Black tie optional: Modeling programming language concepts

J W Coleman, N P Jefferson, and C B Jones

School of Computing Science
University of Newcastle upon Tyne
NE1 7RU, UK
e-mail: {j.w.coleman,n.p.jefferson,cliff.jones}@ncl.ac.uk

Abstract. This paper describes an undergraduate course taught at the University of Newcastle upon Tyne; the title of the module is *Understanding Programming Languages*. The main thrust of the course is to understand how to model features of language semantics. Specifically, (structural) operational semantics (SOS) is taught as a convenient and notational light way of recording and experimenting with features of procedural programming languages. We outline the content, discuss the contentious issue of tool support and relate experiences.

1 Introduction

The course discussed in this paper is entitled "Understanding Programming Languages".[1] (For brevity, the module is referred to below by its number "CSC334".) It teaches the modeling of *concepts* from programming languages. Formally, it covers operational semantics using parts of VDM for the formal notation (including an unconventional emphasis on "abstract syntax" (see Section 4) but tries not to labour the formalism itself. The teaching objectives are about the student being able to

- read a formal (operational) semantics
- experiment with language ideas by sketching a model

The authors of this paper each bring a different viewpoint to the course being discussed

- the third author has worked on formal semantics since the operational semantics of VDL; contributed to the move to the denotational semantics of VDM; and is the principal teacher of the course being described
- the second author is a postgraduate student working with SOS. He is in the unique position of having experienced the course both as a student and a teaching assistant and is a strong advocate of the use of tool support as a teaching aid.

[1] A book with the same title is being written.

– the first author is currently a postgraduate student using SOS to model CA Actions [XRR+99] in BPEL. He's also had the pleasure of demonstrating for CSC334 this past year, watching –and helping– students understand SOS. He's a bit of a closet LISPer, having been part of a functional programming micro-community while in undergrad at Ryerson University (in Toronto). Immediately prior to returning to academia, he spent a couple of years in industry as a software developer.

It might be useful to say a few words about the context in which the course at Newcastle is taught.

The School of Computing Science at Newcastle University offers several undergraduate "degree programmes" each of which offers the CSC334 module as an optional final year module. The module is taught to a wide variety of students with varying degrees of experience with formal methods.

No prerequisites are required of the students who enlist in the CSC334 course, but students from most degree programmes take compulsory 2nd year modules that teach VDM as an introduction to formal methods (the textbook used for this is [FL98]). Interestingly Newcastle University does not offer a module on compiling and this has to be taken into account in the delivery of CSC334.

2 Why teach (formal) semantics?

There are several reasons for teaching formal semantics at undergraduate level. Probably the strongest can be motivated from the "half life of knowledge" that can be imparted: programming languages come and go – over previous ten year periods, there have been complete changes in the fortunes of one or another language (e.g. Pascal, Modula-n, Ada, C, C++ and Java just within main line procedural languages). Any language that we teach in a university course today could have been added to the list of faintly remembered languages in a decade's time. It therefore behooves academics to try to teach something which will last longer and give students a way to look at future languages. There are of course very good books on comparative languages (a recent example is [Wat04]). The fact that so many of the programming languages –even those which are widely used– exhibit really bad design decisions is also worrying and indicates that there is a need to give future computer scientists ways to explore ideas more economically than by building a compiler.

The idea of teaching students a way to *model* concepts in programming languages is attractive in itself but it also provides an opportunity to say things about the fundamental nature of Informatics. Computing science is not a natural science in which one is stuck with modeling the universe as it exists; but nor is it usefully viewed as a branch of mathematics because one cannot ignore what can be realised in an engineering sense. This tension is nowhere more clearly seen than in the design of programming languages. They must find a compromise between

– clarity of expression of programs written *in* the language

– reasonable performance of implementations *of* the language

Of course, this list could be extended to include all sorts of issues like the ability to diagnose programmers' errors but the essential tension is that indicated above.

