one reason for seeking as strong a type discipline as possible.

More promising are the languages which define what should be done on encountering errors. A language might, for example, require that an out-of-bounds exception be raised. Essentially, the idea is to make semantic functions deliver either a normal or abnormal result. This idea originated in [HJ70] and was fully worked out in [ACJ72]; Nipkow [KNvO$^+$02] uses a more economical way of defining the union of the possibilities.

6 Understanding objects

All of the modelling tools to understand –and record our understanding of– an interesting language are to hand. Furthermore, it is possible to look at how object-oriented languages resolve some of the key engineering tensions relating to the design of programming languages. The strands of our story coalesce here.

The language introduced in this section is referred to as "COOL". It is not intended to be a complete OOL (extensions are sketched in Section 6.6). The reader is referred to [KNvO$^+$02] for a description of Java.

Section 5.2 discusses the need to create storage dynamically (on a heap); the necessity to dispose of unwanted items; and resulting issues of garbage collection. Objects collect together data fields for their "instance variables" in a way that gives the power or records. Objects can be dynamically created (and garbage collected).

Locality of reference to the fields of an object by the methods of that class offers a way to resolve the (abstract data type — cf. Section 5.3) issues in a way which lets the implementation of an object be changed without changing its interface.

In one sense, the pure object view that everything (even a constant) is an object sweeps away the distinctions in parameter passing: everything is passed by location — but some objects are immutable.

Most importantly for our concern about concurrency, object-oriented languages provide a natural way to marshal threads. The view is taken here that each object should comprise a thread of control. Because instance variables can only be referred to by the methods of that class[20], there is a natural insulation against interference. Sharing can be established by the passing of object references but this is under clear programmer control. In COOL, the restrictive view is taken that only one method can be active in an object and this eliminates local race conditions. This combination of decisions means that the programmer is also in control of the level of atomicity (of interference).

(Space does not permit a discussion of (the important) issues of why objects work well in design and point the reader at excellent books such as [DW99] for such material.)

6.1 Introducing COOL

It is easiest to get into the spirit of COOL by considering a programming example. The class *Sort* in Figure 6 provides a (sequential) facility for maintaining a sequence of integers in ascending order. (One could add a method that returns –and deletes– the first item but the *insert* and *test* let us show the interesting features.)

A class is a template for object structure and behaviour: it lists the instance variables with their corresponding types and defines the parameters for each method, its result type and its implementation. An instance variable can have one of three types: integer, Boolean or (typed) reference (or "handle"). A reference value is the "handle" of another object; the special value nil is used to indicate when no reference is being held.[21]

Objects of a class (objects corresponding to the class description) can be generated by executing a new statement that creates a new object with which a unique reference is associated, and returns this reference as a result. As implied above, all objects of a class share the same structure and behaviour, however,

[20] Avoiding the use of Java's **public** fields.

[21] To justify some interesting equivalences (see below) any variable declared to be a reference is identified as either shared or private (unique). The latter is written as a keyword (unique); the default is shared. A variable marked as unique can only be assigned a handle of a newly created object and it is prohibited to duplicate its contents: unique variables cannot appear on the right hand side of an assignment statement, be passed as arguments to a method or be returned as a result. These restrictions ensure that the object reference being held is unknown to any other object.

```
Sort class
vars v: ℕ ← 0; l: unique ref(Sort) ← nil
insert(x: ℕ) method
  begin
    if is-nil(l) then (v ← x; l ← new Sort)
    elif v ≤ x then l.insert(x)
    else (l.insert(v); v ← x)
    fi
    ;
    return
  end
test(x: ℕ) method : 𝔹
  if is-nil(l) ∨ x < v then return false
  elif x = v then return true
  else return l.test(x)
  fi
```

Fig. 6. Example Program *Sort* – sequential

each possesses its own copy of the instance variables; it is on these copies that the methods operate.

An object can attempt to invoke[22] a method of any object to which it holds a handle. The concrete syntax for method invocation is $\alpha.m(\tilde{x})$, where α is the identity of the object, m is the method name and $\tilde{x}$ is the list of parameters. When an object accepts a method invocation the client is held in a *rendezvous*. The *rendezvous* is completed when a value is returned; in the simplest case this is by a return statement.[23]

In addition to the statements described above, COOL provides a normal repertoire of simple statements.

It follows from the above that an object can be in one of three states: *quiescent* (idle), *waiting* (held in *rendezvous*) or *active* (executing a method body). Methods can only be invoked in an object which is in the quiescent state; therefore –in COOL– at most one method can be active at any one time in a given object.

These comments should help to clarify most aspects of the sequential version of *Sort*.[24]

The implementation of both these methods is sequential: at most one object is active at any one time. Concurrency can be introduced into this example by

[22] The terms "method invocation" and "method call" are used interchangeably.

[23] The delegate statement allows an object to transfer the responsibility for answering a method call to another object, without itself waiting for the result – see below.

[24] The return statement in Figure 6 has a method call in the place of an expression, which strictly does not conform to the syntax of COOL. One simple remedy would be to assign the result of this call to a temporary variable and return the value of that variable. Since this is straightforward, and adds nothing to the language, it is preferred here to rely on the reader's comprehension.

applying two equivalences. The *insert* method given in Figure 6 is sequential because its client is held in a *rendezvous* until the effect of the insert has passed down the list structure to the appropriate point and the return statements have been executed in every object on the way back up the list. If the return statement of *insert* is commuted to the beginning of the method as in Figure 7, it becomes a release in which the client is able to continue its computation concurrently with the activity of the insertion. Furthermore, as the insertion progresses down the list, objects 'up stream' of the operation are free to accept further method calls. One can thus imagine a whole series of *insert* operations trickling down the list structure concurrently.

```
Sort class
vars v: ℕ ← 0; l: unique ref(Sort) ← nil
insert(x: ℕ) method
  begin
    release;
    if is-nil(l) then (v ← x; l ← new Sort)
    elif v ≤ x then l.insert(x)
    else (l.insert(v); v ← x)
    fi
  end
test(x: ℕ) method : 𝔹
  if is-nil(l) ∨ x < v then return false
  elif x = v then return true
  else delegate l.test(x)
  fi
```

Fig. 7. The concurrent implementation of *Sort*

It is not possible to apply the return commutation equivalence to the *test* method because the client must be held until a result can be returned. It is, however, possible to avoid the entire list being 'locked' throughout the duration of a *test* method. In the sequential implementation, invocations of the *test* method in successive instances of *Sort* run down the list structure until either the value being sought is found or the end of the list is reached; at this point the Boolean result is passed back up the list; when the result reaches the object at the head of the list it is passed to the client. If instead each object has the option to *delegate* the responsibility of answering the client, it is possible for the first object in the list to accept further method calls. Again one can imagine a sequence of *test* method calls progressing down the list concurrently.[25] The

[25] Notice however that although the linear structure of the list prevents overtaking, it is possible for invocations to be answered in a different order from that in which they were accepted. For example –in the situation – if two invocations are accepted in the order *test*(4) followed by *test*(1), it is possible for the result of the second call to be

transformed implementation of *test* is given in Figure 7. A more telling example with trees is given in [Jon96].

Because release statements do not have to come at the end of methods and the use of delegate statements, COOL is already an object-based language which permits concurrency. Other ways in which concurrency can be added are mentioned in Section 6.6.

Sections 6.3–6.5 outline the parts of a formal description. Appendix C fills in the details and collects the description in the same order as Appendix B. But first the overall modelling strategy is discussed.

6.2 Modelling strategy

At one level, objects are just pieces of storage (not unlike records) that can be dynamically created by executing a **new** statement. One can thus anticipate that there will have to be –in our semantic model– a mapping from some *Reference* to the local values. But this does not completely bring out the nature of objects. I owe to the late Ole-Johan Dahl the observation that objects are best understood as "blocks" that can be instantiated multiple times (in contrast to the Algol model where their existence is governed by when control flows through their text). A class defines instance variables and methods just like the local variables and functions/procedures of a block. The instance variables are known only to those methods. One oddity is that the scope of the method names is external to the class (but this is precisely so that they become the access points to actions on the instances (objects) of the class). As mentioned already, the real difference from an Algol block is that instances of classes can be created at will.[26]

This understanding gives us our basic modelling strategy: the run-time information about objects will be stored in a mapping (*ObjMap* in Section 6.5). The *ObjInfo*s stored in this mapping have –as might be expected– a field (*state*) in which the values of the instance variables for the object in question are stored. Because the threads are running interleaved, *ObjInfo* is also keeping track of what text remains to be executed in each active thread. In essence, *ObjInfo* replaces the notion of a "configuration" discussed in Section 3.1. (Section 6.5 discusses the other fields in *ObjInfo*.) Notice that there is no notion of global state here although one might need one if input/output to files were considered.

Section 4.2 points out the need to have access to the text of any program unit which can be used from many places. In COOL, this applies both to the shape (in terms of its instance variables) of a class and the text of the methods of a class for when they are invoked. In Section 6.5 the program text is always available in *Classes*. This leads us to an overall semantic relation:

$$\xrightarrow{s} : \mathcal{P}((Classes \times ObjMap) \times ObjMap)$$

returned before the first has completed. Although this would constitute a modified behaviour when viewed from an all-seeing spectator, no COOL program can detect the difference.

[26] Postponing a discussion of nesting until Section 6.6.

Returning to the question of relating the parameter passing in COOL to what has gone before, it is clear that object references are passed by-reference. This is precisely what lets a programmer set up sharing patterns (which can in turn introduce race conditions).

6.3 Abstract syntax

The aim here is to build up the definition of "COOL" in Appendix C where the description is organized by language construct. Here, the whole abstract syntax is presented at once.

