SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 94-50

A small language definition in Z

Ian Hayes

December 1994

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533

Note: Most SVRC technical reports are available via anonymous ftp, from `ftp.cs.uq.edu.au` in the directory `/pub/SVRC/techreports`.

A small language definition in Z

Ian Hayes

Abstract

We give give a denotational-style specification of a simple programming language in Z and comment on the suitability of Z for this purpose. The language specified is based closely on one specified by Cliff Jones in VDM (Chapter 9 of *Case Studies in Systematic Software Development*, ed. C.B. Jones and R.C.F. Shaw, Prentice Hall, 1990). Specifying the same language allows a comparison to be drawn between the suitability of VDM and Z for the specification of programming languages. Of special interest in the comparison are the use of free type definitions and lambda notation within the specification.

1 Introduction

Although Z has been widely used as a specification language, it has not been used extensively for the specification of languages. The primary objective of this paper is to provide an example of the specification of a simple programming language using Z. The language presented here is based closely on that defined in terms of the VDM specification language in [Jon90a].

The language specified is a simple imperative language with expressions (Section 2), commands (Section 4), block structure and procedures (Section 5). The use of the same language as that presented using VDM in [Jon90a] allows a comparison to be made between the two specification languages. Because most of the features to be compared are amply illustrated in the specification of expressions, we present the comparison in Section 3 immediately after expressions. Sections 3.1 and 3.2 compare the facilities for describing disjoint unions and Section 3.3 for defining constants. A further interesting point comes from the specification of assertions and their relationship to boolean expressions, so we treat that in Section 3.4. Section 3.5 compares the treatments of partial-function definitions.

Readers primarily interested in the example of specifying a simple language can skip the comparison (Section 3), while readers primarily interested in the comparison of VDM and Z can skip the sections on commands and procedures (Sections 4 and 5).

2 Expressions

We specify expressions in the language by first giving an abstract syntax for variable references and expressions (we ignore concrete syntax in this specification), next we give consistency constraints (sometimes called static semantics) for variable references and expressions, and finally we give the semantics of variable references and expressions.

2.1 Abstract syntax of variable references

We assume a set of identifiers *Id* is given.

$$[Id]$$

A variable reference includes the name (identifier) of the variable.

$$Varref ::= varref \langle\langle Id \rangle\rangle$$

Aside: The definition of *Varref* defines it as a disjoint union with only one alternative. It is defined this way to allow for extension to structured variables, eg, subscripted arrays, at a later stage. Note that, if $x \in Id$ then $x \notin Varref$, but $varref(x) \in Varref$. The inverse of a constructor may be used to extract its argument. For example, if $vr = varref(x)$, then the inverse of *varref*, namely $varref^{\sim}$, can be used to extract the identifier x from the variable reference: $varref^{\sim}(vr) = x$. □

2.2 Abstract syntax of expressions

An expression in the language may be either a boolean or an integer constant, a reference to a variable, or an infix expression involving addition, comparison (less than) or boolean disjunction. The abstract syntax for expressions is given in Figure 1.

Boolean and integer constants include the value of the constant. As a boolean type is not predefined in Z, we introduce the following definition.

$$\mathbb{B} ::= True \mid False$$

The infix binary operators for addition, less than and boolean disjunction require two subexpressions. *Expr* is defined to be a new type which is the disjoint union of four alternatives. The constructors *bool*, *int*, *exvar* and *infix* take an argument of the type enclosed in the double-angled brackets and return a unique value of type *Expr*. For example, assuming $x \in Id$ and $b \in Id$ are

$$
\begin{aligned}
Operator &::= plus \mid or \mid lessthan \\
Expr &::= bool\langle\!\langle \mathbb{B} \rangle\!\rangle \\
&\quad \mid int\langle\!\langle \mathbb{Z} \rangle\!\rangle \\
&\quad \mid exvar\langle\!\langle Varref \rangle\!\rangle \\
&\quad \mid infix\langle\!\langle Expr \times Operator \times Expr \rangle\!\rangle
\end{aligned}
$$

Figure 1: Abstract syntax for expressions (in Z)

distinct identifiers,[1] the following are elements of *Expr*:

$$
\begin{aligned}
&tt == bool(True) \\
&zero == int(0) \\
&one == int(1) \\
&varx == exvar(varref(x)) \\
&varb == exvar(varref(b)) \\
&xplus1 == infix(varx, plus, one) \\
&compar == infix(xplus1, lessthan, zero) \\
&disj == infix(compar, or, varb)
\end{aligned}
$$

The constructors for a disjoint union satisfy the following properties:

- Each constructor is a total, one-to-one function, so that for different values of their argument they give different values of type *Expr*.
- The ranges of the constructors are disjoint, so that any value constructed with one constructor is distinct from a value constructed with another constructor.
- The union of the ranges of the constructors is the whole type *Expr*, so that the only values in the type *Expr* are those that can be built using the constructors.
- Any value of type *Expr* can be built from values of other types by a finite number of applications of the constructor functions for *Expr*.

2.3 Consistency constraints for variable references

Not all expressions allowed by the abstract syntax are well formed. Well-formed expressions should be type correct. For example, the operands of the addition operator should both be of type integer.

[1] x and b are Z variables with values of type Id. Hence both x and b could have the same (Id) value.

Only the scalar types boolean and integer are available in the language. We introduce *Sctype* to represent the type of variables and expressions.

$$Sctype ::= int_type \mid bool_type.$$

In order to determine whether an expression is type correct, we need to know the type of every variable referenced by the expression. To achieve this we introduce an environment of type *Senv*, which is a mapping from identifiers to the attributes of the identifiers. The attribute of an identifier declared as a variable both indicates that it is a variable (rather than a procedure) and gives its type. An attribute is defined as a disjoint union with only one alternative to allow for extension later on to cope with procedures.

$$Attr ::= vartype\langle\langle Sctype\rangle\rangle$$
$$Senv == Id \twoheadrightarrow\!\!\!\!\!\!+ \; Attr$$

We follow the convention of using the variable named ρ as an environment. For example, the environment ρ which declares the identifier x to be a variable of type integer and the identifier b to be a variable of type boolean, is the following,

$$\rho == \{x \mapsto vartype(int_type), b \mapsto vartype(bool_type)\}.$$

An environment is a function; hence, given ρ above, we can apply it to any of the identifiers in its domain ($\{x, b\}$):

$$\rho(x) = vartype(int_type)$$
$$\rho(b) = vartype(bool_type).$$

A variable reference is well formed in all environments in which the name of the variable is defined to be a variable. To represent this we define a function *WFVarref*, which gives the set of environments in which a variable reference is well formed.

$$WFVarref : Varref \longrightarrow (\mathbb{P}\, Senv)$$
$$WFVarref(varref(id)) = \{\rho : Senv \mid id \in \mathrm{dom}\,\rho \wedge \rho(id) \in \mathrm{ran}\, vartype\}$$

To determine whether an identifier defined in the environment ρ (ie, in the domain of ρ) is a variable, we test whether its associated attribute is in the range of *vartype*.

For a well-formed reference to a variable in an environment, we can determine the type of the variable by looking up the environment. We define a function *TPVarref*, which given a variable reference, gives a function from environments in which that variable reference is well formed to the type of the variable in that environment.

$$TPVarref : Varref \longrightarrow (Senv \mathrel{+\!\!\!\rightarrow} Sctype)$$
$$TPVarref(varref(id)) = \{\rho : WFVarref(varref(id)); ty : Sctype \mid \rho(id) = vartype(ty) \bullet \rho \mapsto ty\}$$

Aside: An alternative definition of *TPVarref* using lambda (λ) notation is as follows.

$$TPVarref(varref(id)) = (\lambda\, \rho : WFVarref(varref(id)) \bullet vartype^{\sim}(\rho(id)))$$

This makes use of the inverse of the constructor *vartype* to extract the type of a variable. The lambda notation defines a function which takes an argument, ρ, which must be an environment in the set of all environments in which the variable reference is well formed, and extracts the type of the variable from the environment. □

2.4 Consistency constraints for expressions

We define a function *WFExpr* which gives the set of environments for which an element of *Expr* is well formed. In addition, we define functions giving the environments in which elements of *Expr* are well formed and of type integer (*WFIntExpr*), and well formed and of type boolean (*WFBoolExpr*). We also define the function *TPExpr*, which given an expression, gives a function mapping an environment in which that expression is well formed to the type of the expression in the environment. The consistency constraints for expressions are given in Figure 2.