The course then explores language definition questions like

– the separation of syntax and semantic issues
– the nature of procedural programs
– "strong typing"
– linking programs to their data
– the role of objects in OOLs

3 Why choose Operational Semantics?

The next question to consider is why we choose to base the course on "Operational Semantics". Research into formal semantics is often classified under the terms:

– Operational Semantics
– Denotational Semantics
– Axiomatic Semantics

As indicated in Section 1, one of the authors of this paper has been involved in the development of formal semantic techniques since attempting to use the early operational semantics work on VDL [LW69,Luc81] in the design of compilers (cf. [BJ78] for itself and further references). The move to denotational semantics (see [Sto77]) by the IBM Laboratory Vienna gave rise to the language description parts of VDM [BBH+74,BJ78,BJ82].[2] Based on his enthusiasm for the newer work on denotational semantics, this same author's courses in Manchester University (1981–96) used that approach. It is therefore something of a *volte face* that he now chooses to teach operational semantics in the course in Newcastle. The essence of operational semantics is that it provides what John McCarthy called an "abstract interpreter" for the language under study. Both words are important. An interpreter makes clear how programs are executed; for an imperative language, it shows how statements cause changes to the state of the computation. The importance of this being described "abstractly" cannot be overemphasised: the interpretation can be understood because it is presented in terms of abstract objects.

4 Technical material covered

This section describes in more detail the material covered in the subject course; the interesting question of tool support is deferred to Section 8.

Because the interest is in *modeling* (rather than the meta-theory of semantics), the course teaches by example: a series of three language definitions are tackled:

[2] This development is described in [Jon01]; the wider history of VDM in [Jon99].

– *Base* introduces the basic idea of states and abstract interpretation; after beginning with a simple deterministic language, concurrency is used to explain the need to cope with non-determinism; a trivial (and rather dangerous) form of threads with sequences of unguarded assignments is modeled using "Plotkin rules" (see Section 6)
– *Blocks* includes Algol-like blocks and procedures; it is used to show how the key idea of an "environment" can be used to model sharing and the normal range of parameter passing mechanisms are discussed[3]
– *COOL* is a concurrent object-based language – this is where the rule form of description really pays off. The language is rich enough to explore many alternatives.

With each language, there are lecture slots where alternatives suggested by the students are modeled. This gives a feel for real modeling rather than it being a static pre-canned set of examples.

The natural division of discussing syntax and semantics (and the difficult to place issue of context dependencies) is used. Before addressing the semantics of a language, it is necessary to delimit the language to be described. A traditional concrete syntax defines the strings of a language and suggests a parsing of any valid string.[4] Most texts on semantics are content to write semantic rules in terms of concrete syntax. Although this is convenient for small definitions, it really does not scale up to larger languages. We therefore base everything on *Abstract Syntax* descriptions (see Appendix A.1). For example, VDM defines objects like

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

which gives rise to a constructor function yielding tagged values

$$mk\text{-}Assn\colon Id \times Expr \to Assn$$

Using abstract syntax has the advantage of immediately getting the students to think about the information content of a program rather than bothering about the marks inserted just as parsing aids. There is an additional bonus that pattern matching with abstract objects gives a nice way of defining functions (or rules) by cases (see Appendix A.2 and A.3).

The class of *Program*s defined by any context free syntax (concrete or abstract) is too large in the sense that things like type constraints are not required to hold. There are many ways of describing *Context Conditions* but we prefer to write straightforward recursive predicates over abstract programs and static environments (see Appendix A.2) rather than, for example, use type theory as in [Pie02].

[3] We have fitted as much of this definition as space would allow into Appendix A.

[4] The publication of ALGOL-60 [BBG+63] solved the problem of documenting the syntax of programming languages: "(E)BNF" offers an adequate notation for defining the set of strings of a language. But is is a really sad observation that it no longer appears to be normal to include a concrete syntax in books on programming languages: very few books on Java have either a BNF nor a "Pascal tramline" syntax.