A *Program* contains a collection of named *ClassBlock*s; it is assumed that execution begins with a single (parameterless) method call.

$$\begin{array}{lll} \mathit{Program} :: & \mathit{cm} & : \mathit{Classes} \\ & \mathit{start\text{-}class} & : \mathit{Id} \\ & \mathit{start\text{-}meth} & : \mathit{Id} \end{array}$$

$$\mathit{Classes} = \mathit{Id} \xrightarrow{m} \mathit{ClassBlock}$$

Notice that there is (unlike in the *Block* language) no body in a *ClassBlock*; having one would provide another natural concurrency extension – see Section 6.6.

$$\begin{array}{lll} \mathit{ClassBlock} :: & \mathit{vars} & : \mathit{Id} \xrightarrow{m} \mathit{Type} \\ & \mathit{meths} & : \mathit{Id} \xrightarrow{m} \mathit{Meth} \end{array}$$

$$\mathit{Type} = \mathit{Id} \mid \mathit{ScalarType}$$

$$\mathit{ScalarType} = \textsc{IntTp} \mid \textsc{BoolTp}$$

Methods are very like function definitions.

$$\begin{array}{lll} \mathit{Meth} :: & \mathit{returns} & : \mathit{Type} \\ & \mathit{params} & : \mathit{Id}^* \\ & \mathit{paramtps} & : \mathit{Id} \xrightarrow{m} \mathit{Type} \\ & \mathit{body} & : \mathit{Stmt}^* \end{array}$$

All of the points to be illustrated can be made with the following list of statements.

$$\mathit{Stmt} = \mathit{Assign} \mid \mathit{If} \mid \mathit{New} \mid \mathit{MethCall} \mid \mathit{Return} \mid \mathit{Release} \mid \mathit{Delegate}$$

$$\begin{array}{lll} \mathit{Assign} :: & \mathit{lhs} & : \mathit{Id} \\ & \mathit{rhs} & : \mathit{Expr} \end{array}$$

$$\begin{array}{lll} \mathit{If} :: & \mathit{test} & : \mathit{Expr} \\ & \mathit{th} & : \mathit{Stmt}^* \\ & \mathit{el} & : \mathit{Stmt}^* \end{array}$$

$$\begin{array}{lll} \mathit{New} :: & \mathit{targ} & : \mathit{Id} \\ & \mathit{class} & : \mathit{Id} \end{array}$$

$$\begin{array}{lll} \mathit{MethCall} :: & \mathit{lhs} & : \mathit{Id} \\ & \mathit{obj} & : \mathit{Id} \\ & \mathit{meth} & : \mathit{Id} \\ & \mathit{args} & : \mathit{Id}^* \end{array}$$

$$\mathit{Return} :: \mathit{val} : (\mathit{Expr} \mid \textsc{self})$$

$$\mathit{Release} :: \mathit{val} : (\mathit{Expr} \mid \textsc{self})$$

$$\begin{array}{lll} \mathit{Delegate} :: & \mathit{obj} & : \mathit{Id} \\ & \mathit{meth} & : \mathit{Id} \\ & \mathit{args} & : \mathit{Id}^* \end{array}$$

The syntax of expression is presented only in the appendix.

6.4 Context conditions

The context conditions are straightforward (and are given in the appendix). Well-formed COOL programs are statically checked to have only syntactically correct method calls.

6.5 Semantics

Dynamic information about Objects is stored in:

$$\mathit{ObjMap} = \mathit{Reference} \xrightarrow{m} \mathit{ObjInfo}$$

$$\begin{array}{lll} \mathit{ObjInfo} :: & \mathit{class} & : \mathit{Id} \\ & \mathit{state} & : \mathit{VarState} \\ & \mathit{status} & : \mathit{Status} \\ & \mathit{remaining} & : \mathit{Stmt}^* \\ & \mathit{client} & : [\mathit{Reference}] \end{array}$$

For any object, the *class* field is the name of the class to which it belongs. This can be used on invocation of a method to locate its *body*.

The *state* field for an object contains the values of its instance variables.

$$\mathit{VarState} = \mathit{Id} \xrightarrow{m} \mathit{Val}$$

$$\mathit{Val} = \mathit{Reference} \mid \mathbb{Z} \mid \mathbb{B}$$

There is some redundancy in the way the *status* of an object is recorded but it is in all cases essential to be able to distinguish between an object which presents an active thread from one which is idle (methods can only be invoked in idle threads). Furthermore, when an object is waiting for a value to returned, the *Wait* field records where that value will be stored.

$$\mathit{Status} = \textsc{Active} \mid \textsc{Idle} \mid \mathit{Wait}$$

$$\mathit{Wait} :: \mathit{lhs} : \mathit{Id}$$

For an ACTIVE thread (object), the text remaining to be executed in its method is recorded in the *remaining* field and the identity of the client who is awaiting a returned value from any object is recorded in that object's *client* field.

The types of the required relations are

$$\xrightarrow{s} : \mathcal{P}((\mathit{Classes} \times \mathit{ObjMap}) \times \mathit{ObjMap})$$

and

$$\xrightarrow{e} : \mathcal{P}((\mathit{Expr} \times \mathit{VarState}) \times \mathit{Val})$$

Each rule in the semantics for a statement needs to locate an active thread awaiting execution of a statement of that type: thus the general shape of all of the $\xrightarrow{s}$ rules is:

$$\frac{\begin{array}{l} O(a) = \mathit{mk\text{-}ObjInfo}(c, \sigma, \textsc{Active}, [\mathit{mk\text{-}Stmt\text{-}Type}(\ldots)] \curvearrowright \mathit{rl}, \mathit{co}) \\ \vdots \end{array}}{(C, O) \xrightarrow{s} O \dagger \cdots}$$

For **new**, all that is needed is a thread ready to execute $\mathit{mk\text{-}New}(\mathit{targ}, c')$. The execution of that statement is reflected by its removal and a new object (with a brand new *Reference* — and in **Idle** status) is created with appropriate initial values for the instance variables:

$$\frac{\begin{array}{l} O(a) = \mathit{mk\text{-}ObjInfo}(c, \sigma, \textsc{Active}, [\mathit{mk\text{-}New}(\mathit{targ}, c')] \curvearrowright \mathit{rl}, \mathit{co}) \\ b \in (\mathit{Reference} - \mathbf{dom}\ O) \\ \mathit{aobj}' = \mathit{mk\text{-}ObjInfo}(c, \sigma \dagger \{\mathit{targ} \mapsto b\}, \textsc{Active}, \mathit{rl}, \mathit{co}) \\ \sigma_b = \text{initial values} \\ \mathit{nobj} = \mathit{mk\text{-}ObjInfo}(c', \sigma_b, \textsc{Idle}, [\,], \mathbf{nil}) \end{array}}{(C, O) \xrightarrow{s} O \dagger \{a \mapsto \mathit{aobj}', b \mapsto \mathit{nobj}\}}$$

In order for thread a to invoke a method in another thread, the latter must be quiescent (its *status* field must be IDLE). The statements to be executed for the called method are found in C and parameters are passed in an obvious way.

$$\frac{\begin{array}{l} O(a) = \\ \quad \mathit{mk\text{-}ObjInfo}(c, \sigma, \textsc{Active}, [\mathit{mk\text{-}MethCall}(\mathit{lhs}, \mathit{obj}, \mathit{meth}, \mathit{args})] \curvearrowright \mathit{rl}, \mathit{co}) \\ O(\sigma(\mathit{obj})) = \mathit{mk\text{-}ObjInfo}(c', \sigma', \textsc{Idle}, [\,], \textsc{nil}) \\ C(c') = \mathit{mk\text{-}ClassBlock}(\mathit{vars}, \mathit{meths}) \\ \mathit{aobj}' = \mathit{mk\text{-}ObjInfo}(c, \sigma, \mathit{mk\text{-}Wait}(\mathit{lhs}), \mathit{rl}, \mathit{co}) \\ \sigma'' = \sigma' \dagger \{(\mathit{meths}(\mathit{meth}).\mathit{params})(i) \mapsto \sigma(\mathit{args}(i)) \mid i \in \mathbf{inds}\ \mathit{args}\} \\ \mathit{sobj} = \mathit{mk\text{-}ObjInfo}(c', \sigma'', \textsc{Active}, \mathit{meths}(\mathit{meth}).\mathit{body}, a) \end{array}}{(C, O) \xrightarrow{s} O \dagger \{a \mapsto \mathit{aobj}', \sigma(\mathit{obj}) \mapsto \mathit{sobj}\}}$$

When a method finishes (remember the *Release* can have occured earlier) it reverts to the quiescent status.

$$\frac{\begin{array}{l} O(a) = \mathit{mk\text{-}ObjInfo}(c, \sigma, \textsc{Active}, [\,], \mathit{co}) \\ \mathit{aobj}' = \mathit{mk\text{-}ObjInfo}(c, \sigma, \textsc{Idle}, [\,], \mathbf{nil}) \end{array}}{(C, O) \xrightarrow{s} O \dagger \{a \mapsto \mathit{aobj}'\}}$$

Returning values makes the server object IDLE. The thread to which the value is to be returned is found from the *client* field of the completing method. The place to which the returned value should be assigned is found in $mk\text{-}Wait(lhs)$ which was placed there at the time of the method invocation. The server object a becomes idle.

$$\frac{\begin{array}{l} O(a) = mk\text{-}ObjInfo(c, \sigma, \textsc{Active}, [mk\text{-}Return(e)] \curvearrowright rl, co) \\ e \in Expr \\ (e, \sigma) \xrightarrow{e} v \\ O(co) = mk\text{-}ObjInfo(c', \sigma', mk\text{-}Wait(lhs), sl, co') \\ aobj' = mk\text{-}ObjInfo(c, \sigma, \textsc{Idle}, [\,], \mathbf{nil}) \\ cobj' = mk\text{-}ObjInfo(c', \sigma' \dagger \{lhs \mapsto v\}, \textsc{Active}, sl, co') \end{array}}{(C, O) \xrightarrow{s} O \dagger \{a \mapsto aobj', co \mapsto cobj'\}}$$

If SELF is being returned, replace the second line with $v = a$.

Releasing a *rendez vous* is similar except that the a thread remains active:

$$\frac{\begin{array}{l} O(a) = mk\text{-}ObjInfo(c, \sigma, \textsc{Active}, [mk\text{-}Release(e)] \curvearrowright rl, co) \\ (e, \sigma) \xrightarrow{e} v \\ O(co) = mk\text{-}ObjInfo(c', \sigma', mk\text{-}Wait(lhs), sl, co') \\ aobj' = mk\text{-}ObjInfo(c, \sigma, \textsc{Active}, rl, \mathbf{nil}) \\ cobj' = mk\text{-}ObjInfo(c', \sigma' \dagger \{lhs \mapsto v\}, \textsc{Active}, sl, co') \end{array}}{(C, O) \xrightarrow{s} O \dagger \{a \mapsto aobj', co \mapsto cobj'\}}$$

If SELF is being returned, one again replaces the second line with $v = a$.

