The Formal Semantics of ISO VDM-SL*

Peter Gorm Larsen
Institute of Applied Computer Science (IFAD)
Forskerparken 10
DK-5230 Odense M
Denmark

Wiesław Pawłowski
Institute of Computer Science
Polish Academy of Sciences
ul. Abrahama 18
81-825 Sopot
Poland

August 13, 2011

Abstract

This paper provides an overview of the formal semantics of VDM-SL which currently is being standardised by ISO. This is a specification language used in the formal method known as the Vienna Development Method (or simply VDM). In this paper we will focus on the foundations and the semantics of a rather unique combination of looseness and recursion.

1 Introduction

The *Vienna Development Method* (VDM) is a formal method for the description and development of sequential software systems. For quite a long time the VDM specification language was used in a number of different dialects. In the mid-eighties, in order to harmonise these, British Standards Institution (BSI) decided to standardise it [Sen87, Andrews&88, Plat&92, Parkin94]. In the early nineties this was taken up by the International Organisation for Standardisation (ISO) [ISOVDM93].

In this paper we provide an overview of the formal semantics of the standardised specification language VDM-SL. This formal semantics is also known as the *dynamic semantics* since it describes a possible behaviour of specified systems. VDM-SL is a non-executable *model-oriented* specification language. A distinguishing feature of the model-oriented approach is that the intended properties of the specified system are characterised by "building" its explicit abstract "implementation" (model). In VDM-SL, a specification may denote not necessarily a single abstract model, but a (possibly infinite) set of such models. This is due

*Accepted for Publication by a special issue on formal methods and standards of the "Computer Standards and Interfaces" journal (to be published September 1995).

to the fact that the VDM-SL language provides a concept of "looseness". Looseness is an abstraction mechanism which allows certain design decisions to be postponed till a later stage of development.

The dynamic semantics of VDM-SL was first presented in [Arentoft&88] which was based on work by Brian Monahan [Monahan85]. An overview of this was published in [Larsen&89] and the dynamic semantics was checked by two different review boards chaired by Andrzej Blikle and Kees Middelburg, respectively. The latest version of the dynamic semantics is presented in the VDM-SL standard [ISOVDM93]. In this paper we limit ourselves to explain those parts where a novel approach has been taken. Thus, we focus on the foundations of the semantic definition, and in particular on an interesting combination of looseness and recursion which is given a least fixed point semantics.

The paper starts by presenting the pre-requisites of the formal definition, such as an introduction to the mathematical notation and the underlying domain universe. After a short overview of the structure of the dynamic semantics definition, the concept and semantics of looseness is introduced. A few functions from the actual formal definition are presented to illustrate how looseness is dealt with in the presence of recursion. Finally a few examples illustrating this kind of combination of looseness and recursion are given, followed by concluding remarks. To fully understand the formal parts of this paper the reader is assumed to have some knowledge of traditional denotational semantics (e.g. from [Schmidt86]).

2 Mathematical Notation

This section introduces the mathematical notation that will be the "meta-language" employed to define the dynamic semantics of VDM-SL.

The notation is based on set-theory, assuming the existence of certain basic sets and operations on sets and using these to build spaces of values – as well as operations for manipulating these – suitable for defining the semantics of VDM-SL.

The main purpose of this section is to give an overview of the notation which is going to be used in subsequent sections. The logic notation and the basic set-theoretic notation will be presented first. Then the notation for mathematical structures such as Cartesian products, functions, and finite mappings will be described in that order. Finally, we will define the notation for structured expressions (if-then-else etc.) which are used in the definition of the dynamic semantics.

2.1 Logic Notation

The logic employed in the semantics definition is a two-valued predicate logic with strong equality. We use symbols T for truth and F for falsehood. Negation is written as $\mathsf{not}P$, while conjunction, disjunction, implication and equality are written as $\wedge$, $\vee$, $\Rightarrow$ and $=$, respectively. The universal and existential quantifiers will be denoted by $\forall$ and $\exists$.

2.2 Basic Set Theory

All of the constructs in the semantics are given within a "naïve" set theory: that is, a rigorous, but informal theory of arbitrary constructable sets (i.e. avoiding the Russell paradox).

We accept the intuitive concept of a *set* as a collection of objects, called *elements* or *members* of the set. The notation $a \in A$ means that a is an element of the set A. The

empty set is denoted by { }. The *set-theoretic operations* $\cup$, $\bigcup$ (*union* and *distributed union*, respectively) have their usual meaning.