So, given a class of "well formed" abstract programs, how do we give the semantics? McCarthy's formal description of "micro-ALGOL" [McC66] defines an "abstract interpreter" which takes a *Program* and a starting state and delivers the final state. (In the simplest case the states (Σ) are mappings $Id \xrightarrow{m} Value$.) Thus:

$$Program \times \Sigma \to \Sigma$$

is defined by recursive functions over statements and expressions

$$exec\colon Stmt \times \Sigma \to \Sigma \text{ and } eval\colon Expr \times \Sigma \to Value$$

5 An example modeling idea

The course focuses on general modeling ideas like the use of "environments" to model sharing. In languages of the ALGOL family, the nesting of blocks and procedures introduces different scopes for identifiers permitting the same identifier to refer to different variables. Furthermore the ability to pass arguments "by location" ("by reference") makes it possible for different identifiers to refer to the same location. This gave rise to the idea of splitting the map

$$Id \xrightarrow{m} Value$$

into two with an abstract set of *Loc*ations representing the equivalences over identifiers, thus

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

$$\Sigma = Loc \xrightarrow{m} Value$$

The separation of such an environment from the state is an important aid to making properties of a language definition obvious:

$$Stmt \times Env \times \Sigma \to \Sigma$$

says more than

$$Stmt \times (Env \times \Sigma) \to (Env \times \Sigma)$$

This issue has been referred to as "small state vs. grand state". It is probable that one of the main attractions of denotational semantics was that it encouraged "small state" definitions.

The concept of environments and the abstract set of locations make it delightfully easy to illustrate the distinctions between different parameter passing modes ("call by value", "call by reference", "call by value/return", "call by name"). Furthermore, careful modeling of –say– *ArrayLoc* and *StructLoc* can result in a collection of semantic objects which convey a lot of information about a language without even looking at the semantic rules. The semantics given in Appendix A.3 uses "call by reference".

6 Plotkin rules

Non-determinism arises in many ways in programming languages. Certainly the most interesting cause is concurrency but it is also possible to illustrate via non-deterministic constructs like Dijkstra's "guarded commands". Unfortunately, McCarthy's idea to present an abstract interpreter by recursive functions does not easily cope with non-determinacy. A move to producing a set of states, as in *exec*: $Stmt \times \Sigma \to \mathcal{P}(\Sigma)$ is not convenient because of the need to ramify the combinations as follows:

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

$$\textit{exec-sl}(sl, \sigma) \triangleq$$
cases *sl*:
$[\,]$: $\{\sigma\}$
$[s] \curvearrowright \textit{rest}$: $\bigcup\{\textit{exec-sl}(\textit{rest}, \sigma') \mid \sigma' \in \textit{exec}(s, \sigma)\}$

because

$$\textit{exec} : \textit{Stmt} \times \Sigma \to \mathcal{P}(\Sigma)$$

$$\textit{exec}(s, \sigma) \triangleq \ldots$$

An alternative way to mechanise would be

$$\textit{poss-sl} : \textit{Stmt}^* \times \Sigma \times \Sigma \to \mathbb{B}$$

$$\textit{poss-sl}(sl, \sigma, \sigma_r) \triangleq$$
cases *sl*:
$[\,]$: $\sigma_r = \sigma$
$[s] \curvearrowright \textit{rest}$: $\exists \sigma_i \in \Sigma \cdot \textit{poss}(s, \sigma, \sigma_i) \wedge \textit{poss-sl}(\textit{rest}, \sigma_i, \sigma_r)$

with

$$\textit{poss} : \textit{Stmt} \times \Sigma \times \Sigma \to \mathbb{B}$$

$$\textit{poss}(s, \sigma, \sigma_r) \triangleq \ldots$$

But this moves in the direction of populating a logical frame — which topic is discussed in Section 10.

In 1981, Gordon Plotkin produced the technical report [Plo81] on "Structural Operational Semantics".[5] This widely photo-copied contribution revived interest in operational semantics. It can be argued that the most important contribution of [Plo81] was the step to using a "rule notation"; for example we can define a "guarded iteration":

GuardedIter :: *GuardedClause*-**set**

[5] This material, together with a companion note on its origins [Plo03a], has finally been published in a journal [Plo03b].