Boolean and integer constants are well formed in all possible environments; they are of type *bool_type* and *int_type*, respectively. An expression consisting of a variable reference is well formed if the variable reference is well formed; its type is the type of the variable in the environment. The sum of two expressions is well formed in an environment in which both operand expressions are well-formed expressions of type integer; the type of the sum is integer. The comparison of two expressions is well formed in the same environments, but its type is boolean. The disjunction of two expressions is well formed provided the two operand expressions are well-formed expressions of type boolean; the result of the disjunction is of type boolean.

2.5 Semantics of variable references

Having defined the (abstract) syntactic form of expressions and the consistency constraints on expressions, we can define the meaning (semantics) of well-formed expressions. This amounts to defining the value of an expression given the values of the variables used in the expression.

The value of an expression may be either a boolean or integer value. We represent the set of such scalar values as a disjoint union of boolean values and integer values.

$$Scval ::= bool_val\langle\!\langle \mathbb{B} \rangle\!\rangle \mid int_val\langle\!\langle \mathbb{Z} \rangle\!\rangle$$

To be able to define the meaning (value) of an expression, we need to know the values of all the variables used in the expression. To represent this we introduce a store which contains the values of the variables. In order to cope

$$
\begin{array}{l}
WFExpr, WFIntExpr, WFBoolExpr : Expr \longrightarrow (\mathbb{P}\, Senv) \\
TPExpr : Expr \longrightarrow (Senv \nrightarrow Sctype) \\
\hline
WFExpr(bool(b)) = Senv \\
WFExpr(int(n)) = Senv \\
WFExpr(exvar(vr)) = WFVarref(vr) \\
WFExpr(infix(e1, plus, e2)) = WFIntExpr(e1) \cap WFIntExpr(e2) \\
WFExpr(infix(e1, lessthan, e2)) = WFIntExpr(e1) \cap WFIntExpr(e2) \\
WFExpr(infix(e1, or, e2)) = WFBoolExpr(e1) \cap WFBoolExpr(e2) \\
WFIntExpr(ex) = \{\rho : WFExpr(ex) \mid TPExpr(ex)(\rho) = int_type\} \\
WFBoolExpr(ex) = \{\rho : WFExpr(ex) \mid TPExpr(ex)(\rho) = bool_type\} \\
TPExpr(ex) = \{\rho : WFExpr(ex); type : Sctype \mid \\
\quad (\exists\, b : \mathbb{B} \bullet ex = bool(b) \land type = bool_type) \lor \\
\quad (\exists\, n : \mathbb{Z} \bullet ex = int(n) \land type = int_type) \lor \\
\quad (\exists\, vr : Varref \bullet ex = exvar(vr) \land type = TPVarref(vr)(\rho)) \lor \\
\quad (\exists\, e1, e2 : Expr \bullet \\
\qquad ex = infix(e1, plus, e2) \land type = int_type \lor \\
\qquad ex = infix(e1, lessthan, e2) \land type = bool_type \lor \\
\qquad ex = infix(e1, or, e2) \land type = bool_type) \bullet \\
\quad \rho \mapsto type\}
\end{array}
$$

Figure 2: Consistency constraints for expressions

with variable parameters to procedures later on, we need the concept of the location of a variable within the store. We introduce a set of locations *Scloc* on which we define no further structure, other than to insist it be large enough for our needs.

$$[Scloc]$$

A store is a mapping from locations to scalar values.

$$Store == Scloc \pfun Scval$$

We make use of the name σ for stores. For example, a store σ in which location $l1$ has the integer value 3 and location $l2$ has the boolean value *True* is represented by the following mapping,

$$\sigma == \{l1 \mapsto int_val(3), l2 \mapsto bool_val(True)\}.$$

As σ is a function from (scalar) locations to (scalar) values we may apply it to a location in its domain to get the corresponding value. For example,

$$\sigma(l1) = int_val(3)$$
$$\sigma(l2) = bool_val(True).$$

We define a function which extracts the *contents* of a location in a store.

$$contents : Scloc \longrightarrow (Store \pfun Scval)$$
$$contents(loc) = (\lambda\, \sigma : Store \mid loc \in \operatorname{dom} \sigma \bullet \sigma(loc))$$

In addition to a store, we need an environment that gives the location of each variable. Given a variable name x, we look up the environment to find its location, say $l1$, and then we can look up the value stored at that location. We introduce the type *Val* of values to be stored in such environments.[2] It is defined as a disjoint union with only one alternative to allow for later extension to cope with procedures.

$$Val ::= varloc \langle\!\langle Scloc \rangle\!\rangle$$
$$Env == Id \pfun Val$$

For example, an environment that defines variable x to be at location $l1$ and variable b to be at location $l2$ is given by

$$\rho = \{x \mapsto varloc(l1), b \mapsto varloc(l2)\}.$$

Note that the (dynamic) environments being used in this section are different to the (static) environments used in the previous section to give the types of variables.

The meaning associated with a variable reference is a function that, given an environment in which the variable is defined, gives the location of the variable.

$$MVarref : Varref \longrightarrow (Env \pfun Scloc)$$
$$MVarref(varref(id)) = \{\rho : Env; loc : Scloc \mid id \in \operatorname{dom} \rho \land \rho(id) = varloc(loc) \bullet \rho \mapsto loc\}$$

[2] The name Val is not well chosen but is used to be consistent with [Jon90a].

2.6 Semantics of expressions

The meaning of an expression is a function that given an environment (defining the locations of the variables used in the expression) gives a function that given a store returns a value. That is, for an expression ex, environment ρ and store σ,

$$MExpr(ex)(\rho)(\sigma),$$

is the (scalar) value of the expression ex using the environment ρ to determine the location of any variables in ex, and using the store σ to determine the value of the variables, given their locations.

The meanings of expressions are defined in Figure 4. The meaning of a boolean or integer constant is the value of the constant. Its value is independent of the environment and the store. The meaning of a variable reference is the contents of the store at the location defined for the variable in the environment. In Figure 3 we introduce the auxiliary function *MOperator* that is used to define the binary operators acting on our representation of scalar values.

$$MOperator : Operator \longrightarrow (Scval \times Scval \mathrel{+\!\!\!\rightarrow} Scval)$$

$$\begin{aligned}
&MOperator(plus) \in \operatorname{ran} int_val \times \operatorname{ran} int_val \longrightarrow \operatorname{ran} int_val \\
&MOperator(lessthan) \in \operatorname{ran} int_val \times \operatorname{ran} int_val \longrightarrow \operatorname{ran} bool_val \\
&MOperator(or) \in \operatorname{ran} bool_val \times \operatorname{ran} bool_val \longrightarrow \operatorname{ran} bool_val \\
&\\
&MOperator(plus)(int_val(n1), int_val(n2)) = int_val(n1 + n2) \\
&MOperator(lessthan)(int_val(n1), int_val(n2)) = \\
&\qquad\qquad bool_val(True) \Leftrightarrow (n1 < n2) \\
&MOperator(or)(bool_val(b1), bool_val(b2)) = bool_val(True) \\
&\qquad\qquad \Leftrightarrow (b1 = True \lor b2 = True)
\end{aligned}$$

Figure 3: Meaning of operators (in Z)

The function *MOperator* is only defined for pairs of scalar values (elements of *Scval*) that have correct type for the given operator. The set of all scalar values representing integers is given by the range of the constructor int_val. The value of an expression consisting of the sum of two subexpressions, is the sum (using the auxiliary function *MOperator*) of the values of the subexpressions evaluated in the same environment and store. The other binary operators are defined similarly.