The delegate statement is interesting because it works like a combination of method invocation and a release statement:

$$\frac{\begin{array}{l} O(a) = \\ \quad mk\text{-}ObjInfo(c, \sigma, \textsc{Active}, [mk\text{-}Delegate(obj, meth, args)] \curvearrowright rl, co) \\ O(\sigma(obj)) = mk\text{-}ObjInfo(c', \sigma', \textsc{Idle}, [\,], \textsc{nil}) \\ C(c') = mk\text{-}ClassBlock(vars, meths) \\ aobj' = mk\text{-}ObjInfo(c, \sigma, \textsc{Active}, rl, \mathbf{nil}) \\ \sigma'' = \sigma' \dagger \{(meths(meth).params)(i) \mapsto \sigma(args(i)) \mid i \in \mathbf{inds}\ args\} \\ sobj = mk\text{-}ObjInfo(c', \sigma'', \textsc{Active}, meths(meth).body, co) \end{array}}{(C, O) \xrightarrow{s} O \dagger \{a \mapsto aobj', \sigma(obj) \mapsto sobj\}}$$

Rules For *Assign* etc. should be obvious (and are in the appendix).

6.6 Developments from here

There are an enormous number of developments that one can make from the definition in Appendix C. It is straightforward to add new data types (such as strings) or new statement types. A(n OO) purist would point out that COOL is not fully OO (in the sense of Smalltalk) since it uses integer and Boolean values. (There is a subtlety in removing Booleans from the language itself: in order to give a semantics to any statement requiring a truth-valued result, one ends up needing some form of "closure".) Adding arrays is also interesting in as much as it highlights the lack of a location concept for the instance variables (see also below).

More subtly, it is not difficult to partially lift the restriction on "one method active per object" and provide some form of "call back" without introducing race conditions.

Much more interesting is to add to COOL new ways of creating concurrency. In Appendix C, concurrency is achieved by use of **release** (and **delegate**); as an alternative (or addition), a parallel *For* statement could be added and one could, for example, program a parallel version of the "Sieve of Eratosthenes" [Jon96].

A useful extension would be to add "creation code" to each class by including code in the body of the class.

$$\begin{array}{lll} \mathit{ClassBlock} :: & \mathit{vars} & : \mathit{Id} \xrightarrow{m} \mathit{Type} \\ & \mathit{meths} & : \mathit{Id} \xrightarrow{m} \mathit{Meth} \\ & \mathit{constructor} & : [\mathit{CMeth}] \end{array}$$

$$\begin{array}{lll} \mathit{CMeth} :: & \mathit{params} & : \mathit{Id}^* \\ & \mathit{paramtps} & : \mathit{Id} \xrightarrow{m} \mathit{Type} \\ & \mathit{body} & : \mathit{Stmt}^* \end{array}$$

One could then have the **new** statement pass arguments to the creation code

$$\begin{array}{lll} \mathit{New} :: & \mathit{targ} & : \mathit{Id} \\ & \mathit{class} & : \mathit{Id} \\ & \mathit{args} & : \mathit{Id}^* \end{array}$$

Having a creation body in a Class makes it an autonomous locus of control; one could then fire off many processes at once (cf. [Ame89]). Thus the semantic rule for the **new** statement might become:

$$\frac{\begin{array}{l} O(a) = \mathit{mk\text{-}ObjInfo}(c, \sigma, \textsc{Active}, [\mathit{mk\text{-}New}(\mathit{targ}, c', \mathit{args})] \curvearrowright \mathit{rl}, \mathit{co}) \\ b \in (\mathit{Reference} - \mathbf{dom}\ O) \\ \mathit{aobj}' = \mathit{mk\text{-}ObjInfo}(c, \sigma \dagger \{\mathit{targ} \mapsto b\}, \textsc{Active}, \mathit{rl}, \mathit{co}) \\ \mathit{mk\text{-}ClassBlock}(\mathit{vars}, \mathit{meths}, \mathit{cons}) = C(c') \\ \mathit{mk\text{-}CMeth}(\mathit{parms}, \mathit{parmts}, \mathit{cbody}) = \mathit{cons} \\ \sigma_b = \{\mathit{parms}(i) \mapsto \sigma(\mathit{args}(i)) \mid i \in \mathbf{inds}\ \mathit{parms}\} \\ \mathit{nobj} = \mathit{mk\text{-}ObjInfo}(c', \sigma_b, \textsc{Active}, \mathit{cbody}, \mathbf{nil}) \end{array}}{(C, O) \xrightarrow{s} O \dagger \{a \mapsto \mathit{aobj}', b \mapsto \mathit{nobj}\}}$$

COOL's "one method per object" rule means that the constructor will block other method calls until construction is finished.

Another interesting extension would be to allow some access to the instance variables of an object (as in Java's **public**). It would be safe to do this for Idle objects; permitting such access within an Active object would introduce the danger of race conditions.

One could go further and add some form of process algebraic notation for controlling permissible orders of method activation.[27]

Object-oriented purists would also object that COOL offers no form of inheritance. This is –at least in part– intentional because of the confusions surrounding

[27] One could derive intuition from similar ideas in the meta-language as in [FL98], [But00] or [WC02]. My own preference would be to use pi-calculus and have the ν operator create objects (I have given talks on this but not yet written anything).

the idea. One useful handle on the semantics of inheritance is to go back to the observation that classes are like blocks that can be instantiated at will. A nested block offers all of the facilities (variables and functions) of its surrounding block except where overridden. If one thinks of inheritance as creating instances of an inner block, one begins to see what the semantic implications might be (including some doubt about "multiple inheritance").

7 Conclusions

It is hoped that the reader now sees the extent to which semantic models can elucidate and record the understanding of the features of programming languages. There are descriptions of many real languages (full citations are omitted here for space reasons)

- (operational and denotational) of ALGOL 60
- (denotational) of Pascal
- SOS for ML
- (denotational) of PL/I ECMA/ANSI standard
- (denotational) of Ada
- Modula-2 standard
- Java description [KNvO$^+$02]

In fact, the obvious duplication involved in writing descriptions where there is a considerable amount of overlap in the features of the languages has led to attempts to look for ideas that make it possible to document language concepts in a way which facilitates their combination. Early steps towards this are visible in the "combinators" of the Vienna PL/I description [BBH$^+$74]; Mosses' "action semantics" [Mos92] took this much further and he has more recently been studying "Modular SOS" [Mos06].

There is no doubt that reasoning about language descriptions is extremely important. This goes beyond using a semantics to establish facts about particular programs as discussed in Section 3.2. A simple example of a general result is that a well-formed program cannot give rise to run-time type errors. An important class of proofs is the consistency of Floyd-Hoare-like proof rules with respect to a model-oriented semantics. The paper [CJ07] is an example of this (and it contains references to earlier material in this vein) which establishes the soundness of rely/guarantee rules.

The potential that originally took this author to the IBM Vienna Laboratory in 1968 was the use of formal language descriptions as a base for compiler design.[28] A description from the research at that time is [JL71] (but much of the material is only available as Technical Reports). An important historical reference is [MP66].

[28] Notice that post-facto proofs were seen even then as pointless: the pay off of formalism is in the design process.

A knowledgeable reader might question why this text has been based on operational –rather than denotational [Sto77]– semantics. It is claimed in Section 1.3 that the message is "abstraction, abstraction, abstraction" and there is a clear technical sense in which denotational semantics are more abstract than operational. The reasons are largely pedagogic (cf. [CJJ06]) but it is this author's conviction that once concurrency has to be tackled, the cost of extra mathematical apparatus does not present an adequate return.

The other omitted topic that might give rise to comment is that of process algebras such as CSP [Hoa78], CCS [Mil89] or the pi-calculus [MPW92,SW01]. The topic of their semantics and proof methods is itself fascinating.

One topic that links closely with the material above is the mapping of object-oriented languages to process algebras (cf. [Wal91,Jon93]). These semantics have been used to justify the equivalences used in transforming OO programs in Section 6 — see [Wal93,Jon94,San99] and references therein.

Acknowledgments

The author gratefully acknowledges the EPSRC support for his research under the "Splitting (software) atoms safely" grant.

References

[ACJ72] C. D. Allen, D. N. Chapman, and C. B. Jones. A formal definition of ALGOL 60. Technical Report 12.105, IBM Laboratory Hursley, August 1972.

[Ame89] Pierre America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4), 1989.

[BBG+63] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6(1):1–17, 1963.

[BBH+74] H. Bekič, D. Bjørner, W. Henhapl, C. B. Jones, and P. Lucas. A formal definition of a PL/I subset. Technical Report 25.139, IBM Laboratory Vienna, December 1974.

[BJ78] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[BJ82] D. Bjørner and C. B. Jones. *Formal Specification and Software Development*. Prentice Hall International, 1982.

[Boo54] George Boole. *An Investigation of the Laws of Thought*. Macmillan, 1854. Reprinted by Dover, 1958.

[But00] M. J. Butler. CSP2B: A practical approach to combining CSP and B. *Formal Aspects of Computing*, 12(3):182–198, 2000.

[CJ07] J. W. Coleman and C. B. Jones. Guaranteeing the soundness of rely/guarantee rules (revised). *Journal of Logic and Computation*, accepted for publication, 2007.

[CJJ06] Joey W. Coleman, Nigel P. Jefferson, and Cliff B. Jones. Comments on several years of teaching of modelling programming language concepts. Technical Report CS-TR-978, Newcastle University, 2006.

[CM92] J. Camilleri and T. Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, Computer Laboratory, University of Cambridge, August 1992.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DW99] Desmond F. D'Souza and Alan Cameron Wills. *Objects, components, and frameworks with UML: the catalysis approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Eng71] E. Engeler. *Symposium on Semantics of Algorithmic Languages*. Number 188 in Lecture Notes in Mathematics. Springer-Verlag, 1971.

[FL98] John Fitzgerald and Peter Gorm Larsen. *Modelling systems: practical tools and techniques in software development*. Cambridge University Press, 1998.

[Gor79] M. J. C. Gordon. *The Denotational Description of Programming Languages: An Introduction*. Springer-Verlag, 1979.

[Gor88] M. J. C. Gordon. *Programming Language Theory and its Implementation*. Prentice-Hall International, 1988.

[Hen90] Matthew Hennessy. *The Semantics of Programming Languages: an elementary introduction using structural operational semantics*. Wiley, 1990.

[HJ70] W. Henhapl and C. B. Jones. On the interpretation of GOTO statements in the ULD. Technical Report LN 25.3.065, IBM Laboratory, Vienna, March 1970.

[HJ71] W. Henhapl and C. B. Jones. A run-time mechanism for referencing variables. *Information Processing Letters*, 1:14–16, 1971.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, August 1978.