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

An example (concrete) program fragment might be:

$$x := 1; b := \mathbf{true};\ \mathbf{do}\ b \mapsto x := x + 1 \ ||\ b \mapsto b := \mathbf{false}\ \mathbf{od}$$

The guarded statement is non-deterministic in that either guarded clause can be selected when b is **true**.[6] The semantics are given as follows:

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

$$\frac{\begin{array}{l} \mathit{mk\text{-}GuardedClause}(\mathit{test}, \mathit{action}) \in \mathit{gcs} \\ (\mathit{test}, \sigma) \xrightarrow{e} \mathbf{true} \\ (\mathit{action}, \sigma) \xrightarrow{sl} \sigma' \\ (\mathit{mk\text{-}GuardedIter}(\mathit{gcs}), \sigma') \xrightarrow{s} \sigma'' \end{array}}{(\mathit{mk\text{-}GuardedIter}(\mathit{gcs}), \sigma) \xrightarrow{s} \sigma''}$$

$$\frac{\mathit{mk\text{-}GuardedClause}(\mathit{test}, \mathit{action}) \in \mathit{gcs} \ \Rightarrow\ (\mathit{test}, \sigma) \xrightarrow{e} \mathbf{false}}{(\mathit{mk\text{-}GuardedIter}(\mathit{gcs}), \sigma) \xrightarrow{s} \sigma}$$

All that the SOS rules provide are patterns: if the antecedents hold on a given system, then the consequent can follow. As noted about non-determinism above, when the antecedents of multiple rules hold, the choice regarding which one to execute is generally unconstrained.

The advantage of the move to such a rule presentation is the natural way of presenting non-determinacy. Many features of programming languages give rise to non-determinacy in the sense that more than one state can result from a given (program and) starting state. This natural expression extends well to concurrent languages. The advantage of the rule format appears to be that the non-determinacy has been factored out to a "meta-level" at which the choice of order of rule application has been separated from the link between text and states. For this reason, the complications of writing a function which directly defines the set of possible final states are avoided. Here is a case where the notation used to express the concept of relations (on states) is crucial.

Notice that the semantics of *Block* is non-deterministic in that *newlocs* is an underdetermined function. This is important as a way of showing that a compiler writer is allowed to re-use locations in a stack. It is also an interesting property that, in a language without a heap, the non-determinism is eliminated at the end of a block (i.e. a programmer cannot write a program whose result is influenced by the choice of locations).

One of the properties of SOS is that any given rule covers a small, simple concept. For things like assignment –a very simple concept in itself– only one rule is required. Conditional looping constructs, such as while, generally require two: one where the condition is true, one where it is false.

[6] As an example of the limits of ambition in CSC334, we would not (unless pressed) go into issues of "fairness" or transfinite induction.

7 Coping with exceptions

To describe more complex constructs requires that you decompose the concept by examining what it does. A simple exception handler, for example, might look something like s_1 **except** s_2 in a some given language. If we look at how such a construct is processed, we immediately see that there are two cases: one where the processing of s_1 proceeds normally, and one where s_1 causes some error so that s_2 must be evaluated.

This suggests that the statement transition should be modified to:

$$\xrightarrow{s}: Stmt \times \Sigma \to \Sigma \times \left[Abn\right]$$

where $\left[Abn\right]$ is essentially a flag to indicate that an error has occurred (**nil** indicating a normal result and **error** indicating an error). The rule to describe normal processing is then:

$$\frac{(s_1, \sigma) \xrightarrow{s} (\sigma', \mathbf{nil})}{(s_1 \ \mathbf{except} \ s_2, \sigma) \xrightarrow{s} (\sigma', \mathbf{nil})}$$

which simply says that the result of processing s_1 **except** s_2 is the result of s_1 if s_1 does not produce an error.

The complementary rule, when the processing of s_1 does produce an error, requires a little more care. If we decide that the handler, s_2 should see the state resulting from the (failed) processing of s_1, and that if s_2 produces an error that it should be passed on, then we might have a rule like:

$$\frac{(s_1, \sigma) \xrightarrow{s} (\sigma', \mathbf{error}) \quad (s_2, \sigma') \xrightarrow{s} (\sigma'', \alpha)}{(s_1 \ \mathbf{except} \ s_2, \sigma) \xrightarrow{s} (\sigma'', \alpha)}$$

This rule then says that the result of processing s_1 **except** s_2, if s_1 produces an error is the result of processing s_2 in the modified state.

This particular example provides what is a surprisingly close approximation of the usual try... catch... constructs you see in common languages like Java, C++, Python, and so on. It is missing the choice of exception handlers based on the type of exception, but that is not difficult to add.