$$
\begin{array}{|l}
MExpr : Expr \longrightarrow (Env \mathrel{+\!\!\!\rightarrow} (Store \mathrel{+\!\!\!\rightarrow} Scval)) \\
\hline
MExpr(bool(b)) = (\lambda\, \rho : Env \bullet (\lambda\, \sigma : Store \bullet bool_val(b))) \\
MExpr(int(n)) = (\lambda\, \rho : Env \bullet (\lambda\, \sigma : Store \bullet int_val(n))) \\
MExpr(exvar(vr)) = (\lambda\, \rho : \mathrm{dom}(MVarref\ vr) \bullet \\
\qquad contents(MVarref(vr)(\rho))) \\
MExpr(infix(e1, op, e2)) = (\lambda\, \rho : \mathrm{dom}(MExpr\ e1) \cap \mathrm{dom}(MExpr\ e2) \bullet \\
\qquad (\lambda\, \sigma : \mathrm{dom}(MExpr(e1)(\rho)) \cap \mathrm{dom}(MExpr(e2)(\rho)) \mid \\
\qquad (MExpr(e1)(\rho)(\sigma), MExpr(e2)(\rho)(\sigma)) \in \mathrm{dom}(MOperator\ op) \bullet \\
\qquad\quad MOperator(op)(MExpr(e1)(\rho)(\sigma), MExpr(e2)(\rho)(\sigma))))
\end{array}
$$

Figure 4: Meanings of expressions (in Z)

3 VDM and Z compared

Before proceeding with the description of commands and procedures in the simple programming language, we take stock of the differences between VDM and Z as illustrated by their use in the specification of expressions. One noticable difference is the approach to specifying disjoint unions, as used, for example, in specifying the abstract syntax for expressions (Section 3.1). Closely related points are the use of mutual recursion in free type definitions (Section 3.2), the treatment of constants in the specification (Section 3.3) and the extension of free types (Section 3.4). The treatment of partial functions, in particular in lambda notation, is also worthy of discussion (Section 3.5).

3.1 Disjoint unions

The first interesting point of comparison between VDM and Z is the way they handle disjoint unions. In VDM, one may create new types using *composite* types [Jon90b, Chapter 5]. Every new type created in this way is disjoint from all other such types. However, in VDM one can form the union of any collection of sets, including those of different types. Hence, it is possible to form the union of a number of disjoint sets. Given an element of such a union it is possible to determine which alternative of the union it belongs to by a simple membership test. For example, Figure 5 defines, in VDM, the abstract syntax of expressions in the simple programming language as given in [Jon90a, page 238].

New types *Infix*, *Rhsref* and *Varref* are defined as composite types: *Infix* is composed of a left and right subexpression plus an operator, and *Rhsref* and *Varref* are just defined in terms of *Varref* and *Id*, respectively. A composite type

$$
\begin{array}{l}
Expr = Infix \cup Rhsref \cup Const \\
Infix :: \quad l : Expr \\
\qquad\qquad op : Operator \\
\qquad\qquad r : Expr \\
Operator = \{\textsc{Plus}, \textsc{Or}, \textsc{Lessthan}\} \\
Rhsref :: Varref \\
Const = Scval \\
Varref :: Id \\
Scval = \mathbb{B} \cup \mathbb{Z}
\end{array}
$$

Figure 5: Expression abstract syntax (in VDM)

definition, such as *Infix*, introduces a new type, called *Infix*, and a constructor, called *mk-Infix*, which is a total, onto, one-to-one function (a bijection)

$$mk\text{-}Infix : Expr \times Operator \times Expr \rightarrowtail\!\!\!\!\!\twoheadrightarrow Infix.$$

The definitions of *Rhsref* and *Varref* are degenerate forms of composite type definitions. They introduce new types with elements constructed from elements of an existing set. For example, *Varref* is a new type with a constructor,

$$mk\text{-}Varref : Id \rightarrowtail\!\!\!\!\!\twoheadrightarrow Varref.$$

For every element $id : Id$, $mk\text{-}Varref(id)$ is a distinct element of *Varref*. (Note that *id* is not an element of *Varref*.)

In Figure 5 *Expr* is defined to be the union of *Infix*, *Rhsref* and *Const*. As these three sets are disjoint there can be no confusion between their elements. Both *Infix* and *Rhsref* have been defined as new types via composite type introduction, but *Const* has been defined as equal to *Scval*. *Const* is not a new type; it is indistinguishable from *Scval* — there is a significant difference between using a '::' and an '=' in a definition. *Scval*, and hence *Const*, is defined as the union of the booleans and the integers. As the booleans and integers are distinct types this avoids any confusion. The definition of *Expr* in Figure 5 is exactly equivalent to

$$Expr = Infix \cup Rhsref \cup \mathbb{B} \cup \mathbb{Z}.$$

In Z, set union ($\cup$) is only defined between sets of the same type. To form a disjoint union from sets of different types one makes use of Z's *free type* notation [Spi92, page 82]. A new type is formed and a number of constructors, one for each alternative in the disjoint union, are defined to allow one to create elements

of the new type and to determine to which alternative elements of the new type belong.

Figure 1 gives the abstract syntax for expressions in Z. It is difficult to match the VDM and Z definitions one-to-one, because VDM associates new types and constructors with composite types, whereas Z associates them with disjoint unions. In the case of *Varref*, only a new type is being introduced, so there is little difference between the VDM and Z versions. However, for the definition of *Expr* in Z, the constructors are associated with the alternatives of the disjoint union, whereas in VDM they are associated with the individual types of the union.

It is possible to combine the *bool* and *int* branches in the Z version as follows.

$$\begin{aligned} Expr2 ::= &\ const\langle\langle Scval\rangle\rangle \\ \mid &\ \ldots \end{aligned}$$

However, this introduces a new constructor *const*, unlike the VDM version. The Z version becomes inconvenient for later use because, for example, the integer constant five has to be represented as '$const(int(5))$' rather than just '$int(5)$', or in VDM, just '5'. The definition of *MOperator* given in Figure 3 would be more complicated with additional references to the constructor *const*. This should be compared with the somewhat simpler VDM version of *MOperator* in Figure 6. The VDM version could be further simplified because boolean types are

$$MOperator : Operator \longrightarrow (Scval \times Scval \nrightarrow Scval)$$

$$\begin{aligned}
&MOperator(plus) \in \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{Z} \\
&MOperator(lessthan) \in \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{B} \\
&MOperator(or) \in \mathbb{B} \times \mathbb{B} \longrightarrow \mathbb{B} \\[1ex]
&MOperator(plus)(n1, n2) = n1 + n2 \\
&MOperator(lessthan)(n1, n2) = True \Leftrightarrow (n1 < n2) \\
&MOperator(or)(b1, b2) = True \Leftrightarrow (b1 = True \vee b2 = True)
\end{aligned}$$

Figure 6: Meaning of operators (in VDM)

predefined within the language, but that is not our immediate concern here.

The additional layers of Z constructors get in the way of simple expression of the desired specification. This is further highlighted in Section 3.4 where boolean expressions are used as just one alternative in the definition of assertions. One advantage of the Z approach is that the typing mechanism is stronger and hence it may be possible to detect errors with a static type checker that go undetected in the corresponding VDM specification.

3.2 Using schemas in free types

Another difference between the VDM in Figure 5 and the Z in Figure 1 is in the definition of *Infix*. In Z it is given via a Cartesian product while VDM uses a composite object with fields. To try to match the VDM definition more closely we could make use of a Z schema to define *Infix*:

$$Operator ::= plus \mid or \mid lessthan$$

```
┌─ Infix ─────────────────────────
│ l : Expr
│ op : Operator
│ r : Expr
└─────────────────────────────────
```

$$\begin{aligned} Expr ::= {} & bool \langle\!\langle \mathbb{B} \rangle\!\rangle \\ \mid {} & int \langle\!\langle \mathbb{Z} \rangle\!\rangle \\ \mid {} & exvar \langle\!\langle Varref \rangle\!\rangle \\ \mid {} & infix \langle\!\langle Infix \rangle\!\rangle \end{aligned}$$

Unfortunately that leads to a mutually recursive definition which is not allowed in [Spi92]. There is nothing fundamentally wrong with the above mutually recursive definition although in general one needs to be careful about the scope of such mutual recursion if additional predicates are allowed to constrain the sets being defined (that is not the case here).

3.3 Constants

Another closely related area of difference between VDM and Z is in the introduction of constants. In VDM, constants are kept distinct from variables. They are differentiated in the concrete syntax by being in a different font from variables. For example, in Figure 5 *Operator* is defined in terms of the three constants, PLUS, OR and LESSTHAN. All constants in VDM are distinct and furthermore they are not elements of any new types formed via the *composite* type mechanism.