[JL71] C. B. Jones and P. Lucas. Proving correctness of implementation techniques. In *[Eng71]*, pages 178–211. 1971.

[JLRW05] C. B. Jones, D. Lomet, A. Romanovsky, and G. Weikum. The atomicity manifesto, 2005.

[Jon93] C. B. Jones. A pi-calculus semantics for an object-based design notation. In E. Best, editor, *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 158–172. Springer-Verlag, 1993.

[Jon94] C. B. Jones. Process algebra arguments about an object-based design notation. In *A Classical Mind: Essays in Honour of C. A. R. Hoare*, chapter 14. Prentice-Hall, 1994.

[Jon96] C. B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.

[Jon01a] C. B. Jones. On the search for tractable ways of reasoning about programs. Technical Report CS-TR-740, Newcastle University, 2001. Superceded by [Jon03a].

[Jon01b] C. B. Jones. The transition from VDL to VDM. *JUCS*, 7(8):631–640, 2001.

[Jon03a] Cliff B. Jones. The early search for tractable ways of reasonning about programs. *IEEE, Annals of the History of Computing*, 25(2):26–49, 2003.

[Jon03b] Cliff B. Jones. Operational semantics: concepts and their expression. *Information Processing Letters*, 88(1-2):27–32, 2003.

[Jon07] Cliff B. Jones. Understanding programming language concepts via operational semantics. In Chris George, Zhiming Liu, and Jim Woodcock, editors, *Lectures on Domain Modelling and the Duration Calculus*, number 4710 in LNCS, pages 177–235. Springer, 2007.

[KNvO+02] Gerwin Klein, Tobias Nipkow, David von Oheimb, Leonor Prensa Nieto, Norbert Schirmer, and Martin Strecker. Java source and bytecode formalisations in Isabelle. 2002.

[Lis96] Barbara Liskov. A history of CLU. In *History of programming languages—II*, pages 471–510. ACM Press, New York, NY, USA, 1996.

[LW69] P. Lucas and K. Walk. *On The Formal Description of PL/I*, volume 6 of *Annual Review in Automatic Programming Part 3*. Pergamon Press, 1969.

[McC66] J. McCarthy. A formal description of a subset of ALGOL. In *[Ste66]*, pages 1–12, 1966.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mos92] P. D. Mosses. *Action Semantics*. Cambridge Tracts in Theoretical Computer Science, 26. Cambridge University Press, 1992.

[Mos06] Peter D. Mosses. Teaching semantics of programming languages with Modular SOS. In *Teaching Formal Methods: Practice and Experience*, Electr. Workshops in Comput. BCS, 2006.

[MP66] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. Technical Report CS38, Computer Science Department, Stanford University, April 1966. See also pages 33–41 Proc. Symp. in Applied Mathematics, Vol.19: Mathematical Aspects of Computer Science, American Mathematical Society, 1967.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.

[NN92] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992. Available on the WWW as http://www.daimi.au.dk/bra8130/Wiley_book/wiley.html.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981.

[Plo04a] Gordon D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:3–15, July–December 2004.

[Plo04b] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, July–December 2004.

[Rey98] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998. ISBN 0-521-5914-6.

[San99] Davide Sangiorgi. Typed π-calculus at work: a correctness proof of Jones's parallelisation transformation on concurrent objects. *Theory and Practice of Object Systems*, 5(1):25–34, 1999.

[Sco00] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 2000. ISBN 1-55860-578-9.

[Ste66] T. B. Steel. *Formal Language Description Languages for Computer Programming*. North-Holland, 1966.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[SW01] Davide Sangiorgi and David Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[vWSM+76] A. van Wijngaarden, M. Sintzoff, B. J. Mailloux, C. H. Lindsey, J. E. L. Peck, L. G. L. T. Meertens, C. H. A. Koster, and R. G. Fisker. *Revised report on the Algorithmic Language ALGOL 68*. Mathematical Centre Tracts 50. Mathematisch Centrum, Amsterdam, 1976.

[Wal91] D. Walker. π-calculus semantics for object-oriented programming languages. In T. Ito and A. R. Meyer, editors, *TACS'91*, volume 526 of *Lecture Notes in Computer Science*, pages 532–547. Springer-Verlag, 1991.

[Wal93] D. Walker. Process calculus and parallel object-oriented programming languages. In *In T. Casavant (ed), Parallel Computers: Theory and Practice*. Computer Society Press, 1993.

[Wat04] David A. Watt. *Programming Language Design Concepts*. John Wiley, 2004.

[WC02] Jim Woodcock and Ana Cavalcanti. The semantics of circus. In *ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 184–203. Springer-Verlag, 2002.

[Wex81] Richard L. Wexelblat, editor. *History of Programming Languages*. Academic Press, 1981. ISBN 0-12-745040-8.

[WH66] Niklaus Wirth and C. A. R. Hoare. A contribution to the development of algol. *Commun. ACM*, 9(6):413–432, 1966.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, 1993. ISBN 0-262-23169-7.

[Zem66] H. Zemanek. Semiotics and programming languages. *Communications of the ACM*, 9:139–143, 1966.

A Base language

Notice that the formulae in this appendix separate abstract syntax, context conditions and semantics. This is not the order used in other appendices[29] but it serves at this stage to emphasize the distinctions.

A.1 Abstract syntax

$$\begin{array}{lll} \mathit{Program} :: & \mathit{vars} & : \mathit{Id} \xrightarrow{m} \mathit{ScalarType} \\ & \mathit{body} & : \mathit{Stmt}^* \end{array}$$

$$\mathit{ScalarType} = \textsc{IntTp} \mid \textsc{BoolTp}$$

$$\mathit{Stmt} = \mathit{Assign} \mid \mathit{If} \mid \mathit{While}$$

$$\begin{array}{lll} \mathit{Assign} :: & \mathit{lhs} & : \mathit{Id} \\ & \mathit{rhs} & : \mathit{Expr} \end{array}$$

[29] For reference purposes, this is normally most convenient. There remains the decision whether to present the parts of a language in a top-down (from *Program* to *Expr*) order or bottom-up: this decision is fairly arbitrary. What is really needed is an interactive support system!

$$
\begin{array}{lll}
\mathit{If} :: & \mathit{test} & : \mathit{Expr} \\
& \mathit{th} & : \mathit{Stmt}^* \\
& \mathit{el} & : \mathit{Stmt}^*
\end{array}
$$

$$
\begin{array}{lll}
\mathit{While} :: & \mathit{test} & : \mathit{Expr} \\
& \mathit{body} & : \mathit{Stmt}^*
\end{array}
$$

$$\mathit{Expr} = \mathit{ArithExpr} \mid \mathit{RelExpr} \mid \mathit{Id} \mid \mathit{ScalarValue}$$

$$
\begin{array}{lll}
\mathit{ArithExpr} :: & \mathit{opd}1 & : \mathit{Expr} \\
& \mathit{operator} & : \textsc{Plus} \mid \textsc{Minus} \\
& \mathit{opd}2 & : \mathit{Expr}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{RelExpr} :: & \mathit{opd}1 & : \mathit{Expr} \\
& \mathit{operator} & : \textsc{Equals} \mid \textsc{NotEquals} \\
& \mathit{opd}2 & : \mathit{Expr}
\end{array}
$$

$$\mathit{ScalarValue} = \mathbb{Z} \mid \mathbb{B}$$

A.2 Context conditions

In order to define the Context Conditions below, an auxiliary object is required in which the types of declared identifiers can be stored.

$$\mathit{TypeMap} = \mathit{Id} \xrightarrow{m} \mathit{ScalarType}$$

$$\mathit{wf\text{-}Program} : \mathit{Program} \to \mathbb{B}$$

$$\mathit{wf\text{-}Program}(\mathit{mk\text{-}Program}(\mathit{vars}, \mathit{body})) \triangleq \mathit{wf\text{-}StmtList}(\mathit{body}, \mathit{vars})$$

$$\mathit{wf\text{-}StmtList} : (\mathit{Stmt}^*) \times \mathit{TypeMap} \to \mathbb{B}$$

$$\mathit{wf\text{-}StmtList}(\mathit{sl}, \mathit{tpm}) \triangleq \forall i \in \mathbf{inds}\ \mathit{sl} \cdot \mathit{wf\text{-}Stmt}(\mathit{sl}(i), \mathit{tpm})$$

$$\mathit{wf\text{-}Stmt} : \mathit{Stmt} \times \mathit{TypeMap} \to \mathbb{B}$$

$$\mathit{wf\text{-}Stmt}(s, \mathit{tpm}) \triangleq \text{given by cases below}$$

$$
\begin{array}{l}
\mathit{wf\text{-}Stmt}(\mathit{mk\text{-}Assign}(\mathit{lhs}, \mathit{rhs}), \mathit{tpm}) \triangleq \\
\quad \mathit{lhs} \in \mathbf{dom}\ \mathit{tpm} \wedge \\
\quad \mathit{c\text{-}tp}(\mathit{rhs}, \mathit{tpm}) = \mathit{tpm}(\mathit{lhs})
\end{array}
$$

$$
\begin{array}{l}
\mathit{wf\text{-}Stmt}(\mathit{mk\text{-}If}(\mathit{test}, \mathit{th}, \mathit{el}), \mathit{tpm}) \triangleq \\
\quad \mathit{c\text{-}tp}(\mathit{test}, \mathit{tpm}) = \textsc{BoolTp} \wedge \\
\quad \mathit{wf\text{-}StmtList}(\mathit{th}, \mathit{tpm}) \wedge \mathit{wf\text{-}StmtList}(\mathit{el}, \mathit{tpm})
\end{array}
$$

$$\textit{wf-Stmt}(\textit{mk-While}(\textit{test}, \textit{body}), \textit{tpm}) \triangleq$$
$$\quad \textit{c-tp}(\textit{test}, \textit{tpm}) = \textsc{BoolTp} \wedge$$
$$\quad \textit{wf-StmtList}(\textit{body}, \textit{tpm})$$