The specific details of how an implementation might notice that processing caused an error is entirely left out of the rules. That is not a necessary part of this model, and is usually not part of a programmer's model when they are debugging code (unless, of course, the programmer is debugging the exception handling mechanism itself).

The two first year computer science courses at Ryerson did, at one point, use the Scheme programming language, and part of the syllabus involved teaching "the environment model of evaluation" essentially as it is found in [ASS85]. The model was taught through classroom examples and practical assignments.

That model, being concerned with how the language is evaluated, gives a completely operational definition of the language. It was not presented to the students as Plotkin-style rules, of course, but it did give them a fundamentally operational model with which to reason about the language. Most importantly,

those students were using an operational model to design and debug their programs.

8 Tool support

A key question for teaching CSC334 has been that of tool support. Indeed, this is an innovation by the second author into a course he experienced (from the third author) without any tool support. The addition of tool support to the module has the potential of both affording the students additional understanding and introducing an extra element of confusion. A student can for example create example programs which are objects of the abstract syntax and execute them using the semantic rules they have created. However, the student must learn how to use the tool and be aware differences between the mathematical VDM syntax taught in class and that used by the tool. For these reasons, the choice of whether to use the tool support is left to the individual choice of the students and teaching support is provided for both.

The tool used is the IFAD VDMTools® [IFA01b,IFA01a]. Many of the CSC334 students have experience of the tool from other modules. It provides an environment in which a VDM specification may be syntax and type checked and explicit functions may be executed via an interpreter. The students are provided with language specifications translated into ASCII VDM-SL. The semantic rules are translated into functions so that they can be executed in the Toolbox interpreter. This translation can in some cases produce functions that are significantly different to their equivalent semantic rules that are taught in class. This can put many students off using the tools, especially those who are not familiar with the ASCII syntax.

As an example of this process we will consider the semantic rule for the execution of a block:

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

Where *genlocs* and *b-FunDen* are auxiliary functions. *genlocs* generates a new environment *varenv* and state σ' which collectively contain new locations for any variables that are declared in the block. *b-FunDen* generates the semantic objects necessary to execute a function call. Both these functions are presented in the appendix.

The equivalent function as used with the IFAD VDMTools® would be as follows:

```
exec: Stmt * Env * State -> State
exec(s,env,sigma) ==
  cases s:
```

```
mk_Block(vars,funs,body) ->
  let mk_(varenv,sigma') = genlocs(vars,sigma) in
  let funenv =
    {f |-> b_FunDen(funs(f),env ++ varenv)|f in set dom funs}
  in
  let env' = env ++ varenv ++ funenv in
  let sigma'' = exec_sl(body,env',sigma') in
  (dom sigma) <: sigma''
end;
```

The above only covers the portion of exec that deals with the execution of blocks. The full definition contains separate cases for each type of statement; a call to exec made within the IFAD VDMTools® interpreter will execute the semantics of the relevant statement as defined by the semantic rules.

Beyond the syntactic differences between classical VDM-SL and that used by the tool, there are often cases where the appearance of a semantic rule can differ wildly with that of its functional ASCII equivalent. This is most evident when translating an implicit definition. For example the definition of *genlocs* supplied to the students (see the appendix) is highly implicit in nature, specifying the properties the new environment and state must hold rather than how they are created. In order for a specification of genlocs to be executed within the IFAD VDMTools® interpreter, the function has to be redefined as an explicit function. The process whereby this is accomplished is beyond the scope of this paper and often results in a large, ugly and confusing specification. Furthermore it is of no practical benefit for the students to study and comprehend these explicit definitions; it is important that they focus on the meaning of the semantics and disregard the implementation issues. For these reasons the students are shielded from much of the underlying explicit implementation by separating it from the main language specification through the use of mechanisms made available by the tool.

Quite often the main hurdle that a student must overcome when using the tool is the differentiation between the syntax and semantics of the meta-language and that of the language semantics they are building. Initially many find it difficult to distinguish between errors in the VDM syntax and errors in their semantics. Similarly a student will often be working under the misconception that the tool is nothing more than a compiler of example programs and attempt to tackle problems as they would a programming assignment or imagine they must define everything from scratch without identifying the relevant abstractions that a modeling language like VDM affords.