Strictly speaking, in Z there are no 'constant' identifiers. The Z definition of *Operator* in Figure 1 introduces a new type, *Operator*, and three variables: *plus*, *or* and *lessthan*. The free type definition ensures that these variables have distinct values of type *Operator* and, as *Operator* is a fresh type, these values are distinct from values in all other types.

Overall these two mechanisms for introducing 'constants' achieve the same effect. One advantage of the VDM approach is that it is easier to form a new type which is an extension of *Operator* with some additional constants. In Z, a new free type needs to be introduced with the old type *Operator* used in the constructor for one of the branches and 'constants' used in the other branches.

3.4 Extending free types

Before leaving disjoint unions it is worthwhile to examine an additional difficulty with the extension of free types.

In dealing with the axiomatic semantics of a language we need to introduce assertions about the state of a program. Assertions are similar to boolean expressions within the language, but are typically more expressive, for example, they may allow universal and existential quantification. Figure 7 shows an example abstract syntax for assertions. An assertion may be a boolean expression, a conjunction, a disjunction, or a universal quantification. There are two points

$$
\begin{array}{lll}
\mathit{Operator_as} & ::= & \mathit{or_as} \mid \mathit{and_as} \\
\mathit{Assertion} & ::= & \mathit{bool_expr}\langle\!\langle \mathit{Expr} \rangle\!\rangle \\
 & \mid & \mathit{infix_as}\langle\!\langle \mathit{Assertion} \times \mathit{Operator_as} \times \mathit{Assertion} \rangle\!\rangle \\
 & \mid & \mathit{forall}\langle\!\langle \mathit{Id} \times \mathit{Assertion} \rangle\!\rangle
\end{array}
$$

Figure 7: Abstract syntax of assertions (in Z)

to note about the definition. Firstly, the *Expr* for the *bool_expr* alternative should be a boolean expression and not an integer expression; we can take care of this via a consistency constraint. Secondly, although a boolean expression has an alternative, *or*, which represents a logical disjunction, we need to introduce a separate alternative for logical disjunction at the assertion level because the disjuncts at the assertion level are assertions (perhaps including quantifications) rather than boolean expressions. Although this may appear to be a duplication at first sight, there may well be differences between the semantics of boolean expressions and assertions. For example, boolean disjunction may be defined in the language to be the asymmetric *conditional-or*, whereas the disjunction of assertions may be symmetric.

In VDM, the abstract syntax for assertions can be defined as in Figure 8. Again this avoids adding another level of constructor on top of expressions.

Another possible way of defining assertions in Z is to not make use of expressions but give all the alternatives explicitly, including those for expressions. Expressions can then be defined as a subset of assertions — see Figure 9. This abstract syntax covers both expressions (boolean and integer) as well as assertions. Of course, not all members of *AssertExpr* are valid, but consistency constraints can be used to select the valid assertions and expressions.

Although this approach may not be warranted for this example because we may want to give different semantics to boolean expressions and assertions, in general, it could be useful. One problem remaining, however, is that in using Z free types we cannot present the alternatives for boolean expression and then

$$
\begin{array}{lll}
\mathit{Operator_as} & = & \{\textsc{or_as}, \textsc{and_as}\} \\
\mathit{Assertion} & = & \mathit{Expr} \cup \mathit{Infix_as} \cup \mathit{Forall} \\
\mathit{Infix_as} & :: & \mathit{Assertion} \times \mathit{Operator_as} \times \mathit{Assertion} \\
\mathit{Forall} & :: & \mathit{Id} \times \mathit{Assertion}
\end{array}
$$

Figure 8: Abstract syntax of assertions (in VDM)

$$
\begin{array}{lll}
\mathit{AssertExpr} & ::= & \mathit{bool_as}\langle\langle \mathbb{B} \rangle\rangle \\
& | & \mathit{int_as}\langle\langle \mathbb{Z} \rangle\rangle \\
& | & \mathit{exvar_as}\langle\langle \mathit{Varref} \rangle\rangle \\
& | & \mathit{infix_as}\langle\langle \mathit{AssertExpr} \times \mathit{Operator} \times \mathit{AssertExpr} \rangle\rangle \\
& | & \mathit{forall_as}\langle\langle \mathit{Id} \times \mathit{AssertExpr} \rangle\rangle
\end{array}
$$

Figure 9: Abstract syntax of assertions (direct)

extend them later with the additional alternatives for assertions. Instead we have to give the complete set of alternatives up front. This is an area where the notation unwantedly constrains the order of presentation of the specification.

Interestingly, if we avoid the free type notation of Z and define expressions and assertions in more primitive Z notation, the problem can be overcome. In Z, the free type definition of *AssertExpr* is equivalent to introducing the basic type *AssertExpr* along with a set of constructors as in Figure 10. At this

$$[AssertExpr]$$

$$
\begin{array}{l}
bool_as : \mathbb{B} \rightarrowtail AssertExpr \\
int_as : \mathbb{Z} \rightarrowtail AssertExpr \\
exvar_as : Varref \rightarrowtail AssertExpr \\
infix_as : AssertExpr \times Operator \times AssertExpr \rightarrowtail AssertExpr \\
forall_as : Id \times AssertExpr \rightarrowtail AssertExpr \\
\hline
\mathsf{disjoint}\ \langle \mathrm{ran}\ bool_as, \mathrm{ran}\ int_as, \mathrm{ran}\ exvar_as, \mathrm{ran}\ infix_as, \mathrm{ran}\ forall_as \rangle \\
(\forall W : \mathbb{P}\ AssertExpr \bullet \\
\quad bool_as(\!|\mathbb{B}|\!) \cup int_as(\!|\mathbb{Z}|\!) \cup exvar_as(\!|Varref|\!) \cup \\
\quad\quad infix_as(\!|W \times Operator \times W|\!) \cup forall_as(\!|Id \times W|\!) \subseteq W \\
\quad \Rightarrow AssertExpr \subseteq W)
\end{array}
$$

Figure 10: Abstract syntax of assertions (expanded)

level it is now possible to split the definition of *AssertExpr*. The basic type and constructors needed for expressions can be introduced initially along with a disjointness constraint for this subset. It is even possible to define a subset of *AssertExpr* corresponding to objects constructable with just this subset of constructors complete with an induction principle (see Figure 11). Then, at a later stage the remaining constructors can be introduced, the disjointness condition extended and the induction principle for the whole type introduced. The main drawback of this approach is that the expanded definition is rather unwieldy compared with the free type notation.

3.5 Partial function definition

Another area where VDM and Z differ is in the definition of partial functions, especially using lambda notation. For example, the definition of meaning of an infix operator in VDM can simply be written

$$
\begin{array}{l}
MInfix(mk\text{-}Infix(e1, op, e2))(\rho) \mathrel{\widehat{=}} \\
\quad (\lambda\, \sigma \bullet MOperator(op)(MExpr(e1)(\rho)(\sigma), MExpr(e2)(\rho)(\sigma))).
\end{array}
$$

$$
\begin{array}{l}
Exp : \mathbb{P}\, AssertExpr \\
\hline
\langle bool_as (\!| \mathbb{B} |\!), int_as (\!| \mathbb{Z} |\!), exvar_as (\!| Varref |\!), \\
\qquad\qquad infix_as (\!| Exp \times Operator \times Exp |\!) \rangle \\
\qquad \text{partition } Exp \\
(\forall W : \mathbb{P}\, Exp \bullet \\
\qquad bool_as (\!| \mathbb{B} |\!) \cup int_as (\!| \mathbb{Z} |\!) \cup exvar_as (\!| Varref |\!) \cup \\
\qquad\qquad infix_as (\!| W \times Operator \times W |\!) \subseteq W \\
\qquad \Rightarrow Exp \subseteq W)
\end{array}
$$

Figure 11: Defining expressions as a subset of assertions

In Z this has to be written

$$
\begin{array}{l}
MExpr(infix(e1, op, e2)) = \\
\quad (\lambda\, \rho : \mathrm{dom}(MExpr\ e1) \cap \mathrm{dom}(MExpr\ e2) \bullet \\
\qquad (\lambda\, \sigma : \mathrm{dom}(MExpr(e1)(\rho)) \cap \mathrm{dom}(MExpr(e2)(\rho)) \mid \\
\qquad (MExpr(e1)(\rho)(\sigma), MExpr(e2)(\rho)(\sigma)) \in \mathrm{dom}(MOperator\ op) \bullet \\
\qquad\quad MOperator(op)(MExpr(e1)(\rho)(\sigma), MExpr(e2)(\rho)(\sigma)))).
\end{array}
$$

In Z, the domain of a partial function defined with lambda notation has to be made explicit. In VDM, the function is defined whenever the expression is well defined and not defined whenever the expression is not.