For any set $\mathbf{A}$, $\mathbb{P}(\mathbf{A})$ denotes the powerset of $\mathbf{A}$, i.e. the set of all subsets of $\mathbf{A}$. Similarly, $\mathbb{F}(\mathbf{A})$ denotes the set of all finite subsets of $\mathbf{A}$.

Within the mathematical notation two different methods will be used to construct sets. Either the elements will be enumerated, written as $\{a,b,c,\ldots\}$, or the set will be constructed in an implicit manner by means of a set comprehension, which is written $\{\ \mathbf{f}(\mathbf{a}) \mid \mathbf{a} \in \mathbf{A} \cdot \mathbf{P}(\mathbf{a})\}$. Both of these constructs have their usual meanings.

2.3 Cartesian Products

The (generalised) Cartesian product of $\mathbf{n}$ sets $A_1, \ldots, A_n$ is defined by:

$$\bigtimes_{\mathbf{i}=1}^{\mathbf{n}} \mathbf{A_i} = \{\ (\mathbf{a}_1, \ldots, \mathbf{a_n}) \mid \mathbf{a}_1 \in \mathbf{A}_1, \ldots\ , \mathbf{a_n} \in \mathbf{A_n}\ \}$$

The *tupling* operator $(\text{-}, \ldots, \text{-})$ used in the definition has the following property:

$$\forall \mathbf{a}_1, \mathbf{a}_1{}' \in \mathbf{A}_1, \ldots, \forall \mathbf{a_n}, \mathbf{a_n}{}' \in \mathbf{A_n} \cdot$$

$$(\mathbf{a}_1, \ldots, \mathbf{a_n}) = (\mathbf{a}_1{}', \ldots, \mathbf{a_n}{}') \Longleftrightarrow (\mathbf{a}_1 = \mathbf{a}_1{}') \wedge \ \ldots\ \wedge (\mathbf{a_n} = \mathbf{a_n}{}')$$

Note that tuples with different lengths come from different Cartesian products, and hence are incomparable.

For the usual (binary) Cartesian product, which is a special case of the above construction, we shall use the standard notation $\mathbf{A} \times \mathbf{B}$.

2.4 Functions

For sets $\mathbf{A}$ and $\mathbf{B}$, $\mathbf{A} \rightharpoonup \mathbf{B}$ denotes the set of all *partial functions* from $\mathbf{A}$ to $\mathbf{B}$. This set is defined by:

$$\mathbf{A} \rightharpoonup \mathbf{B} = \{\ \mathbf{f} \mid \mathbf{f} \in \mathbb{P}(\mathbf{A} \times \mathbf{B}) \cdot\ \forall (\mathbf{a}_1, \mathbf{b}_1), (\mathbf{a}_2, \mathbf{b}_2) \in \mathbf{f} \cdot \mathbf{a}_1 = \mathbf{a}_2\ \Rightarrow\ \mathbf{b}_1 = \mathbf{b}_2\}.$$

In addition to writing $f \in A \rightharpoonup B$ we will also use the more common notation $f : A \rightharpoonup B$.

For every function $f : A \rightharpoonup B$, one can specify two sets called the *domain* and the *range* of $\mathbf{f}$, respectively. They are defined by:

$$\delta_0(\mathbf{f}) = \{\mathbf{a} \mid \mathbf{a} \in \mathbf{A} \cdot\ \exists \mathbf{b} \in \mathbf{B}\ \cdot (\mathbf{a}, \mathbf{b}) \in \mathbf{f}\}$$

$$\delta_1(\mathbf{f}) = \{\mathbf{b} \mid \mathbf{b} \in \mathbf{B} \cdot\ \exists \mathbf{a} \in \mathbf{A}\ \cdot (\mathbf{a}, \mathbf{b}) \in \mathbf{f}\}.$$

For a function $f : A \rightharpoonup B$ and an element $a \in \delta_0(f)$ the *application* of $\mathbf{f}$ to $\mathbf{a}$ is defined to be the unique element $b \in \delta_1(f)$ such that $(a,\ b) \in f$. We will use the notation $f(a)$ for function application.