It is our belief that for some students at least, the benefits of using the tool outweigh the negatives. Through use of the tool, the students can:

- easily identify bugs in their VDM syntax
- spot type errors in their specifications
- execute test programs to improve their understanding
- detect semantic errors

Continuing on with the example of block execution as given above, the definition of such a rule has a number of pitfalls. For instance some students do not initially grasp the concept of separating the environment and the state and the purpose of locations. This leads to many misunderstandings when defining semantic rules for blocks and function calls. A common mistake is to confuse the two mapping relations and attempt to place values in the environment or ids in the state. In both cases the tool will flag these as VDM type errors.

By far the vast majority of mistakes made are errors in the semantic definitions. Here lies the major benefit of using the tool. The creation and execution of test programs not only highlights such semantic slips but also fosters a greater understanding by allowing students to see the consequences of their design decisions and perhaps identify some ramifications they had not previously considered. Using the example given above, it is not always clear to students why they must 'clean up' the state by removing the new locations as they leave. Should the old locations be allowed to remain, the consequences of this decision are not immediately obvious. However it becomes very apparent if the program is executed using the tool. The students can clearly see that the final program state has become 'polluted' with excess locations which are taking up large amounts of space.

Two methods are made available by the tool to aid in the analysis of the language specifications. The IFAD VDMTools® supplies a set of VDM libraries that provide simple I/O. These can be used to insert 'debugging' information into the specification, which can be used, for example to output the contents of the environment and state.

The second method involves using the debugging options made available by the tool. This involves placing break points in the specification and inspecting the function stack as the program execution is stepped through one function at a time.

The tool aids the students in the task of constructing example programs; they are forced to think of an abstract syntax program as a VDM object rather than a piece of concrete syntax. Similarly it helps them to distinguish between the dynamic semantics expressed in the rules and the static semantics expressed in the context conditions. The use of the tools debugging options introduces many complications for the students as they have to learn more aspects of the tool. However it does provide a good mechanism for demonstrating facets of the specification to groups of students. For example, when checking a program is well formed, the tool can show the hierarchical composition of the context conditions, clearly illustrated as each recursive call is placed onto the function stack.

Finally as an aside, the use of the tool as a teaching aid and the consequential inputting of the specification into the IFAD VDMTools® has helped identify and correct typos in our own specification. Thus shielding the students from unnecessary hardship.

9 Pedagogic experience

The CSC334 course is evaluated positively by the students who take it. As an optional module, they obviously tend to self select and about one third of the potential cohort choose to pursue it. The limited number makes it possible to adopt a reactive learning environment experimenting with ideas from the students.

The practical work of CSC334 is based heavily on problem solving. Threading through the semester is a large project to make non-trivial extensions to the *Block* definition, and the lecturer tries to keep things timed so that he is introducing concepts just before they are needed.

For the first couple of weeks –before the project is handed out– the students have the opportunity to gain familiarity with the toolset. After that they can choose either to use the tool support or to work on paper. Interestingly, the student numbers tend to split roughly in half on their choice, giving us two equally sized groups.

The format of the course's final exam stresses problem solving: it is an open-book exam, and relies on a language specification included from the lectures (this year *COOL*, in previous years *Block*).[7]

One of the course's final events is a partially structured specification walk-through. A few of the students have been chosen to study one of the specifications used in the lectures (the same specification as on the exam), and they will have the chance to grill the lecturer on the choices made in the design of that language. Part of the students' task is to collect questions and comments from their classmates as their role is partially representative, and the un-nominated students will also have to opportunity to ask questions through the session. One of the teaching assistants will be chairing the discussion (the other will be in Copenhagen :-).

During the practical sessions of the course the teaching assistants focused strongly on making sure that the students understood what information the abstract syntax provides, and what details it leaves out completely. This particular problem was somewhat more pronounced using the toolset simply because of its interactive nature. Taking assignment as an example, the students were apt to confuse the construct that models assignment with a command that causes assignment in the toolset syntax. There was the also the initial tendency for the students to attribute meaning to things like the ordering of fields in a construct of the abstract syntax.

Once the students had the notation clear in their minds, they were able to think about the language in terms of what the constructs offer, rather than in terms of what the parser recognises.