An auxiliary function *c-tp* is defined

$$\textit{c-tp} : \textit{Expr} \times \textit{TypeMap} \to (\textsc{IntTp} \mid \textsc{BoolTp} \mid \textsc{Error})$$

$$\textit{c-tp}(e, \textit{tpm}) \triangleq \text{ given by cases below}$$

$$\textit{c-tp}(\textit{mk-ArithExpr}(e1, \textit{opt}, e2), \textit{tpm}) \triangleq$$
$$\quad \textbf{if } \textit{c-tp}(e1, \textit{tpm}) = \textsc{IntTp} \wedge \textit{c-tp}(e2, \textit{tpm}) = \textsc{IntTp}$$
$$\quad \textbf{then } \textsc{IntTp}$$
$$\quad \textbf{else } \textsc{Error}$$

$$\textit{c-tp}(\textit{mk-RelExpr}(e1, \textit{opt}, e2), \textit{tpm}) \triangleq$$
$$\quad \textbf{if } \textit{c-tp}(e1, \textit{tpm}) = \textsc{IntTp} \wedge \textit{c-tp}(e2, \textit{tpm}) = \textsc{IntTp}$$
$$\quad \textbf{then } \textsc{BoolTp}$$
$$\quad \textbf{else } \textsc{Error}$$

For the base cases:

$$e \in \textit{Id} \;\Rightarrow\; \textit{c-tp}(e, \textit{tpm}) = \textit{tpm}(e)$$

$$e \in \mathbb{Z} \;\Rightarrow\; \textit{c-tp}(e, \textit{tpm}) = \textsc{IntTp}$$

$$e \in \mathbb{B} \;\Rightarrow\; \textit{c-tp}(e, \textit{tpm}) = \textsc{BoolTp}$$

A.3 Semantics

An auxiliary object is needed to describe the Semantics — this "Semantic Object" (Σ) stores the association of identifiers and their values.

$$\Sigma = \textit{Id} \xrightarrow{m} \textit{ScalarValue}$$

$$\sigma_0 = \{\textit{id} \mapsto 0 \mid \textit{id} \in \mathbf{dom}\ \textit{vars} \wedge \textit{vars}(\textit{id}) = \textsc{IntTp}\} \cup$$
$$\qquad \{\textit{id} \mapsto \mathbf{true} \mid \textit{id} \in \mathbf{dom}\ \textit{vars} \wedge \textit{vars}(\textit{id}) = \textsc{BoolTp}\}$$
$$\frac{(\textit{body}, \sigma_0) \xrightarrow{sl} \sigma'}{(\textit{mk-Program}(\textit{vars}, \textit{body})) \xrightarrow{p} \textsc{Done}}$$

The semantic transition relation for statement lists is

$$\xrightarrow{sl} : \mathcal{P}((\textit{Stmt}^* \times \Sigma) \times \Sigma)$$

$$\frac{}{([\,], \sigma) \xrightarrow{sl} \sigma}$$

$$\frac{(s, \sigma) \xrightarrow{s} \sigma' \quad (rest, \sigma') \xrightarrow{sl} \sigma''}{([s] \curvearrowright rest, \sigma) \xrightarrow{sl} \sigma''}$$

The semantic transition relation for single statements is

$$\xrightarrow{s} : \mathcal{P}((Stmt \times \Sigma) \times \Sigma)$$

$$\frac{(rhs, \sigma) \xrightarrow{e} v}{(mk\text{-}Assign(lhs, rhs), \sigma) \xrightarrow{s} \sigma \dagger \{lhs \mapsto v\}}$$

$$\frac{(test, \sigma) \xrightarrow{e} \textbf{true} \quad (th, \sigma) \xrightarrow{sl} \sigma'}{(mk\text{-}If(test, th, el), \sigma) \xrightarrow{s} \sigma'}$$

$$\frac{(test, \sigma) \xrightarrow{e} \textbf{false} \quad (el, \sigma) \xrightarrow{sl} \sigma'}{(mk\text{-}If(test, th, el), \sigma) \xrightarrow{s} \sigma'}$$

$$\frac{(test, \sigma) \xrightarrow{e} \textbf{true} \quad (body, \sigma) \xrightarrow{sl} \sigma' \quad (mk\text{-}While(test, body), \sigma') \xrightarrow{s} \sigma''}{(mk\text{-}While(test, body), \sigma) \xrightarrow{s} \sigma''}$$

$$\frac{(test, \sigma) \xrightarrow{e} \textbf{false}}{(mk\text{-}While(test, body), \sigma) \xrightarrow{s} \sigma}$$

The semantic transition relation for expressions is

$$\xrightarrow{e} : \mathcal{P}((Expr \times \Sigma) \times ScalarValue)$$

$$\frac{(e1, \sigma) \xrightarrow{e} v1 \quad (e2, \sigma) \xrightarrow{e} v2}{(mk\text{-}ArithExpr(e1, \textsc{Plus}, e2), \sigma) \xrightarrow{e} v1 + v2}$$

$$\frac{(e1, \sigma) \xrightarrow{e} v1 \quad (e2, \sigma) \xrightarrow{e} v2}{(mk\text{-}ArithExpr(e1, \textsc{Minus}, e2), \sigma) \xrightarrow{e} v1 - v2}$$

$$\frac{(e1, \sigma) \xrightarrow{e} v1 \quad (e2, \sigma) \xrightarrow{e} v2 \quad v1 = v2}{(mk\text{-}RelExpr(e1, \textsc{Equals}, e2), \sigma) \xrightarrow{e} \textbf{true}}$$

$$\frac{(e1,\sigma) \xrightarrow{e} v1 \quad (e2,\sigma) \xrightarrow{e} v2 \quad v1 = v2}{(\mathit{mk\text{-}RelExpr}(e1, \textsc{NotEquals}, e2),\sigma) \xrightarrow{e} \textbf{false}}$$

$$\frac{e \in \mathit{Id}}{(e,\sigma) \xrightarrow{e} \sigma(e)}$$

$$\frac{e \in \mathit{ScalarValue}}{(e,\sigma) \xrightarrow{e} e}$$

B The language "Blocks"

This appendix summarizes one of the definitions discussed in Section 4 and shows a useful way in which a complete definition can be ordered. The "Blocks" language is described here with parameter passing by-location.

B.1 Auxiliary objects

The context conditions use:

$$\mathit{Types} = \mathit{Id} \xrightarrow{m} \mathit{Type}$$

$$\mathit{Type} = \mathit{ScalarType} \mid \mathit{FunType}$$

$$\begin{array}{lcll} \mathit{FunType} & :: & \mathit{returns} & : \mathit{ScalarType} \\ & & \mathit{paramtpl} & : \mathit{ScalarType}^* \end{array}$$

The semantic rules use:

$$\mathit{Env} = \mathit{Id} \xrightarrow{m} \mathit{Den}$$

$$\mathit{Den} = \mathit{ScalarLoc} \mid \mathit{FunDen}$$

Where *ScalarLoc* is an infinite set chosen from *Token*.

The types of the semantic relations are

$$\xrightarrow{p} : \mathcal{P}(\mathit{Program} \times \Sigma$$

$$\xrightarrow{sl} : \mathcal{P}(((\mathit{Stmt}^*) \times \mathit{Env} \times \Sigma) \times \Sigma)$$

$$\xrightarrow{s} : \mathcal{P}((\mathit{Stmt} \times \mathit{Env} \times \Sigma) \times \Sigma)$$

$$\xrightarrow{e} : \mathcal{P}((\mathit{Expr} \times \mathit{Env} \times \Sigma) \times \mathit{ScalarValue})$$

Abbreviations

$\sigma \in \Sigma$	a single "state"
Σ	the set of all "States"
Arith	Arithmetic
Def	Definition
Den	Denotation
env	a single "environment"
Env	the set of all "Environments"
Expr	Expression
Proc	Procedure
opd	operand
Rel	Relational
Sc	Scalar
Seq	Sequence
Stmt	Statement
...	

B.2 Programs

Abstract syntax

$$\mathit{Program} :: \mathit{Block}$$

$$\mathit{wf\text{-}Program} : \mathit{Program} \to \mathbb{B}$$

Context conditions $\mathit{wf\text{-}Program}(\mathit{mk\text{-}Program}(b)) \triangleq \mathit{wf\text{-}Block}(b, \{\,\})$

Semantics $$\frac{(b, \{\,\}, \{\,\}) \xrightarrow{s} \sigma'}{(\mathit{mk\text{-}Program}(b)) \xrightarrow{p} \sigma'}$$

B.3 Blocks

$$\begin{aligned} \mathit{Block} :: \ & \mathit{vars} : \mathit{Id} \xrightarrow{m} \mathit{ScalarType} \\ & \mathit{funs} : \mathit{Id} \xrightarrow{m} \mathit{Fun} \\ & \mathit{body} : \mathit{Stmt}^* \end{aligned}$$

Abstract syntax $\mathit{ScalarType} = \textsc{IntTp} \mid \textsc{BoolTp}$

$wf\text{-}Block : Block \times Types \to \mathbb{B}$

Context conditions $wf\text{-}Block(mk\text{-}Block(vars, funs, body), tps) \triangleq$
$\quad \mathbf{dom}\, vars \cap \mathbf{dom}\, funs = \{\,\} \wedge$
$\quad \mathbf{let}\ var\text{-}tps = tps \dagger vars\ \mathbf{in}$
$\quad \mathbf{let}\ fun\text{-}tps =$
$\quad\quad \{f \mapsto mk\text{-}FunType(funs(f).returns,$
$\quad\quad\quad apply(funs(f).params, funs(f).paramtps)) \mid$
$\quad\quad f \in \mathbf{dom}\, funs\}\ \mathbf{in}$
$\quad \forall f \in \mathbf{dom}\, funs \cdot wf\text{-}Fun(funs(f), var\text{-}tps)$
$\quad wf\text{-}StmtList(body, var\text{-}tps \dagger fun\text{-}tps)$

Notice that this rules out recursion.