The set of all *total functions* between sets $\mathbf{A}$ and $\mathbf{B}$ is a subset $A \to B \subseteq A \rightharpoonup B$ defined by:

$$\mathbf{A} \to \mathbf{B} = \{\mathbf{f} \mid \mathbf{f} : \mathbf{A} \rightharpoonup \mathbf{B} \cdot \delta_0(\mathbf{f}) = \mathbf{A}\}.$$

In the sequel we shall define semantic functions from the dynamic semantics using the following notation:

$$FunctionName : FunctionType$$
$$FunctionName(arguments) \triangleq Function\text{-}definition$$

To give a simple example let us define the square function on natural numbers:

$$Square : \mathbb{N} \to \mathbb{N}$$
$$Square(n\ ) \triangleq n \times n$$

We shall also define functions using a typed λ-notation[1]. In order to define the meaning of a λ-expression we shall briefly discuss the concepts of free and bound variables and substitution.

In an expression of the form $\lambda \mathbf{x} \in \mathbf{A}.\mathbf{e}$ all occurrences of the variable $\mathbf{x}$ in the expression $\mathbf{e}$ are bound occurrences. All occurrences of variables in an expression that are not bound are free occurrences. $\mathbf{e}[\mathbf{e}'/\mathbf{x}]$ is the expression resulting from substituting free occurrences of the variable $\mathbf{x}$ in expression $\mathbf{e}$ with expression $\mathbf{e}'$, provided that no free variables in $\mathbf{e}'$ are captured within $\mathbf{e}$.

Now we can define the meaning of $\lambda \mathbf{x} \in \mathbf{A}.\mathbf{e}$. If $\mathbf{e}[\mathbf{a}/\mathbf{x}] \in \mathbf{B}$ whenever $\mathbf{a} \in \mathbf{A}$ then $\lambda \mathbf{x} \in \mathbf{A}.\mathbf{e} \in \mathbf{A} \to \mathbf{B}$ and it denotes the function:

$$\{(\mathbf{a}, \mathbf{b}) \mid \mathbf{a} \in \mathbf{A} \cdot \mathbf{b} = \mathbf{e}[\mathbf{a}/\mathbf{x}]\}.$$

2.5 Finite Maps

Finite maps or *mappings* are partial functions having a finite domain. For sets $\mathbf{A}$ and $\mathbf{B}$, the set of all mappings from $\mathbf{A}$ to $\mathbf{B}$ will be denoted by $\mathbb{M}(\mathbf{A}, \mathbf{B})$. It can be formally defined as follows:

$$\mathbb{M}(\mathbf{A}, \mathbf{B}) = \{\mathbf{m} : \mathbf{A} \nrightarrow \mathbf{B} \mid \delta_0(\mathbf{m}) \text{ is finite}\}.$$

Mappings can be defined by enumeration and by map comprehension. For $\mathbf{a}_1, \ldots, \mathbf{a_n} \in \mathbf{A}$, where the $\mathbf{a_i}$ are distinct, and $\mathbf{b}_1, \ldots, \mathbf{b_n} \in \mathbf{B}$,

$$\mathbf{m} = \{\mathbf{a}_1 \ \mapsto\ \mathbf{b}_1, \ldots, \mathbf{a_n} \ \mapsto\ \mathbf{b_n}\},$$

is a mapping in $\mathbb{M}(\mathbf{A}, \mathbf{B})$ such that

$$\delta_0(\mathbf{m}) = \{\mathbf{a}_1, \ldots, \mathbf{a_n}\} \wedge\ \forall \mathbf{i} \in \{1, \ldots, \mathbf{n}\} \cdot \mathbf{m}(\mathbf{a_i}) = \mathbf{b_i}.$$

For $\mathbf{f} : \mathbf{A} \nrightarrow \mathbf{B}$, $\mathbf{A}' \in \mathbb{F}(\mathbf{A})$, such that $A' \subseteq \delta_0(f)$ and $\mathbf{P}$ is a unary predicate on $\mathbf{A}$,

$$\mathbf{m} = \{\mathbf{a} \ \mapsto\ \mathbf{f}(\mathbf{a}) \mid \mathbf{a} \in \mathbf{A}' \cdot \mathbf{P}(\mathbf{a})\}$$

[1] In the definition of the dynamic semantics there will be places where the type will be omitted as it is "obvious" (e.g. $\lambda env \in ENV\ .\ body$ will be abbreviated to $\lambda env\ .\ body$). This convention is used for notational convenience and we believe that the mnemonics used (for variable identifiers) provide the reader with sufficient knowledge to understand the definition.

is a mapping in $\mathbb{M}(\mathbf{A},\mathbf{B})$ such that $\mathbf{m}(\mathbf{a}) = \mathbf{f}(\mathbf{a})$ if $\mathbf{a} \in \mathbf{A}'$ and $\mathbf{P}(\mathbf{a})$ is true; $\mathbf{m}(\mathbf{a})$ is undefined otherwise. The empty map is denoted by $\{\mapsto\}$.