[7] Had the class size been smaller, there was the possibility of an alternative to the written exam: an informal (continental style) viva to test the students' practical knowledge was discussed. The justification for a viva-style examination lies in the strong emphasis in the course that the subject material should not be memorised, but rather taken as a method to be applied.

Another common problem exists as the students figure out where they should be putting various details of their specification: in the abstract syntax, in the context conditions, or in the rules.

At one point, one group was debating the specification of a multiple assignment construct. The exercise was given in such a way so that the students would have to define some semantics for it, then justify their choice. Typically, the first reaction of the students to the example syntax (`x,y:=y,x;`) was to equate that to a sequence of assignments (i.e. `x:=y;y:=x`). The more mischievous teaching assistant saw this as a good opportunity to point out to them that it could be equivalent to a value swap.

This particular exercise worked very well to drive home that the choice depends on what the designer wants to do with the language in question. Understanding that point is not always obvious for the students when they are used to a "right" way of writing programs.

The choice between using VDM-style functions or using Plotkin-style rules tends to give students some trouble. The project work is all done under the assumption that everything is deterministic and sequential (no concurrency). The first impression of many of the students is to ask why they need another form of notation (as VDM-style semantic functions are covered first).

The Plotkin-style rules became more popular with the students who choose to work on paper as two things became apparent. First, that the antecedent portion of a rule is used to pattern-match, rather than create the world. The difference is subtle: the students use to VDM-style functions to specify what to do, and initially try to use the Plotkin-style rules in the same way. As they realise that the Plotkin-style rules are used to describe the world we desire, they become far more comfortable using them (and a lot less concerned with how hardware-level details of the operation).

10 Alternatives/Future Directions

This section looks at what would have been called in IBM "outplan" items. In some cases they indicate directions in which we hope to move in the future.

There is, of course, much related material that could usefully be taught on semantics. Textbooks such as [NN92] and [Win93] provide excellent introductions to the basic notions of semantics, but –to our taste– do so without a practical context. Their concern with meta-properties of the language would motivate our students less well than experiments with modeling a range of programming language issues. In our case, students have had prior exposure to the VDM ideas of specification and development (a second year course based on [FL98]) and the link from Operational Semantics to the justification of proof rules for "operation decomposition" is made in a single late lecture in CSC334.

An interesting discussion is that on further tool support. One direction which would be relatively easy is to use ideas of hyperlink-like objects to build support into an editor to jump back and forth between various parts of a specification. For example, while looking at one particular rule in a specification, you see a

constructor, *mk-Assign*. It is easy to imagine being able to say, right-click on it and see a menu that allows you to jump to the construct definition, or present a list of rules and functions in which it is used. The general idea is to help the reader find the relevant bits of the specification without requiring that they have their fingers stuck into several sections of a book.

A much more ambitious tool support programme would be to mechanise the Plotkin rules in the way envisaged by Tom Melham [CM92] and executed by Tobias Nipkow and colleagues [KNvO$^+$02,Nip04] for Java.

Acknowledgments

All of the authors acknowledge the support of EPSRC's funding of the *Interdisciplinary Research Collaboration* on the *Dependability of Computer-Based Systems* (DIRC). In addition the second author is grateful to the EPSRC funded *Diversity with Off-The-Shelf* (DOTS) for providing his studentship and the first author to the University of Newcastle for his IRS award.

References

[ASS85] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and interpretation of computer programs*. MIT Press, 1985.

[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.

[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.

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

[IFA01a] IFAD. *VDMTools®: The IFAD VDM-SL Language*. http://www.ifad.dk, 2001.

[IFA01b] IFAD. *VDMTools®: VDM-SL Toolbox Manual*. http://www.ifad.dk, 2001.

[Jon99] C. B. Jones. Scientific decisions which characterise VDM. In *FM'99 – Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 28–47. Springer-Verlag, 1999.

[Jon01] C. B. Jones. The transition from VDL to VDM. *JUCS*, 7(8):631–640, 2001.

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

[Luc81] P. Lucas. Formal semantics of programming languages: VDL. *IBM Journal of Research and Development*, 25(5):549–561, September 1981.

[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.

[Nip04] Tobias Nipkow. Jinja: Towards a comprehensive formal semantics for a java-like language. Manuscript, Munich, 2004.