Semantics
$$\frac{\begin{array}{l}(varenv, \sigma') = newlocs(vars, \sigma) \\ funenv = \\ \quad \{f \mapsto b\text{-}FunDen(funs(f), env \dagger varenv) \mid f \in \mathbf{dom}\, funs\} \\ env' = env \dagger varenv \dagger funenv \\ (body, env', \sigma') \xrightarrow{sl} \sigma''\end{array}}{(mk\text{-}Block(vars, funs, body), env, \sigma) \xrightarrow{s} (\mathbf{dom}\, \sigma) \lhd \sigma''}$$

$newlocs\ (vars{:}\,(Id \xrightarrow{m} ScalarType), \sigma{:}\,\Sigma)\ varenv{:}\,Env, \sigma'{:}\,\Sigma$
post $\mathbf{dom}\, varenv = \mathbf{dom}\, vars \wedge$
$\quad disj(\mathbf{rng}\, varenv, \mathbf{dom}\, \sigma) \wedge$
$\quad one\text{-}one(varenv) \wedge$
$\quad \sigma' = \sigma \cup \{varenv(id) \mapsto 0 \mid id \in \mathbf{dom}\, vars \wedge vars(id) = \textsc{IntTp}\} \cup$
$\quad\quad \{varenv(id) \mapsto \mathbf{true} \mid id \in \mathbf{dom}\, vars \wedge vars(id) = \textsc{BoolTp}\}$

B.4 Function definitions

Fun	:: $returns$	:	$ScalarType$
	$params$	:	Id^*
	$paramtps$	:	$Id \xrightarrow{m} ScalarType$
	$body$	:	$Stmt^*$
	$result$	:	$Expr$

Abstract syntax

[30] The auxiliary function *one-one* is defined:

$one\text{-}one : (X \xrightarrow{m} Y) \to \mathbb{B}$
$one\text{-}one(m) \triangleq \forall a, b \in \mathbf{dom}\, m \cdot m(a) = m(b) \Rightarrow a = b$

$wf\text{-}Fun : Fun \times Types \to \mathbb{B}$

Context conditions

$wf\text{-}Fun(mk\text{-}Fun(returns, params, paramtps, body, result), tps) \triangleq$
$uniquel(params) \wedge$
$\textbf{elems}\ params = \textbf{dom}\ paramtps \wedge$
$tp(result) = returns \wedge$
$wf\text{-}StmtList(body, tps \dagger paramtps)$

$b\text{-}Fun\text{-}Den : Fun \times Env \to FunDen$

$b\text{-}Fun\text{-}Den(mk\text{-}Fun(returns, params, paramtps, body, result), env) \triangleq$
$mk\text{-}FunDen(params, body, result, env)$

B.5 Statement lists

$wf\text{-}StmtList : Stmt^* \times Types \to \mathbb{B}$

Context conditions $wf\text{-}StmtList(sl, tps) \triangleq$
$\forall i \in \textbf{inds}\ sl \cdot wf\text{-}Stmt(sl(i), tps)$

Semantics $\dfrac{}{([\,], env, \sigma) \xrightarrow{sl} \sigma}$

$$\frac{(s, env, \sigma) \xrightarrow{s} \sigma' \quad (rest, env, \sigma') \xrightarrow{sl} \sigma''}{([s] \curvearrowright rest, env, \sigma) \xrightarrow{sl} \sigma''}$$

B.6 Statements

Abstract syntax $Stmt = Block \mid Assign \mid If \mid Call$

B.7 Assignments

$Assign :: lhs : Id$
$\quad\quad rhs : Expr$

Abstract syntax

Context conditions $wf\text{-}Stmt(mk\text{-}Assign(lhs, rhs), tps) \triangleq$
$tp(rhs, tps) = tp(lhs, tps)$

Semantics $\dfrac{(lhs, env, \sigma) \xrightarrow{lhv} l \quad (rhs, env, \sigma) \xrightarrow{e} v}{(mk\text{-}Assign(lhs, rhs), env, \sigma) \xrightarrow{s} \sigma \dagger \{l \mapsto v\}}$

B.8 If statements

$$\begin{array}{lll} If :: test & : & Expr \\ \quad th & : & Stmt^* \\ \quad el & : & Stmt^* \end{array}$$

Abstract syntax

Context conditions $wf\text{-}Stmt(mk\text{-}If(test, th, el), tps) \triangleq$
$tp(test, tps) = \textsc{BoolTp} \wedge$
$wf\text{-}StmtList(th, tps) \wedge wf\text{-}StmtList(el, tps)$

Semantics $\dfrac{(test, env, \sigma) \xrightarrow{e} \textbf{true} \quad (th, env, \sigma) \xrightarrow{sl} \sigma'}{(mk\text{-}If(test, th, el), env, \sigma) \xrightarrow{s} \sigma'}$

$$\frac{(test, env, \sigma) \xrightarrow{e} \textbf{false} \quad (el, env, \sigma) \xrightarrow{sl} \sigma'}{(mk\text{-}If(test, th, el), env, \sigma) \xrightarrow{s} \sigma'}$$

B.9 Call statements

$$\begin{array}{lll} Call :: lhs & : & VarRef \\ \quad fun & : & Id \\ \quad args & : & Id^* \end{array}$$

Abstract syntax

Context conditions $wf\text{-}Stmt(mk\text{-}Call(lhs, fun, args), tps) \triangleq$
$fun \in \textbf{dom}\ tps \wedge$
$tps(fun) \in FunType \wedge$
$tp(lhs, tps) = (tps(fun)).returns \wedge$
$\textbf{len}\ args = \textbf{len}\ (tps(fun)).paramtpl \wedge$
$\forall i \in \textbf{inds}\ args \cdot tp(args(i), tps) = ((tps(fun)).paramtpl)(i)$

$$\textbf{Semantics}\ \frac{\begin{array}{l}(lhs, env, \sigma) \xrightarrow{lhv} l \\ mk\text{-}FunDen(parms, body, result, context) = env(f) \\ \textbf{len}\ arglocs = \textbf{len}\ args \\ \forall i \in \textbf{inds}\ arglocs \cdot (args(i), env, \sigma) \xrightarrow{lhv} arglocs(i) \\ parm\text{-}env = \{parms(i) \mapsto arglocs(i) \mid i \in \textbf{inds}\ parms\} \\ (body, (context \dagger parm\text{-}env), \sigma) \xrightarrow{sl} \sigma' \\ (result, (context \dagger parm\text{-}env), \sigma') \xrightarrow{e} res\end{array}}{(mk\text{-}Call(lhs, f, args), env, \sigma) \xrightarrow{s} (\sigma' \dagger \{l \mapsto res\})}$$

B.10 Expressions

Abstract syntax $Expr = ArithExpr \mid RelExpr \mid Id \mid ScalarValue$

$$\begin{array}{lll} ArithExpr :: & opd1 & : Expr \\ & operator & : \textsc{Plus} \\ & opd2 & : Expr \end{array}$$

$$\begin{array}{lll} RelExpr :: & opd1 & : Expr \\ & operator & : \textsc{Equals} \\ & opd2 & : Expr \end{array}$$

$ScalarValue = \mathbb{Z} \mid \mathbb{B}$

$$\textbf{Semantics}\ \frac{\begin{array}{l}(e1, env, \sigma) \xrightarrow{e} v1 \\ (e2, env, \sigma) \xrightarrow{e} v2\end{array}}{(mk\text{-}ArithExpr(e1, \textsc{Plus}, e2), env, \sigma) \xrightarrow{e} v1 + v2}$$

$$\frac{\begin{array}{l}(e1, env, \sigma) \xrightarrow{e} v1 \\ (e2, env, \sigma) \xrightarrow{e} v2 \\ v1 = v2\end{array}}{(mk\text{-}RelExpr(e1, \textsc{Equals}, e2), env, \sigma) \xrightarrow{e} \textbf{true}}$$

$$\frac{\begin{array}{l}e \in Id \\ (id, env, \sigma) \xrightarrow{lhv} l\end{array}}{(e, env, \sigma) \xrightarrow{e} \sigma(l)}$$

$$\frac{e \in ScalarValue}{(e, env, \sigma) \xrightarrow{e} e}$$

C COOL

Reordered definition from Section 6.

C.1 Auxiliary objects

The objects required for both Context Conditions and Semantic Rules are given first.

Objects needed for context conditions

The following objects are needed in the description of the Context Conditions.

$$ClassTypes = Id \xrightarrow{m} ClassInfo$$

$$ClassInfo = Id \xrightarrow{m} MethInfo$$

The only information required about methods is about their types (arguments and results):

$$\begin{array}{lll} MethInfo :: & return & : Type \\ & parms & : Type^* \end{array}$$

$$Type = Id \mid ScalarType$$

$$ScalarType = \textsc{IntTp} \mid \textsc{BoolTp}$$

When checking for the well-formedness of the body of a *Method*, information about its instance variables is also needed

$$VarEnv = Id \xrightarrow{m} Type$$

Semantic objects

In addition to the abstract syntax of *Classes* (see below), the following objects are needed in the description of the Semantics.

$$ObjMap = Reference \xrightarrow{m} ObjInfo$$

$$\begin{array}{lll} ObjInfo :: & class & : Id \\ & state & : VarState \\ & status & : Status \\ & remaining & : Stmt^* \\ & client & : [Reference] \end{array}$$

$$VarState = Id \xrightarrow{m} Val$$

$$Val = [Reference] \mid \mathbb{Z} \mid \mathbb{B}$$

The set *Reference* is infinite and $\mathbf{nil} \notin Reference$.

$$Status = \textsc{Active} \mid \textsc{Idle} \mid Wait$$

$$Wait :: lhs : Id$$

The types of the semantic relations are

$$\xrightarrow{p} : \mathcal{P}(\mathit{Program} \times \textsc{Done})$$

$$\xrightarrow{s} : \mathcal{P}((\mathit{Classes} \times \mathit{ObjMap}) \times \mathit{ObjMap})$$

$$\xrightarrow{e} : \mathcal{P}((\mathit{Expr} \times \mathit{VarState}) \times \mathit{Val})$$

Abbreviations

Arith	Arithmetic
Expr	Expression
Obj	Object
opd	operand
Meth	Method
Rel	Relational
Stmt	Statement
Var	Variable

C.2 Programs

Abstract syntax

$$\begin{array}{lll} \mathit{Program} :: & \mathit{cm} & : \mathit{Classes} \\ & \mathit{start\text{-}class} & : \mathit{Id} \\ & \mathit{start\text{-}meth} & : \mathit{Id} \end{array}$$

$$\mathit{Classes} = \mathit{Id} \xrightarrow{m} \mathit{ClassBlock}$$

Context conditions

$$\mathit{wf\text{-}Program} : \mathit{Program} \to \mathbb{B}$$

$$\begin{array}{l} \mathit{wf\text{-}Program}(\mathit{mk\text{-}Program}(\mathit{cm}, \mathit{start\text{-}c}, \mathit{start\text{-}m})) \triangleq \\ \quad \mathit{start\text{-}c} \in \mathbf{dom}\ \mathit{cm} \wedge \\ \quad \mathit{start\text{-}m} \in \mathbf{dom}\ (\mathit{cm}(\mathit{start\text{-}c}).\mathit{meths}) \wedge \\ \quad \mathbf{let}\ \mathit{ctps} = \{c \mapsto \mathit{c\text{-}tp}(\mathit{cm}(c)) \mid c \in \mathbf{dom}\ \mathit{cm}\} \\ \quad \mathbf{in}\ \forall c \in \mathbf{dom}\ \mathit{cm} \cdot \mathit{wf\text{-}ClassBlock}(\mathit{cm}(c), \mathit{ctps}) \end{array}$$