There are two standard operators on mappings used in the paper. One of them is the *domain* operator:

$$\underline{\text{dom}} : \mathbb{M}(\mathbf{A},\mathbf{B}) \to \mathbb{F}(\mathbf{A}) \quad \underline{\text{dom}}(\mathbf{m}) = \delta_0(\mathbf{m})$$

The other one is called *overwrite*. I thas the following type:

$$\underline{\text{overwrite}} : \mathbb{M}(\mathbf{A},\mathbf{B}) \times \mathbb{M}(\mathbf{A},\mathbf{B}) \to \mathbb{M}(\mathbf{A},\mathbf{B})$$

The value of $\underline{\text{overwrite}}(\mathbf{m}_1,\mathbf{m}_2)$ is a mapping m such that $\underline{\text{dom}}(m) = \underline{\text{dom}}(m_1) \cup \underline{\text{dom}}(m_2)$ and for all $\mathbf{a} \in \underline{\text{dom}}(m)$ if $a \in \underline{\text{dom}}(m_2)$ then $m(a) = m_2(a)$ else $m(a) = m_1(a)$.

2.6 Structured Expressions

A number of structured constructs are used in the semantics definition.
The **if** expression has the usual meaning:

$$\textbf{if } \mathsf{T} \textbf{ then } \mathbf{e}_1 \textbf{ else } \mathbf{e}_2 \;=\; \mathbf{e}_1$$

$$\textbf{if } \mathsf{F} \textbf{ then } \mathbf{e}_1 \textbf{ else } \mathbf{e}_2 \;=\; \mathbf{e}_2.$$

The **let** construct

$$\textbf{let } \mathbf{x} = \mathbf{e} \textbf{ in } \mathbf{e}'$$

can be read as a shorthand for:

$$(\lambda\, \mathbf{x}\colon \mathbf{X}.\mathbf{e}')(\mathbf{e}).$$

where the type $\mathbf{X}$ is the type of the expression $\mathbf{e}$.

3 The Domain Universe

The domain universe for VDM-SL has been inspired by [Tarlecki&90]; it provides denotations for all values expressible in VDM-SL. This section presents a short overview of the domain universe which is needed to understand the semantics of VDM-SL. This will mainly be a collection of definitions which introduces the concepts on which the semantics is based.

3.1 Basic Definitions

In order to understand the basics of the domain universe construction below it is necessary to give a few definitions related to complete partial orders and fixed points. This is well-known material and we mainly present it here in order to define the notation used below.

Definition 3.1 (Partial Ordering) A binary relation $\sqsubseteq$ on $\mathbf{D}$ is called a *partial ordering* on $\mathbf{D}$ iff it is:

1. Reflexive: for all $\mathbf{a} \in \mathbf{D}, \mathbf{a} \sqsubseteq \mathbf{a}$.

2. Antisymmetric: for all $\mathbf{a}, \mathbf{b} \in \mathbf{D}, \mathbf{a} \sqsubseteq \mathbf{b}$ and $\mathbf{b} \sqsubseteq \mathbf{a}$ imply $\mathbf{a} = \mathbf{b}$.
3. Transitive: for all $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbf{D}, \mathbf{a} \sqsubseteq \mathbf{b}$ and $\mathbf{b} \sqsubseteq \mathbf{c}$ imply $\mathbf{a} \sqsubseteq \mathbf{c}$.

Definition 3.2 (Partially ordered set) A *partially ordered set* $\mathbf{A}$ is a pair $(|\mathbf{A}|, \sqsubseteq_{\mathbf{A}})$ where $|\mathbf{A}|$ is a set and $\sqsubseteq_{\mathbf{A}}$ is a partial ordering on $|\mathbf{A}|$.

Let $\mathbf{A} = (|\mathbf{A}|, \sqsubseteq_{\mathbf{A}})$ be a partially ordered set in the following definitions.