[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.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

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

[Plo03a] Gordon D. Plotkin. The origins of structural operational semantics. *Journal of Functional and Logic Programming*, 2003. forthcoming.

[Plo03b] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Functional and Logic Programming*, 2003. forthcoming.

[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.

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

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

[XRR+99] J. Xu, B. Randell, A. Romanovsky, R. J. Stroud, A. F. Zorzo, E. Canver, and F. von Henke. Rigorous development of a safety-critical system based on coordinated atomic actions. In *Proc. of 29th Int. Symp. Fault-Tolerant Computing*. IEEE Computer Society Press, 1999.

A An example definition

This appendix contains much of the definition of one version of *Blocks* as taught in CSC334. Here, the definition is presented in the order abstract syntax, context conditions, and Semantics; many of the language definitions printed in the forthcoming book order the definition by language construct. Apart from dropping *Expr*ession etc. for space reasons, the definition here avoids arrays and records to minimise length. (The parameter passing mode here is by location (aka by reference).)

A.1 Abstract Syntax

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

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

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

$$Stmt = Block \mid Assign \mid If \mid GuardedIter \mid Call$$

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

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

$$GuardedIter :: GuardedClause\text{-}\mathbf{set}$$

$$\begin{array}{lll} GuardedClause :: & test & : Expr \\ & action & : Stmt^* \end{array}$$

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

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

A.2 Context conditions

We first define some Auxiliary objects

$$Types = Id \xrightarrow{m} (ScalarType \mid FunType)$$

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

Then the Well formedness predicates are:

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

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

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

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

$$\begin{aligned} &wf\text{-}Stmt(mk\text{-}If(test, th, el), tps) \triangleq \\ &\quad tp(test, tps) = \textsc{BoolTp} \wedge \\ &\quad wf\text{-}StmtList(th, tps) \wedge wf\text{-}StmtList(el, tps) \end{aligned}$$

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

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

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

The auxiliary function tp is defined

$$tp : Expr \times Types \to (ScalarType \mid \textsc{Error}))$$

$$tp(e, tps) \triangleq \text{given by cases}$$

A.3 Semantics

We first define the Semantic objects

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

$$Den = ScalarLoc \mid FunDen$$

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

$$\begin{aligned} FunDen :: \ & parms : Id^* \\ & body : Stmt^* \\ & result : Expr \\ & context : Env \end{aligned}$$

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

Then the semantic rules are

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

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

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

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

The semantic transition relation for statement lists is

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

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

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

For Executing statements

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

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

$$\frac{\begin{array}{l}(\mathit{test}, \mathit{env}, \sigma) \xrightarrow{e} \mathbf{true}\\ (\mathit{th}, \mathit{env}, \sigma) \xrightarrow{sl} \sigma'\end{array}}{(\mathit{mk\text{-}If}(\mathit{test}, \mathit{th}, \mathit{el}), \mathit{env}, \sigma) \xrightarrow{s} \sigma'}$$

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

$$\frac{mk\text{-}GuardedClause(test, action) \in gcs \quad (test, \sigma) \xrightarrow{e} \textbf{true} \quad (action, \sigma) \xrightarrow{sl} \sigma' \quad (mk\text{-}GuardedIter(gcs), \sigma') \xrightarrow{s} \sigma''}{(mk\text{-}GuardedIter(gcs), \sigma) \xrightarrow{s} \sigma''}$$

$$\frac{mk\text{-}GuardedClause(test, action) \in gcs \;\Rightarrow\; (test, \sigma) \xrightarrow{e} \textbf{false}}{(mk\text{-}GuardedIter(gcs), \sigma) \xrightarrow{s} \sigma}$$

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

For evaluating expressions

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

Finally, for evaluating left-hand-side values

$$\xrightarrow{lhv} : (Id \times Env \times \Sigma) \times ScalarLoc$$

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

The abbreviations used include

$\sigma \in \Sigma$	a single "state"
Σ	the set of all "States"
Arith	Arithmetic
env	a single "environment"
Env	the set of all "Environments"
eval	evaluate
exec	execute
Expr	Expression
Fun	Function
opd	operand
Rel	Relational
Stmt	Statement