The following two functions extract *ClassInfo* and *MethInfo* respectively.

$$\mathit{c\text{-}tp} : \mathit{ClassBlock} \to \mathit{ClassInfo}$$

$$\begin{array}{l} \mathit{c\text{-}tp}(\mathit{mk\text{-}ClassBlock}(\mathit{tpm}, \mathit{mm})) \triangleq \\ \quad \{m \mapsto \mathit{c\text{-}minfo}(\mathit{mm}(m)) \mid m \in \mathbf{dom}\ \mathit{mm}\} \end{array}$$

$$c\text{-}minfo : Meth \to MethInfo$$

$$c\text{-}minfo(mk\text{-}Meth(ret, pnl, ptm, b)) \triangleq mk\text{-}MethInfo(ret, apply(pnl, ptm))$$

Semantics

With no input/output statements, the execution of a *Program* actually leaves no trace. One might say that, for $mk\text{-}Program(cm, init\text{-}class, init\text{-}meth)$, the initial O is such that

$$\begin{aligned}
&a \in Reference \\
&mk\text{-}ClassBlock(vars_0, meths_0) = cm(init\text{-}class) \\
&\sigma_0 = \{v \mapsto \mathbf{nil} \mid v \in \mathbf{dom}\,(vars_0) \wedge vars_0(v) \notin ScalarType\} \cup \\
&\qquad \{v \mapsto \mathbf{false} \mid v \in \mathbf{dom}\,(vars_0) \wedge vars_0(v) = \textsc{BoolTp}\} \cup \\
&\qquad\quad \{v \mapsto 0 \mid v \in \mathbf{dom}\,(vars_0) \wedge vars_0(v) = \textsc{IntTp}\} \\
&sl_0 = meths_0(init\text{-}meth).body \\
&O = \{a \mapsto mk\text{-}ObjInfo(init\text{-}class, \sigma_0, \textsc{Active}, sl_0, \mathbf{nil})\}
\end{aligned}$$

and that execution ceases when there are no more "threads" active. It would, of course, be more useful to look at running a program against an "object store" from the file system; such an extension is straightforward but somewhat outside the realm of the language itself.

C.3 Classes

Abstract syntax

$$\begin{aligned}
ClassBlock \;::\; &vars \;\; : Id \xrightarrow{m} Type \\
&meths \; : Id \xrightarrow{m} Meth
\end{aligned}$$

Context conditions

$$wf\text{-}ClassBlock : ClassBlock \times ClassTypes \to \mathbb{B}$$

$$\begin{aligned}
&wf\text{-}ClassBlock(mk\text{-}ClassBlock(tpm, mm), ctps) \triangleq \\
&\quad \forall id \in \mathbf{dom}\, tpm \cdot (tpm(id) \in ScalarType \vee tpm(id) \in \mathbf{dom}\, ctps) \wedge \\
&\quad \forall m \in \mathbf{dom}\, mm \cdot wf\text{-}Meth(mm(m), ctps, tpm)
\end{aligned}$$

Semantics

There are no semantics for classes as such — see the semantics of *New* in Section C.8.

C.4 Methods

Abstract syntax

$$
\begin{array}{lll}
Meth :: & returns & : Type \\
& params & : Id^* \\
& paramtps & : Id \xrightarrow{m} Type \\
& body & : Stmt^*
\end{array}
$$

Context conditions

$$wf\text{-}Meth : Meth \times ClassTypes \times VarEnv \to \mathbb{B}$$

$$
\begin{array}{l}
wf\text{-}Meth(mk\text{-}Meth(ret, pnl, ptm, b), ctps, v\text{-}env) \quad \triangleq \\
\quad (ret \in ScalarType \lor ret \in \mathbf{dom}\ ctps)) \land \\
\quad \forall id \in \mathbf{dom}\ ptm \cdot (ptm(id) \in ScalarType \lor ptm(id) \in \mathbf{dom}\ ctps) \land \\
\quad \mathbf{elems}\ pnl \subseteq \mathbf{dom}\ ptm \land \\
\quad \forall i \in \mathbf{inds}\ b \cdot wf\text{-}Stmt(b(i), ctps, v\text{-}env \dagger ptm)
\end{array}
$$

Semantics

There are no semantics for methods as such — see the semantics of method invocation in Section C.9.

C.5 Statements

$$Stmt = Assign \mid If \mid New \mid MethCall \mid Return \mid Release \mid Delegate$$

Context conditions

$$wf\text{-}Stmt : Stmt \times ClassTypes \times VarEnv \to \mathbb{B}$$

$$wf\text{-}Stmt(s, ctps, v\text{-}env) \quad \triangleq \quad \text{by cases below}$$

C.6 Assignments

Remember that method calls cannot occur in an *Assign* – method invocation is covered in Section C.9.

Abstract syntax

$$\begin{array}{lll} \mathit{Assign} :: & \mathit{lhs} & : \mathit{Id} \\ & \mathit{rhs} & : \mathit{Expr} \end{array}$$

Context conditions

$$\begin{array}{l} \mathit{wf\text{-}Stmt}(\mathit{mk\text{-}Assign}(\mathit{lhs}, \mathit{rhs}), \mathit{ctps}, \mathit{v\text{-}env}) \quad \triangleq \\ \quad \mathit{lhs} \in \mathbf{dom}\ \mathit{v\text{-}env} \wedge \\ \quad \mathit{tp}(\mathit{rhs}, \mathit{ctps}, \mathit{v\text{-}env}) = \mathit{v\text{-}env}(\mathit{lhs}) \end{array}$$

Semantics

$$\frac{\begin{array}{l} O(a) = \mathit{mk\text{-}ObjInfo}(c, \sigma, \textsc{Active}, [\mathit{mk\text{-}Assign}(\mathit{lhs}, \mathit{rhs})] \curvearrowright \mathit{rl}, \mathit{co}) \\ (\mathit{rhs}, \sigma) \xrightarrow{e} v \\ \mathit{aobj}' = \mathit{mk\text{-}ObjInfo}(c, \sigma \dagger \{\mathit{lhs} \mapsto v\}, \textsc{Active}, \mathit{rl}, \mathit{co}) \end{array}}{(C, O) \xrightarrow{s} O \dagger \{a \mapsto \mathit{aobj}'\}}$$

C.7 If statements

Abstract syntax

$$\begin{array}{lll} \mathit{If} :: & \mathit{test} & : \mathit{Expr} \\ & \mathit{th} & : \mathit{Stmt}^* \\ & \mathit{el} & : \mathit{Stmt}^* \end{array}$$

Context conditions

$$\begin{array}{l} \mathit{wf\text{-}Stmt}(\mathit{mk\text{-}If}(\mathit{test}, \mathit{th}, \mathit{el}), \mathit{ctps}, \mathit{v\text{-}env}) \quad \triangleq \\ \quad \mathit{tp}(\mathit{test}, \mathit{ctps}, \mathit{v\text{-}env}) = \textsc{BoolTp} \wedge \\ \quad \forall i \in \mathbf{inds}\ \mathit{th} \cdot \mathit{wf\text{-}Stmt}(\mathit{th}(i), \mathit{ctps}, \mathit{v\text{-}env}) \wedge \\ \quad \forall i \in \mathbf{inds}\ \mathit{el} \cdot \mathit{wf\text{-}Stmt}(\mathit{el}(i), \mathit{ctps}, \mathit{v\text{-}env}) \end{array}$$

Semantics

$$\frac{\begin{array}{l} O(a) = \mathit{mk\text{-}ObjInfo}(c, \sigma, \textsc{Active}, [\mathit{mk\text{-}If}(\mathit{test}, \mathit{th}, \mathit{el})] \curvearrowright \mathit{rl}, \mathit{co}) \\ (\mathit{test}, \sigma) \xrightarrow{e} \mathbf{true} \\ \mathit{aobj}' = \mathit{mk\text{-}ObjInfo}(c, \sigma, \textsc{Active}, [\mathit{th}] \curvearrowright \mathit{rl}, \mathit{co}) \end{array}}{(C, O) \xrightarrow{s} O \dagger \{a \mapsto \mathit{aobj}'\}}$$

$$\frac{\begin{array}{l} O(a) = \mathit{mk\text{-}ObjInfo}(c, \sigma, \textsc{Active}, [\mathit{mk\text{-}If}(\mathit{test}, \mathit{th}, \mathit{el})] \curvearrowright \mathit{rl}, \mathit{co}) \\ (\mathit{test}, \sigma) \xrightarrow{e} \mathbf{false} \\ \mathit{aobj}' = \mathit{mk\text{-}ObjInfo}(c, \sigma, \textsc{Active}, [\mathit{el}] \curvearrowright \mathit{rl}, \mathit{co}) \end{array}}{(C, O) \xrightarrow{s} O \dagger \{a \mapsto \mathit{aobj}'\}}$$

C.8 Creating objects

Abstract syntax

$$\begin{array}{lrcl} \mathit{New} & :: & \mathit{targ} & : & \mathit{Id} \\ & & \mathit{class} & : & \mathit{Id} \end{array}$$

Context conditions

$$\begin{array}{l} \mathit{wf\text{-}Stmt}(\mathit{mk\text{-}New}(\mathit{targ}, \mathit{class}), \mathit{ctps}, \mathit{v\text{-}env}) \quad \triangleq \\ \quad \mathit{class} \in \mathbf{dom}\ \mathit{ctps} \wedge \\ \quad \mathit{class} = \mathit{v\text{-}env}(\mathit{targ}) \end{array}$$