The advantage of the VDM approach is that one can ignore the detail of definedness and concentrate on the main characteristics of a definition. The Z approach, while more verbose, has the advantage that all the definedness constraints have to be made explicit and hence are less likely to be overlooked.

4 Commands

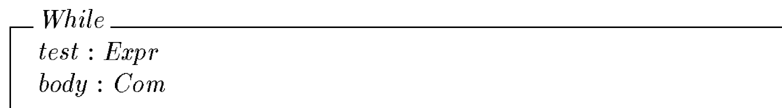
4.1 Abstract syntax of commands

The abstract syntax of commands is built up from descriptions of the different programming language commands. For example, an assignment consists of a left side variable reference and a right side expression. It is represented by the following schema.

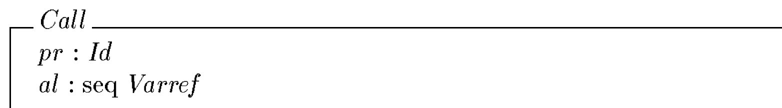
$$
\begin{array}{l}
\underline{\ Assign\ } \\
lhs : Varref \\
rhs : Expr \\
\hline
\end{array}
$$

The schema *If* has a component, *test*, of type *Expr* and two components, *then* and *else*, of type *Com*.

A *while* loop consists of an expresion, *test*, and a command, *body*.



A procedure call consists of a procedure identifier, *pr*, and an argument list, *al*. We only allow variable (**var**) parameters at this stage. Hence, the argument list is a sequence of variable references.



The abstract syntax for a command is given in Figure 12. A command can be any of the above alternatives, or a null command, or a block (discussed in Section 5.1).

$$
\begin{array}{rcl}
Com & ::= & null \\
& | & assign \langle\langle Assign \rangle\rangle \\
& | & if \langle\langle If \rangle\rangle \\
& | & while \langle\langle While \rangle\rangle \\
& | & call \langle\langle Call \rangle\rangle \\
& | & block \langle\langle Block \rangle\rangle
\end{array}
$$

Figure 12: Abstract syntax for commands

The form of definition of *Com* used here is not strictly allowed by [Spi92] because the definition of the schemas and *Com* above are mutually recursive as discussed in Section 3.2. An equivalent definition of *Com* could be written in Z by using Cartesian products directly in the free type definition of *Com* rather than using schemas. Such a definition is somewhat less readable.

4.2 Consistency constraints on commands

As with the consistency constraints on expressions, we define the set of environments in which a command is well formed — see Figure 13. A null command

$$
\begin{array}{|l}
WFCom : Com \longrightarrow (\mathbb{P}\, Senv) \\
\hline
WFCom(null) = Senv \\
WFCom(assign(asn)) = \{\rho : WFVarref(asn.lhs) \cap WFExpr(asn.rhs) \mid \\
\qquad TPVarref(asn.lhs)(\rho) = TPExpr(asn.rhs)(\rho)\} \\
WFCom(if(ift)) = WFBoolExpr(ift.test) \cap \\
\qquad WFCom(ift.then) \cap WFCom(ift.else) \\
WFCom(while(wh)) = WFBoolExpr(wh.test) \cap WFCom(wh.body)
\end{array}
$$

Figure 13: Consistency constraints for commands

is always well formed. An assignment command is well formed in an environment in which its left side is a well-formed variable reference, its right side is a well-formed expression, and the types of both sides are the same. A conditional command is well formed in an environment in which its *test* is a well-formed boolean expression and its *then* and *else* parts are well-formed commands. A *while* loop is well formed in an environment in which its *test* is a well-formed boolean expression and its body is a well-formed command.

We discuss the consistency constraints for procedure calls and blocks in Section 5.2.

4.3 Dynamic semantics of commands

Unlike an expression, a command does not have a value. Rather a command may affect the values of the variables in the store. We can model a command by a function from *Store* to *Store*. Given an initial store, σ, such a function returns the value of the store, σ', after execution of the command. Our commands are modelled by store-to-store transformations.

$$Tr == Store \nrightarrow Store$$

The meaning of a command is a function that, given an environment, returns a store-to-store transformation.

$$
\begin{array}{|l}
MCom : Com \longrightarrow (Env \nrightarrow Tr) \\
\hline
MCom(null) = (\lambda\, \rho : Env \bullet \mathrm{id}\, Store)
\end{array}
$$

The null command does not modify the contents of the store. We represent it by the identity store-to-store transformation.

$$(\mathrm{id}\, Store)(\sigma) = \sigma$$

An assignment command may modify the contents of the store. The location of the left side variable, *lhsloc*, is determined from the environment. The value of the right side expression in the environment ρ and store σ is determined and the resulting store is σ overridden ($\oplus$) by the mapping of *lhsloc* to *rhsval*.

$$\begin{array}{l}
MCom(assign(asn)) = \\
\quad (\lambda\, \rho : \mathrm{dom}(MVarref\ asn.lhs) \cap \mathrm{dom}(MExpr\ asn.rhs) \bullet \\
\quad\quad (\mathbf{l}et\ lhsloc == MVarref(asn.lhs)(\rho) \bullet \\
\quad\quad (\lambda\, \sigma : \mathrm{dom}(MExpr(asn.rhs)(\rho)) \mid lhsloc \in \mathrm{dom}(\sigma) \bullet \\
\quad\quad\quad (\mathbf{let}\ rhsval == MExpr(asn.rhs)(\rho)(\sigma) \bullet \\
\quad\quad\quad\quad \sigma \oplus \{lhsloc \mapsto rhsval\}))))
\end{array}$$

For the semantics of the *if* command we make use of the auxiliary function *Cond*, which takes three arguments:

- a function *ex* (representing the test in the conditional) that, given a store returns the value of the test expression in that store,
- a store-to-store transformation $st1$ (representing the *then* part), and
- a store-to-store transformation $st2$ (representing the *else* part).

The resultant store-to-store transformation behaves like $st1$ in those stores in which the test expression *ex* evaluates to true, and it behaves like $st2$ in those stores where *ex* evaluates to false. The domain of the result is all those store values that are either,

- in the domain of $st1$ and for which the value of *ex* is true, or
- in the domain of $st2$ and for which the value of *ex* is false.

$$\begin{array}{|l}
Cond : (Store \leftrightarrow Scval) \times Tr \times Tr \longrightarrow Tr \\
\hline
Cond(ex, st1, st2) = \\
\quad ex^{\sim} (\!| \{bool_val(True)\} |\!) \lhd st1\ \cup \\
\quad ex^{\sim} (\!| \{bool_val(False)\} |\!) \lhd st2
\end{array}$$

The inverse image of a value v through *ex*, i.e., $ex^{\sim}(\!|\{v\}|\!)$, is the set of all stores in which *ex* has the value v.