Definition 3.3 (Upper bound) An *upper bound* of a set $\mathbf{X} \subseteq |\mathbf{A}|$ is an element $\mathbf{a} \in |\mathbf{A}|$ such that for all $\mathbf{x} \in \mathbf{X}$, $\mathbf{x} \sqsubseteq_{\mathbf{A}} \mathbf{a}$.

Definition 3.4 (Least upper bound) A *least upper bound* of a set $\mathbf{X} \subseteq |\mathbf{A}|$ in $\mathbf{A}$ is an element $\bigsqcup_{\mathbf{A}} \mathbf{X} \in |\mathbf{A}|$ such that $\bigsqcup_{\mathbf{A}} \mathbf{X}$ is an upper bound of $\mathbf{X}$, and for any upper bound $\mathbf{a}$ of $\mathbf{X}$, $\bigsqcup_{\mathbf{A}} \mathbf{X} \sqsubseteq_{\mathbf{A}} \mathbf{a}$.

Definition 3.5 (Least element (bottom)) An element $\bot_{\mathbf{A}} = \bigsqcup_{\mathbf{A}} \{\}$, if it exists, is called the *least element* of $\mathbf{A}$.

Definition 3.6 (Countable chain) A sequence $(\mathbf{a_i} \mid \mathbf{i} \in \omega)$ of elements of $|\mathbf{A}|$ is called a *countable chain* in $\mathbf{A}$ if it is increasing (i.e. $\mathbf{a}_0 \sqsubseteq_{\mathbf{A}} \mathbf{a}_1 \sqsubseteq_{\mathbf{A}} \ldots$), where ω is the first uncountable ordinal.

Definition 3.7 (Complete partial order) $\mathbf{A}$ is called a *complete partial order*, or a *cpo*, if it has a least element and for each countable chain of its elements there exists a least upper bound of the chain in $\mathbf{A}$.

Definition 3.8 (Continuous function) Given two cpos $\mathbf{A}$ and $\mathbf{B}$, a function : $\mathbf{f}\colon |\mathbf{A}| \to |\mathbf{B}|$ is *continuous* if it preserves the least upper bounds of all countable chains in $\mathbf{A}$, i.e. for any chain $(\mathbf{a_i} \mid \mathbf{i} \in \omega)$, we have:

$$\mathbf{f}(\bigsqcup_{\mathbf{A}} \{\mathbf{a_i} \mid \mathbf{i} \in \omega\}) = \bigsqcup_{\mathbf{B}} \{\mathbf{f}(\mathbf{a_i}) \mid \mathbf{i} \in \omega\}.$$

Proposition 3.9 For any cpo $\mathbf{A}$ and continuous function $\mathbf{f}\colon |\mathbf{A}| \to |\mathbf{A}|$, the least upper bound:

$$\bigsqcup_{\mathbf{A}} (\mathbf{f^n}(\bot_{\mathbf{A}}) \mid \mathbf{n} \in \omega)$$

of the chain $(\mathbf{f^n}(\bot_{\mathbf{A}}) \mid \mathbf{n} \in \omega)$, where:

- $\mathbf{f}^0(\bot_{\mathbf{A}}) = \bot_{\mathbf{A}}$, and
- $\mathbf{f}^{\mathbf{n}+1}(\bot_{\mathbf{A}}) = \mathbf{f}(\mathbf{f^n}(\bot_{\mathbf{A}}))$ for $\mathbf{n} \geq 0$,

is the least fixed point of the function $\mathbf{f}$, i.e. the least solution to the recursive equation

$$\mathbf{x} = \mathbf{f}(\mathbf{x}).$$

We write the least fixed point of $\mathbf{f}$ as $\mathsf{Y}f$ where Y is a least fixed point operator.

3.2 A Universe of Complete Partial Orders

In the domain universe construction a universe of complete partial orders is built. This universe contains a family of cpos which contains a number of basic cpos (corresponding to the basic types in VDM-SL) and is closed under a number of cpo operators (corresponding to the type constructors from VDM-SL[2]). In addition it is closed under unions of countably many, union-compatible sets of cpos. Union-compatible cpos are families of cpos where the union of the cpos itself is a cpo. Thus, the universe of cpos, which will be called **CPO**, is rich enough to provide meaning for all the type operators in VDM-SL, as well as certain recursive type definitions over these type operators. However, inside **CPO** it is not possible to describe types which are restricted by some kind of invariant constraint. In the process of construction this universe all values become tagged with information about what kind of value it is (e.g. a set, a map or whatever). The detailed construction of this universe is presented in [Tarlecki&90] and in [ISOVDM93].