Semantics

$$\frac{\begin{array}{l} O(a) = \mathit{mk\text{-}ObjInfo}(c, \sigma, \textsc{Active}, [\mathit{mk\text{-}New}(\mathit{targ}, c')] \curvearrowright \mathit{rl}, \mathit{co}) \\ b \in (\mathit{Reference} - \mathbf{dom}\ O) \\ \mathit{aobj}' = \mathit{mk\text{-}ObjInfo}(c, \sigma \dagger \{\mathit{targ} \mapsto b\}, \textsc{Active}, \mathit{rl}, \mathit{co}) \\ \sigma_b = \\ \{v \mapsto 0 \mid v \in \mathbf{dom}\,(C(c').\mathit{vars}) \wedge (C(c').\mathit{vars})(v) = \textsc{IntTp}\} \cup \\ \{v \mapsto \mathbf{false} \mid v \in \mathbf{dom}\,(C(c').\mathit{vars}) \wedge (C(c').\mathit{vars})(v) = \textsc{BoolTp}\} \cup \\ \quad \{v \mapsto \mathbf{nil} \mid v \in \mathbf{dom}\,(C(c').\mathit{vars}) \wedge (C(c').\mathit{vars})(v) \notin \mathit{ScalarType}\} \\ \mathit{nobj} = \mathit{mk\text{-}ObjInfo}(c', \sigma_b, \textsc{Idle}, [\,], \mathbf{nil}) \end{array}}{(C, O) \xrightarrow{s} O \dagger \{a \mapsto \mathit{aobj}', b \mapsto \mathit{nobj}\}}$$

C.9 Invoking and completing methods

Abstract syntax

$$\begin{array}{lrcl} \mathit{MethCall} & :: & \mathit{lhs} & : & \mathit{Id} \\ & & \mathit{obj} & : & \mathit{Id} \\ & & \mathit{meth} & : & \mathit{Id} \\ & & \mathit{args} & : & \mathit{Id}^* \end{array}$$

Context conditions

$$\begin{array}{l} \mathit{wf\text{-}Stmt}(\mathit{mk\text{-}MethCall}(\mathit{lhs}, \mathit{obj}, \mathit{meth}, \mathit{args}), \mathit{ctps}, \mathit{v\text{-}env}) \quad \triangleq \\ \quad \mathit{obj} \in \mathbf{dom}\ \mathit{ctps} \wedge \\ \quad \mathit{meth} \in \mathbf{dom}\,(\mathit{ctps}(\mathit{obj})) \wedge \\ \quad ((\mathit{ctps}(\mathit{obj}))(\mathit{meth})).\mathit{return} = \mathit{v\text{-}env}(\mathit{lhs}) \wedge \\ \quad \mathbf{len}\ \mathit{args} = \mathbf{len}\,((\mathit{ctps}(\mathit{obj}))(\mathit{meth})).\mathit{parms} \wedge \\ \quad \forall i \in \mathbf{inds}\ \mathit{args} \cdot \\ \qquad \mathit{tp}(\mathit{args}(i), \mathit{ctps}, \mathit{v\text{-}env}) = (((\mathit{ctps}(\mathit{obj}))(\mathit{meth})).\mathit{parms})(i) \end{array}$$

Semantics

$$\frac{\begin{array}{l} O(a) = \\ \quad \textit{mk-ObjInfo}(c, \sigma, \textsc{Active}, [\textit{mk-MethCall}(lhs, obj, meth, args)] \curvearrowright rl, co) \\ O(\sigma(obj)) = \textit{mk-ObjInfo}(c', \sigma', \textsc{Idle}, [\,], \textsc{nil}) \\ C(c') = \textit{mk-ClassBlock}(vars, meths) \\ aobj' = \textit{mk-ObjInfo}(c, \sigma, \textit{mk-Wait}(lhs), rl, co) \\ \sigma'' = \sigma' \dagger \{(meths(meth).params)(i) \mapsto \sigma(args(i)) \mid i \in \textbf{inds}\ args\} \\ sobj = \textit{mk-ObjInfo}(c', \sigma'', \textsc{Active}, meths(meth).body, a) \end{array}}{(C, O) \xrightarrow{s} O \dagger \{a \mapsto aobj', \sigma(obj) \mapsto sobj\}}$$

When a method has no more statements to execute (remember the *Release* can have occured earlier) it returns to the quiescent status.

$$\frac{\begin{array}{l} O(a) = \textit{mk-ObjInfo}(c, \sigma, \textsc{Active}, [\,], co) \\ aobj' = \textit{mk-ObjInfo}(c, \sigma, \textsc{Idle}, [\,], \textbf{nil}) \end{array}}{(C, O) \xrightarrow{s} O \dagger \{a \mapsto aobj'\}}$$

C.10 Returning values

Abstract syntax

$$\textit{Return} :: val : (\textit{Expr} \mid \textsc{Self})$$

Context conditions

$$\textit{wf-Stmt}(\textit{mk-Return}(val), ctps, \textit{v-env}) \quad \triangleq$$
incomplete

Semantics

The cases of an *Expr* and Self separately.

$$\frac{\begin{array}{l} O(a) = \textit{mk-ObjInfo}(c, \sigma, \textsc{Active}, [\textit{mk-Return}(e)] \curvearrowright rl, co) \\ e \in \textit{Expr} \\ (e, \sigma) \xrightarrow{e} v \\ O(co) = \textit{mk-ObjInfo}(c', \sigma', \textit{mk-Wait}(lhs), sl, co') \\ aobj' = \textit{mk-ObjInfo}(c, \sigma, \textsc{Idle}, [\,], \textbf{nil}) \\ cobj' = \textit{mk-ObjInfo}(c', \sigma' \dagger \{lhs \mapsto v\}, \textsc{Active}, sl, co') \end{array}}{(C, O) \xrightarrow{s} O \dagger \{a \mapsto aobj', co \mapsto cobj'\}}$$

$$\frac{\begin{array}{l} O(a) = \textit{mk-ObjInfo}(c, \sigma, \textsc{Active}, [\textit{mk-Return}(e)] \curvearrowright rl, co) \\ e = \textsc{Self} \\ O(co) = \textit{mk-ObjInfo}(c', \sigma', \textit{mk-Wait}(lhs), sl, co') \\ aobj' = \textit{mk-ObjInfo}(c, \sigma, \textsc{Idle}, [\,], \textbf{nil}) \\ cobj' = \textit{mk-ObjInfo}(c', \sigma' \dagger \{lhs \mapsto a\}, \textsc{Active}, sl, co') \end{array}}{(C, O) \xrightarrow{s} O \dagger \{a \mapsto aobj', co \mapsto cobj'\}}$$

Release is more general than a *Return* in the sense that the former does not have to terminate a method.

Abstract syntax

$Release :: val : (Expr \mid \textsc{Self})$

Context conditions

$wf\text{-}Stmt(mk\text{-}Release(val), ctps, v\text{-}env) \triangleq$
incomplete

Semantics

The cases of an *Expr* and SELF are considered separately.

$$\frac{\begin{array}{l} O(a) = mk\text{-}ObjInfo(c, \sigma, \textsc{Active}, [mk\text{-}Release(e)] \curvearrowright rl, co) \\ e \in Expr \\ (e, \sigma) \xrightarrow{e} v \\ O(co) = mk\text{-}ObjInfo(c', \sigma', mk\text{-}Wait(lhs), sl, co') \\ aobj' = mk\text{-}ObjInfo(c, \sigma, \textsc{Active}, rl, \mathbf{nil}) \\ cobj' = mk\text{-}ObjInfo(c', \sigma' \dagger \{lhs \mapsto v\}, \textsc{Active}, sl, co') \end{array}}{(C, O) \xrightarrow{s} O \dagger \{a \mapsto aobj', co \mapsto cobj'\}}$$

$$\frac{\begin{array}{l} O(a) = mk\text{-}ObjInfo(c, \sigma, \textsc{Active}, [mk\text{-}Release(e)] \curvearrowright rl, co) \\ e = \textsc{Self} \\ O(co) = mk\text{-}ObjInfo(c', \sigma', mk\text{-}Wait(lhs), sl, co') \\ aobj' = mk\text{-}ObjInfo(c, \sigma, \textsc{Active}, rl, \mathbf{nil}) \\ cobj' = mk\text{-}ObjInfo(c', \sigma' \dagger \{lhs \mapsto a\}, \textsc{Active}, sl, co') \end{array}}{(C, O) \xrightarrow{s} O \dagger \{a \mapsto aobj', co \mapsto cobj'\}}$$

C.11 Delegation

Abstract syntax

$$\begin{array}{lll} Delegate :: & obj & : Id \\ & meth & : Id \\ & args & : Id^* \end{array}$$

Context conditions

$wf\text{-}Stmt(mk\text{-}Delegate(obj, meth, args), ctps, v\text{-}env) \triangleq$
incomplete

Semantics

$$
\begin{array}{l}
O(a) = \\
\quad \mathit{mk\text{-}ObjInfo}(c, \sigma, \textsc{Active}, [\mathit{mk\text{-}Delegate}(\mathit{obj}, \mathit{meth}, \mathit{args})] \curvearrowright \mathit{rl}, \mathit{co}) \\
O(\sigma(\mathit{obj})) = \mathit{mk\text{-}ObjInfo}(c', \sigma', \textsc{Idle}, [\,], \textbf{nil}) \\
C(c') = \mathit{mk\text{-}ClassBlock}(\mathit{vars}, \mathit{meths}) \\
\mathit{aobj}' = \mathit{mk\text{-}ObjInfo}(c, \sigma, \textsc{Active}, \mathit{rl}, \textbf{nil}) \\
\sigma'' = \sigma' \dagger \{(\mathit{meths}(\mathit{meth}).\mathit{params})(i) \mapsto \sigma(\mathit{args}(i)) \mid i \in \textbf{inds}\ \mathit{args}\} \\
\mathit{sobj} = \mathit{mk\text{-}ObjInfo}(c', \sigma'', \textsc{Active}, \mathit{meths}(\mathit{meth}).\mathit{body}, \mathit{co}) \\
\hline
(C, O) \xrightarrow{s} O \dagger \{a \mapsto \mathit{aobj}', \sigma(\mathit{obj}) \mapsto \mathit{sobj}\}
\end{array}
$$

C.12 Expressions

Abstract syntax

$$\mathit{Expr} = \mathit{ArithExpr} \mid \mathit{RelExpr} \mid \mathit{TestNil} \mid \mathit{Id} \mid \mathit{ScalarValue} \mid \textbf{nil}$$

$$
\begin{array}{lll}
\mathit{ArithExpr} :: & \mathit{opd}1 & : \mathit{Expr} \\
& \mathit{operator} & : \textsc{Plus} \\
& \mathit{opd}2 & : \mathit{Expr}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{RelExpr} :: & \mathit{opd}1 & : \mathit{Expr} \\
& \mathit{operator} & : \textsc{Equals} \\
& \mathit{opd}2 & : \mathit{Expr}
\end{array}
$$

$$\mathit{TestNil} :: \mathit{obj} : \mathit{Id}$$

$$\mathit{ScalarValue} = \mathbb{Z} \mid \mathbb{B}$$