The meaning of the *if* command is defined in terms of *Cond*. The function *ex* is the meaning of the *test*, and $st1$ and $st2$ are the meanings of the *then* and

else parts of the *if* command.

$$
\begin{aligned}
&MCom(if(ift)) = \\
&\quad (\lambda\, \rho : \mathrm{dom}(MExpr\ ift.test) \cap \\
&\qquad\qquad\qquad \mathrm{dom}(MCom\ ift.then) \cap \mathrm{dom}(MCom\ ift.else) \bullet \\
&\qquad\qquad Cond(MExpr(ift.test)(\rho), \\
&\qquad\qquad\qquad MCom(ift.then)(\rho), \\
&\qquad\qquad\qquad MCom(ift.else)(\rho)))
\end{aligned}
$$

Defining the meaning of a *while* loop is complicated by the fact that we need to represent the repetitive nature of the loop. One approach to its definition is the following. If the test is false then the meaning of the loop is simply the identity function of stores, but if the test is true the meaning is the composition of the meaning of the body of the loop and the meaning of the whole loop.

$$
\begin{aligned}
&MCom(while(wh)) = \\
&\quad (\lambda\, \rho : \mathrm{dom}(MExpr\ wh.test) \cap \mathrm{dom}(MCom\ wh.body) \bullet \\
&\qquad Cond(MExpr(wh.test)(\rho), \\
&\qquad\qquad MCom(wh.body)(\rho) \mathbin{;} MCom(while(wh))(\rho), \\
&\qquad\qquad \mathrm{id}\ Store))
\end{aligned}
$$

The above 'definition' of the meaning of a *while* loop has the problem that it does not necessarily uniquely determine the meaning. For example, consider the *while* loop with test *True* and body the *null* command, that is, an infinite loop.

$$inf_loop == while(\mu\ While \mid test = bool(True) \wedge body = null)$$

The above definition becomes

$$MCom(inf_loop) = (\lambda\, \rho : Env \bullet MCom(inf_loop)(\rho))$$

This allows the function for the meaning of *inf_loop* to be any store-to-store function at all.

An alternative approach, that determines a unique meaning for the semantics of a *while* loop, considers all functions, f, that satisfy the equation:

$$f = Cond(MExpr(wh.test)(\rho), MCom(wh.body)(\rho) \mathbin{;} f, \mathrm{id}\ Store).$$

Such functions are said to be *fixed points* of the above equation. Of these functions the least defined such function is chosen as the meaning. That, is the function with the smallest domain.

$$
\begin{array}{|l}
least : (\mathbb{P}_1\ Tr) \rightarrowtail\!\!\!\!\!\!+ \ Tr \\
\hline
least = \{sf : \mathbb{P}_1\ Tr;\ lst : Tr \mid lst \in sf \wedge (\forall f : sf \bullet lst \subseteq f)\}
\end{array}
$$

The meaning of a *while* loop is defined to be the least fixed point of the above equation: the least defined function that is a fixed point of the above equation.

$$
\begin{aligned}
&MCom(while(wh)) = \\
&\quad (\lambda\, \rho : \mathrm{dom}(MExpr\ wh.test) \cap \mathrm{dom}(MCom\ wh.body) \bullet \\
&\qquad least\{f : Tr \mid \\
&\qquad\quad f = Cond(MExpr(wh.test)(\rho), \\
&\qquad\qquad MCom(wh.body)(\rho) \mathbin{;} f, \\
&\qquad\qquad \mathrm{id}\ Store)\})
\end{aligned}
$$

For example,

$$MCom(inf_loop) = (\lambda\, \rho : Env \bullet \varnothing),$$

where $\varnothing$ is the empty — nowhere defined — store-to-store mapping.

We treat the semantics of procedure calls and blocks in Section 5.3.

5 Procedures and Blocks

Procedures in the simple language have only variable (**var**) parameters and the body of a procedure is a single command. A command may, however, be a block, and within a block local variable and procedure declarations may be introduced. The body of a block is a sequence of commands.

5.1 Abstract syntax of procedures and blocks

The formal parameter list of a procedure is a sequence of variable declarations. We model it in the abstract syntax by a combination of a sequence, *fpl*, of the names of the formal parameters, and a mapping, *typem*, giving the type of each of the formal parameters. The body of a procedure is a command.

$$
\begin{array}{|l}
\textit{Proc} \\
\hline
fpl : \mathrm{seq}\ Id \\
typem : Id \pfun Sctype \\
body : Com \\
\hline
\mathrm{ran}\ fpl = \mathrm{dom}\ typem \\
\forall\, i, j : \mathrm{dom}\ fpl \bullet i \neq j \Rightarrow fpl(i) \neq fpl(j) \\
\hline
\end{array}
$$

The types of all the formal parameters must be defined by *typem*, and all the formal parameters must have distinct names.

We model a block by a combination of a mapping, *typem*, giving the types of all the variables declared locally to the block, a mapping, *procm*, giving all the procedures defined in the block, and a sequence of commands representing the body of the block.

$$
\begin{array}{|l}
\hline
\textit{Block} \\
typem : Id \pfun Sctype \\
procm : Id \pfun Proc \\
body : \mathrm{seq}\ Com \\
\hline
\mathrm{dom}\ typem \cap \mathrm{dom}\ procm = \{\} \\
\hline
\end{array}
$$

The same name may not be used for both a local variable and a procedure.

A program consists of a single command.

$$Program ::= program \langle\!\langle Com \rangle\!\rangle$$

5.2 Consistency constraints for procedures and blocks

A procedure is well-formed in an environment provided its body is well-formed in the same environment augmented with the declarations of the procedure's formal parameters.

$$
\begin{array}{|l}
WFProc : Proc \longrightarrow (\mathbb{P}\ Senv) \\
\hline
WFProc(pr) = \{\rho : Senv \mid \\
\quad (\rho \oplus (pr.typem \fcmp vartype)) \in WFCom(pr.body)\}
\end{array}
$$

The type map of the procedure's formal parameters is a mapping from identifiers to scalar types. That is almost an environment. However, an environment is a mapping from identifiers to attributes. To convert a scalar type into an attribute of a variable we need to apply the constructor *vartype*. To apply *vartype* to all the elements of the range of *pr.typem* to get an environment, all we need to do is to compose *pr.typem* and *vartype*. For example,

$$
\begin{array}{l}
pr.typem = \{x \mapsto int_type, b \mapsto bool_type\} \\
\Rightarrow \\
pr.typem \fcmp vartype = \{x \mapsto vartype(int_type), b \mapsto vartype(bool_type)\}
\end{array}
$$

To define the environment for the body of a block, we need to define the attributes of all the procedures defined locally to the block. At this stage we need to extend the definition of attributes and environments given in Section 2.4 so that we can distinguish between identifiers declared as variables and as procedures. For a variable the attribute recorded is its type (as before), and for a procedure the attribute recorded is the sequence of types of its arguments.

$$Attr ::= vartype \langle\!\langle Sctype \rangle\!\rangle \mid proctype \langle\!\langle \mathrm{seq}\ Sctype \rangle\!\rangle$$

Aside: The redefinition of *Attr* here is not strictly allowed in Z. One should have defined *Attr* initially as it is defined here. However, at that point procedures had not been introduced, so the introduction of the *proctype* alternative seems quite out of place in the presentation. This is an example of where the preferred presentation order conflicts with the free type notation, as discussed in Section 3.4. 2

The function *TPProcs* takes the procedure map (of a block) and returns a mini-environment representing the procedure definitions. The attribute recorded for each procedure is the sequence of types of its formal parameters.

$$
\begin{array}{|l}
TPProcs : (Id \mathrel{+\!\!\!\!\!\twoheadrightarrow} Proc) \longrightarrow Senv \\
\hline
TPProcs(procm) = (\lambda\, id : \mathrm{dom}\, procm \bullet \\
\qquad proctype((procm\ id).fpl \mathbin{;} (procm\ id).typem))
\end{array}
$$

The mini-environment is formed by mapping the name of each procedure into a procedure type attribute consisting of the sequence of types of the formal parameters. This sequence is formed by composing the sequence of formal parameter names – a function from the natural numbers to identifiers – with the type map for the formal parameters. This has the effect of forming a new sequence consisting of the result of applying the type map to each of the identifiers in the formal parameter sequence.