3.3 A Universe of Domains

All legal type equations written in VDM-SL (including invariant constraints) are given semantics in terms of VDM domains which are defined as:

Definition 3.10 (VDM domain) A *VDM domain* is a pair:

$$\mathbf{A} = ((|\mathbf{A}|, \sqsubseteq_{\mathbf{A}}), \|\mathbf{A}\|)$$

where $(|\mathbf{A}|, \sqsubseteq_{\mathbf{A}})$ is a cpo (from **CPO**) and $\|\mathbf{A}\| \subseteq |\mathbf{A}| \setminus \{\bot_{\mathbf{A}}\}$.

Having defined the concept of a VDM domain, we are now ready for the last step of our universe construction.

Construction 3.11 (Domain universe) The universe of domains for VDM, **DOM**, is defined by:

$$\mathbf{DOM} = \{\, ((|\mathbf{A}|, \sqsubseteq_{\mathbf{A}}), \|\mathbf{A}\|) \mid (|\mathbf{A}|, \sqsubseteq_{\mathbf{A}}) \in \mathbf{CPO} \wedge \|\mathbf{A}\| \subseteq |\mathbf{A}| \setminus \{\bot_{\mathbf{A}}\} \,\}.$$

The operators used to close the *CPO* universe are now lifted to work at the *DOM* level. Finally, the collection of all values which can be used in a VDM-SL specification is constructed.

Construction 3.12 (Value universe) The universe of values for VDM, **VAL**, is defined by:

$$\mathbf{VAL} = \bigcup \{\, |\mathbf{A}| \mid ((|\mathbf{A}|, \sqsubseteq_{A}), \|\mathbf{A}\|) \in \mathbf{DOM} \,\}$$

[2]Some of these cpo operators have specifically been introduced with limitations in the way the type constructors can be used. Elements of sets and domains of maps must be flat values (i.e. may not contain any function values or values with functions as components). This have been done to ensure continuity of basic VDM-SL operations such as membership of a set.

4 Overview of the Semantic Definition

In traditional denotational semantics it is customary to provide a meaning function for each kind of syntactic component. Such a meaning function is a mapping from a syntax category to its meaning. This is done by means of a composition of the meaning of the components of the abstract syntax category. This means that in the case of a specification language with looseness, this approach would explicitly map the abstract syntax of the entire specification to the set of models which the specification denotes. However, this explicit style traditionally uses an order in which the definitions from the object language (in this case VDM-SL) must appear. In VDM-SL such an ordering is not defined and in general there can be mutual dependencies between definitions in different syntactic categories. The presence of looseness also makes definitions formulated with the explicit style more difficult to read [Arentoft&88]. Therefore the dynamic semantics of VDM-SL has been formulated in an implicit relational style instead of the traditional constructive style of denotational semantics.

4.1 *SemSpec* and *IsAModelOf*

The top-level function which gives meaning to a syntactic specification is defined as:

$$
\begin{array}{l}
SemSpec : Document \to \mathbb{P}(ENV) \\
SemSpec(doc) \triangleq \\
\quad \{\, env \mid env \in ENV \cdot IsAModelOf(env,\ doc) \,\}
\end{array}
$$

"Candidate" models (also called environments) are taken from the set *ENV* which contains all maps from identifiers to possible denotations for VDM-SL constructs (including *VAL* and *DOM*). *IsAModelOf* is a predicate which checks whether a given environment satisfies (in a formal sense) a given specification. If it does, it is called a *model* of the specification. The semantic function *SemSpec* for a given specification yields the set of all its models. The predicate *IsAModelOf* naturally needs to check whether all identifiers which have been defined in the specification are present in the environment. If it is the case, the environment is expanded with a number of constructs, which the definitions from the specification implicitly define[3]. Each component of the specification is now verified according to such an expanded environment. Because the definitions can be mutually dependent upon constructs from different categories, this is done for each category (functions, types, operations, etc.) by a meaning function for that category. Here it is important that the denotation of the constructs from the category being verified is removed from the environment. The remaining part of the environment provides the context in which the meaning of the constructs in this category is to be found. In this way an order between the definitions becomes available because of the implicit style of definition. The rationale behind the removal of the constructs from the category being verified is that in case of mutual recursion between constructs in such a category the semantics of these constructs should not be affected by the denotations of those constructs in the candidate model.

[3] As an example of such constructs we can mention selector, constructor and modifier functions for composite types.

4.2 Definers and Loose Definers

The meaning of the different kinds of definitions can be considered as an environment-to-environment transformation, adding more information to the environment. We call such transformations definers and loose definers (in case a construct can be potentially loosely specified). These can be explained by:

$$\begin{aligned} \text{Def} &= \text{ENV} \to (\text{ENV} \cup \{ \underline{\text{err}} \}) \\ \text{LDef} &= \mathbb{P}(\text{Def}) \end{aligned}$$

where $\underline{\text{err}}$ is a special symbol indicating that in the given environment the syntactic definition cannot be given any sensible meaning.

4.3 Semantics of the different kind of Constructs

Type definitions in VDM-SL is given a least fixed point semantics using the domain universe introduced in Section 3. No looseness is permitted in invariant expressions so types denote unique domain values from DOM.

Value (i.e. constant) definitions and function definitions can be loose, and the interpretation of this looseness will be discussed in the remaining part of this paper. Mutually recursive definitions are given a least-fixed-point semantics except for implicitly defined functions which are given an all-fixed-point semantics because no real functional is available in this case. Functions can also be polymorphic, but for simplicity, we do not take that into account in this paper.

Operation definitions can also contain looseness but here it is treated as non-determinism. Thus, an operation will denote a relation between input value (and state) and corresponding output value (and state). Implicitly defined operations are given an all-fixed-point semantics like for implicitly defined functions. The semantics of explicitly defined operations resembles a least-fixed-point semantics, but we cannot claim it to be so because there is no proper ordering between the operation denotations.

5 The Semantics of Looseness

We have mentioned the concept of 'looseness' a number of times above without being precise about its semantics. In this section we will, explain how looseness can be interpreted.

Looseness can at least be interpreted in two different ways: as *under-determinedness* (allowing several different deterministic implementations) or as *non-determinism* (allowing non-deterministic implementations). As illustrated in [Søndergaard&87, Søndergaard&92] there are different choices when a non-deterministic semantics is used. With under-determined interpretation of looseness functions have the referential transparency property which is discussed in [Søndergaard&88, Søndergaard&90]. In VDM-SL, functions are given an under-determined semantics, whereas operations are given a non-deterministic semantics[4]. The complexity of an arbitrary combination of these can be found in [Wieth89].

[4]The kind of non-determinism used in operations in VDM-SL using the terminology from the referenced papers is strong, unbounded, erratic and loose non-determinism with singular binding.

The difference between using the classical Hilbert epsilon operator [Leisenring69], the under-determinedness semantics and the non-deterministic semantics can be illustrated by a few examples. The expression:

$$(\lambda\, \mathbf{v}\colon \mathbf{N} \cdot \textbf{let } \mathbf{x_n}\{1,2\} \textbf{ in } \mathbf{x})(5) = (\lambda\, \mathbf{v}\colon \mathbf{N} \cdot \textbf{let } \mathbf{x_n}\{1,2\} \textbf{ in } \mathbf{x})(5)$$

will be **true** in the Hilbert framework (using epsilon for the let-be expression) because the two choices from the same set must yield the same result. If we instead have two non-deterministic choices from the set, the comparison yields a non-deterministic choice between **true** and **false**. Considering the under-determined interpretation, it is different we cannot use the same approach as Hilbert; choices in different parts of a program might be implemented differently even if they are made from sets which are equal[5]. However, due to the nature of the under-determined semantics this is not a set of constant evaluators, since the choice of the resulting value (in this case either **true** or **false**) may of course depend on the argument environment, even when it does not depend syntactically upon the environment at all. Thus, the under-determined meaning is:

The difference between non-deterministic and under-determined semantics can be illustrated by another example. Consider the expression:

$$(\lambda\, \mathbf{f}\colon \mathbf{N} \to \mathbf{N} \cdot \mathbf{f}(5) = \mathbf{f}(5))(\lambda\, \mathbf{v}\colon \mathbf{N} \cdot \textbf{let } \mathbf{x_n}\{1,2\} \textbf{ in } \mathbf{x})$$

It will yield **true** with the under-determined semantics because the two function applications yield the same result no matter which of the possible deterministic implementation of the function is considered. In a typical non-deterministic framework, non-deterministic implementations of the function would be allowed so the result would be a non-deterministic choice between **true** and **false**.