A procedure call is well formed in an environment in which the procedure identifier is defined and the argument list has the correct number of arguments, each of the correct type. The sequence of types of arguments expected by the procedure, *prty*, is extracted from the environment. It must be the same length as the actual argument list, *al*, in the call, and each actual argument must be a well-formed variable reference of the type expected by the procedure.

$$
\begin{array}{l}
WFCom(call(cl)) = \\
\quad \{\rho : Senv \mid (\exists\, prty : \mathrm{seq}\, Sctype \bullet \\
\qquad cl.pr \in \mathrm{dom}(\rho) \land \rho(cl.pr) = proctype(prty) \land \\
\qquad \#cl.al = \#prty \land \\
\qquad (\forall\, i : \mathrm{dom}\, cl.al \bullet \rho \in WFVarref(cl.al(i)) \land \\
\qquad\quad TPVarref(cl.al(i))(\rho) = prty(i)))\}
\end{array}
$$

A block is well-formed in an environment ρ provided,

- all the procedures local to the block are well-formed in ρ augmented by the mini-environment representing the variables declared locally to the block (*lvars*) but excluding the names of all the procedures defined locally to the block — the procedures defined locally to the block are excluded from the environment used to test their well-formedness because we do not allow recursive calls on procedures in our example language — and

- all the commands in the body of the block are well formed in ρ augmented with *lvars* plus the mini-environment representing the local procedures (*TPProcs*(*bl.procm*)).

$$
\begin{array}{l}
WFCom(block(bl)) = \{\rho : Senv \mid \\
\quad (\mathbf{let}\ lvars == bl.typem \mathbin{;} vartype \bullet \\
\quad (\mathbf{let}\ inproc == ((\mathrm{dom}\, bl.procm) \lhd \rho) \oplus lvars; \\
\qquad inbody == \rho \oplus lvars \oplus TPProcs(bl.procm) \bullet \\
\quad (\forall\, pr : \mathrm{ran}\, bl.procm \bullet inproc \in WFProc(pr)) \land \\
\quad (\forall\, com : \mathrm{ran}\, bl.body \bullet inbody \in WFCom(com))))\}
\end{array}
$$

A program consists of just a single command. The input to a program is the initial value of the variable *in* and the output of a program is the final value of the variable *out*. Hence, a program is well formed if its command is well formed in an environment that defines the distinct variables *in* and *out* to be of type integer.

$$\begin{array}{|l}
WFProgram : \mathbb{P}\ Program \\
in, out : Id \\
\hline
in \neq out \\
\mathbf{let}\ \rho == \{in \mapsto vartype(int_type), out \mapsto vartype(int_type)\} \bullet \\
\quad WFProgram = \{com : Com \mid \rho \in WFCom(com) \bullet program(com)\}
\end{array}$$

5.3 Dynamic semantics of procedures and blocks

The body of a procedure consists of a command and the meaning of a procedure call is essentially the meaning of the command. In the case of a procedure with no arguments it is exactly this but, in general, the meaning of a procedure call depends on the arguments. To model a procedure independently of any particular call, we need to account for the arguments to the procedure. We do this by making the denotation of a procedure a function that takes a sequence of locations, representing the locations of the actual parameters to the procedure, and returns a store-to-store transformation.

$$Procden == (\mathrm{seq}\ Scloc) \mathrel{+\!\!\!\rightarrow} Tr$$

We need to record the meaning of a procedure in the environment. Each identifier in an environment is either a variable, in which case it has an associated location, or it is a procedure, in which case it has an associated procedure denotation. We extend the type *Val* as follows.

$$Val ::= varloc\langle\!\langle Scloc \rangle\!\rangle \mid procval\langle\!\langle Procden \rangle\!\rangle$$

Aside: Again, this redefinition of *Val* is not strictly allowed in Z. This is another example of where we would like to be able to extend a free type definition as discussed in Section 3.4. 2

The meaning of a procedure, given a list of the locations of its parameters, is the meaning of the body of the procedure in an environment augmented to reflect the locations of the parameters — see Figure 14.

The effect of a procedure call on the store depends on the store-to-store transformation performed by the body of the procedure, and this in turn depends on the actual arguments supplied to the procedure call. The called procedure must be defined within the environment as a procedure. Its associated value is a procedure denotation (*prden*). This denotation is applied to the list of actual argument locations to return the store-to-store transformation performed by the call. The list of argument locations is determined from the environment by determining the location of each argument in the actual argument list from

$$
\begin{array}{|l}
MProc : Proc \longrightarrow (Env \longrightarrow Procden) \\
\hline
MProc(pr) = (\lambda\, \rho : Env \bullet \\
\quad \{arglocs : \operatorname{seq} Scloc;\ \rho' : Env \mid \\
\quad\quad \#arglocs = \#pr.fpl \land \\
\quad\quad \rho' = \rho \oplus \{i : \operatorname{dom} pr.fpl \bullet pr.fpl(i) \mapsto varloc(arglocs(i))\} \land \\
\quad\quad \rho' \in \operatorname{dom}(MCom\ pr.body) \bullet \\
\quad\quad\quad arglocs \mapsto MCom(pr.body)(\rho')\})
\end{array}
$$

Figure 14: Meaning of a procedure (in Z)

the environment.

$$
\begin{array}{l}
MCom(call(cl)) = \\
\quad \{\rho : Env;\ prden : Procden;\ arglocs : \operatorname{seq} Scloc \mid \\
\quad\quad cl.pr \in \operatorname{dom}(\rho) \land \rho(cl.pr) = procval(prden) \land \\
\quad\quad (\forall\, i : \operatorname{dom} cl.al \bullet \rho \in \operatorname{dom}(MVarref(cl.al(i)))) \land \\
\quad\quad arglocs = \{i : \operatorname{dom} cl.al \bullet i \mapsto MVarref(cl.al(i))(\rho)\} \land \\
\quad\quad arglocs \in \operatorname{dom} prden \bullet \\
\quad\quad\quad \rho \mapsto prden(arglocs)\}
\end{array}
$$

To define the meaning of a block, we need to define the meaning of a command sequence. To do that we make use of the auxiliary function *CompSeq* which takes a sequence of store-to-store transformations, representing the meanings of the commands in the sequence, and composes them to form the meaning of the whole command sequence.

$$
\begin{array}{|l}
CompSeq : (\operatorname{seq} Tr) \longrightarrow Tr \\
\hline
CompSeq(\langle\rangle) = \operatorname{id} Store \\
CompSeq(\langle st \rangle \frown sts) = st \fatsemi (CompSeq\ sts)
\end{array}
$$

The composition of an empty sequence of transformations is defined to be the identity function, and a sequence consisting of a command st and a sequence sts is the composition of st and the sequence composition of sts.

The meaning of a command sequence is the composition of the sequence of meanings of the individual commands.

$$
\begin{array}{|l}
MComSeq : (\operatorname{seq} Com) \longrightarrow (Env \twoheadrightarrow Tr) \\
\hline
MComSeq(sts) = (\lambda\, \rho : Env \mid (\forall\, i : \operatorname{dom} sts \bullet \rho \in \operatorname{dom}(MCom(sts(i)))) \bullet \\
\quad\quad CompSeq(\{i : \operatorname{dom} sts \bullet i \mapsto MCom(sts(i))(\rho)\}))
\end{array}
$$

To define a block we need to be able to extend a store with new locations for the local variables, initialise the new locations, execute the body of the block and then deallocate the (new) locations. The function *NewLocs* takes a set of identifiers and a store and allocates new distinct locations for all the identifiers. The store is passed as a parameter so that none of the currently used locations ($\mathrm{dom}\,\sigma$) is used.

$$
\begin{array}{|l}
NewLocs : (\mathbb{P}\, Id) \times Store \longrightarrow (Id \twoheadrightarrow\!\!\!\!\!\!\!\!\;\;\;\;\; Scloc) \\
\hline
NewLocs(ids, \sigma) = locm \Rightarrow \\
\qquad \mathrm{dom}\, locm = ids \wedge \\
\qquad \mathrm{ran}\, locm \cap \mathrm{dom}\, \sigma = \{\} \wedge \\
\qquad (\forall\, id1, id2 : \mathrm{dom}\, locm \bullet id1 \neq id2 \Rightarrow locm(id1) \neq locm(id2))
\end{array}
$$

The function *Initialise* takes a function giving the types of the new variables and a function giving the locations of the new variables and initialises the store so that integer variables have the value zero and boolean variables have the value *False*.

$$
\begin{array}{|l}
Initialise : (Id \twoheadrightarrow Sctype) \times (Id \twoheadrightarrow Scloc) \longrightarrow Tr \\
\hline
Initialise = \\
(\lambda\, typem : Id \twoheadrightarrow Sctype;\ locm : Id \twoheadrightarrow Scloc \mid \mathrm{dom}\, typem = \mathrm{dom}\, locm \bullet \\
\qquad (\lambda\, \sigma : Store \bullet \\
\qquad\qquad\qquad \sigma \oplus (\{id : \mathrm{dom}\, typem \bullet locm(id) \mapsto typem(id)\} \mathbin{;} \\
\qquad\qquad\qquad \{int_type \mapsto int_val(0), bool_type \mapsto bool_val(False)\})))
\end{array}
$$

At the end of a block all of the local variable locations are deallocated from the store.

$$
\begin{array}{|l}
Deallocate : (\mathbb{P}\, Scloc) \longrightarrow Tr \\
\hline
Deallocate(locs) = (\lambda\, \sigma : Store \bullet locs \lhd \sigma)
\end{array}
$$

For a block, the environment, ρ, needs to be augmented with the attributes of the local variables. Fresh locations are allocated for the variables ($locm$) and ρ' is the augmented environment. As well, the procedures defined locally to the block need to be added to the environment (ρ''). Starting in state σ, the variables are initialised, the body of the block is executed and the local variables are deallocated to get the final value of the store after execution of the block.

$$
\begin{array}{|l}
MBlock : Block \longrightarrow (Env \times Store \rightarrowtail\!\!\!\!\!\!+\; Store) \\
\hline
MBlock(bl) = \{\rho : Env;\ \sigma, \sigma' : Store \mid \\
\quad (\mathbf{let}\ locm == NewLocs(\mathrm{dom}\ bl.typem, \sigma) \bullet \\
\quad (\mathbf{let}\ \rho' == \rho \oplus (locm \mathbin{;} varloc) \bullet \\
\quad (\forall\ id : \mathrm{dom}\ bl.procm \bullet \rho' \in \mathrm{dom}(MProc(bl.procm(id)))) \land \\
\quad (\mathbf{let}\ \rho'' == \rho' \oplus \\
\qquad (\lambda\ id : \mathrm{dom}\ bl.procm \bullet procval(MProc(bl.procm(id))(\rho'))) \bullet \\
\quad \rho'' \in \mathrm{dom}(MComSeq(bl.body)) \land \\
\quad (\sigma \mapsto \sigma') \in \\
\qquad Initialise(bl.typem, locm) \mathbin{;} \\
\qquad MComSeq(bl.body)(\rho'') \mathbin{;} \\
\qquad Deallocate(\mathrm{ran}\ locm)))) \bullet \\
\qquad\quad (\rho, \sigma) \mapsto \sigma'\}
\end{array}
$$

$$
\begin{array}{l}
MCom(block(bl)) = (\lambda\ \rho : Env \bullet \\
\quad (\lambda\ \sigma : Store \mid (\rho, \sigma) \in \mathrm{dom}(MBlock\ bl) \bullet MBlock(bl)(\rho, \sigma)))
\end{array}
$$

The meaning of a program is a function that takes an integer as input and returns an integer as output. The input value is the initial value of the variable *in* and the output value is the final value of the variable *out* after executing the command which is the program.

$$
\begin{array}{|l}
MProgram : Program \rightarrowtail\!\!\!\!\!\!+\; (\mathbb{Z} \rightarrowtail\!\!\!\!\!\!+\; \mathbb{Z}) \\
extract_result : Scloc \longrightarrow (Store \rightarrowtail\!\!\!\!\!\!+\; \mathbb{Z}) \\
\hline
extract_result = (\lambda\ outloc : Scloc \bullet \\
\quad \{\sigma : Store;\ result : \mathbb{Z} \mid outloc \in \mathrm{dom}\,\sigma \land \sigma(outloc) = int_val(result) \bullet \\
\qquad \sigma \mapsto result\}) \\
\\
MProgram = \\
\quad (\mathbf{let}\ \rho_{pr} == NewLocs(\{in, out\}, \{\}) \mathbin{;} varloc; \\
\qquad in_loc == varloc^{\sim}(\rho_{pr}(in)); \\
\qquad out_loc == varloc^{\sim}(\rho_{pr}(out)) \bullet \\
\quad (\lambda\ pgm : Program \mid \rho_{pr} \in \mathrm{dom}(MCom(program^{\sim}(pgm))) \bullet \\
\qquad (\lambda\ n : \mathbb{Z} \bullet \{in_loc \mapsto int_val(n), out_loc \mapsto int_val(0)\}) \mathbin{;} \\
\qquad MCom(program^{\sim}(pgm))(\rho_{pr}) \mathbin{;} \\
\qquad extract_result(out_loc)))
\end{array}
$$

The environment of the program's command (ρ_{pr}) contains the variables *in* and *out* with locations *in_loc* and *out_loc*, respectively. In the initial store *in* has the value n and *out* has the value zero. The result of the program is extracted from the value of *out* in the store after execution of the program's command.

6 Conclusions

As Z has not been designed with the thought of specifying languages — unlike VDM which has been used extensively for this purpose — some of the features

of Z become awkward in this context. In particular, if one would like to come up with the simplest specification of language features, the form of free type notation and the definition of functions via lambda notation in Z are awkward. These problems have been further illustrated in the definition of commands and procedures. For example, the definition of *MProc* in VDM [Jon90a, page 248] follows.

$$MProc : Proc \longrightarrow Env \longrightarrow Procden$$
$$MProc(mk\text{-}Proc(fpl, tm, s))(\rho) \mathrel{\widehat{=}}$$
$$(\lambda\, ll \bullet MCom(s)(\rho \dagger \{fpl(i) \mapsto ll(i) \mid i \in \mathrm{inds}\, fpl\})$$

This should be compared with the definition of *MProc* in Z given in Figure 14.

One issue that we have not discussed so far is the underlying semantics of the specification languages and how that affects the semantics of the languages defined. The VDM specification language is based on domain theory and has predefined mechanisms (such as **def**) for defining the semantics of constructs such as a *while* loop.

Z uses set theory and has no predefined domain theory. The specifier has to build the semantics on top of set theory. In the case of the simple programming language, this can be done by representing commands as partial functions and defining meaning functions appropriately. Some care needs to be taken with the semantics of the *while* loop given in Section 4.3. The semantics of *while* are given as the least fixed point of an equation. We need to show that the least fixed point exists and is unique. This is straightforward for the case of the *while* loop as the corresponding functional is monotonic and continuous.

The simple programming language discussed in this paper avoids some of the more complex problems where a thorough knowledge of domain theory is required. For example, the introduction of procedure-valued variables would require the use of reflexive domains. In Z, such a theory is not available and would have to be built.

Our overall conclusion is that, for specifying simple programming languages or program input languages, Z is adequate. The questions raised in Section 3 indicate that there are issues in the design of specification languages that impact on their suitability for specifying languages. Areas where Z could be improved for this purpose include the following.

- Allowing the use mutual recursion and schemas to specify the abstract syntax of constructs like commands in Section 4.1. This provides a more readable specification than the alternative single recursive definition.

- The Z free type mechanism can be a bit cumbersome when multiple layers of free type definitions are involved, such as when extension of free types is required.

- The requirement to precisely specify the domain of partial functions when using lambda notation in Z can be cumbersome, but this needs to be weighed against the fact that the explicit domain specification provides an additional cross check on the specifier's intentions.

7 Acknowledgements

I would like to acknowledge the work of Cliff Jones [Jon90a] on which this paper and the comparison is based. I would also like to thank David Carrington, Gordon Rose and Luke Wildman for their comments on earlier drafts of this paper. This paper grew out of lecture notes for *CS322 Semantics of Programming Languages* [Hay92]. The lecture notes also include concrete syntax, value parameters for procedures, a more tutorial approach to recursive definitions, and axiomatic semantics for the language.

References

[Hay92] I. J. Hayes. CS322 Semantics of Programming Languages. Lecture notes, Department of Computer Science, University of Queensland, Brisbane, Australia, 1992.

[Jon90a] C. B. Jones. A small language definition. In C. B. Jones and R. C. F. Shaw, editors, *Case Studies in Systematic Software Development*, chapter 9, pages 235–256. Prentice Hall International, 1990.

[Jon90b] C. B. Jones. *Systematic Software Development using VDM.* Prentice Hall International, second edition, 1990.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International, second edition, 1992.