Also note that the first of the above examples is the result of β-reducing the second example. The two examples do, however, have different semantics, so β-reduction is *not* valid in general for VDM-SL functions. It is valid, however, if the semantics of the argument is a singleton set, e.g., if it does not contain any uses of the let-be expression (or other constructs where looseness is introduced).

Internal versus External Looseness

For some systems, the behaviour should not be too precisely determined by the specification. The notion of looseness enables the designer to postpone certain decisions to a later stage of development (e.g. the final implementation stage). In general, looseness can be seen as a mean to specify at a much higher level of abstraction than that of a final implementation, and in this way leave as much freedom as possible to the implementor.

The kind of looseness presented in the examples above is known as external looseness [Hayes&89] because the external behaviour of these expressions is not fully determined. This kind of external looseness is commonly used when the external behaviour in some cases simply need to satisfy certain conditions which not necessarily restrict the result to a unique value. We use the term external looseness when it is visible at a given abstraction level that different behaviour is permitted for a given specification.

[5]The rationale behind this is that even in cases where two functions are syntactically identical it is desirable that such loose functions can be implemented independently of each other.

Another kind of looseness is known as internal looseness. This may be used when the external behaviour of a system is defined to be deterministic, but freedom in some of the components of a specification is desirable. Such freedom can be used by the implementor to develop more efficient implementations of a given system. An obvious example of this would be an allocation of additional storage in a computer system. As a user of such storage we do not care about its physical location as long as we can use it freely (and possibly release it again later). Design decisions about the storage management inside a larger system could be left open by using looseness, but it would (hopefully) not be visible in the external behaviour of our system. Note that the given abstraction level is essential for this distinction, because the actual allocation function naturally is externally loose (at the function level), but if we look at the system level (and consider the allocation function to be hidden inside) the looseness is only visible internally. The last example in Section 10 also illustrates internal looseness.

6 Semantics of Expressions

An environment associates identifiers with values. Expression evaluation can be described as replacing each identifier in the expression by its value in the environment, and computing the result. Thus, the value of an expression is dependent on the environment in which it is being evaluated. Therefore one could think that the type of the evaluation function for expressions (as used in [Monahan85]) should be:

$$\mathit{EvalExpr} : \mathit{Expr} \rightarrow \mathit{ENV} \rightarrow \mathit{VAL}$$

However, because expressions can be loose, an expression may possibly yield more than one value. A next attempt (used in [Arentoft&88] and [Larsen&89]) could be:

$$\mathit{EvalExpr} : \mathit{Expr} \rightarrow \mathit{ENV} \rightarrow \mathbb{P}(\mathit{VAL})$$

Unfortunately this leads to very serious problems with the least-fixed-point semantics for recursive loose definitions. Therefore, eventually the type of the evaluation function *EvalExpr* has to be changed to:

$$\mathit{EvalExpr} : \mathit{Expr} \rightarrow \mathbb{P}(\mathit{ENV} \rightarrow \mathit{VAL})$$

where the looseness has been abstracted 'one level up'. We can now talk about deterministic expression evaluators for which least fixed points can be found. An expression will denote a set of such expression evaluators, which we call a loose expression evaluator. Because this technique is used for all kinds of expressions we define:

$$\begin{aligned} \mathrm{EEval} &= \mathrm{ENV} \rightarrow \mathrm{VAL} \\ \mathrm{LEEval} &= \mathbb{P}(\mathrm{EEval}) \end{aligned}$$

Sub-functions of *EvalExpr* are defined for all expression kinds. These functions are all written in roughly the same style:

$$
\begin{array}{l}
\mathit{EvalAnExpr} : \mathit{AnExpr} \to \mathit{LEEval} \\
\mathit{EvalAnExpr}(\mathit{MkTag}(\text{`}\mathit{AnExpr}\text{'}, (\mathit{expr}_1,\ldots,\mathit{op},\ldots,\mathit{expr}_n))) \triangleq \\
\quad \{\, \lambda \mathit{env} \,.\, \textbf{let}\ \mathit{val}_1 = \mathit{ev}_1(\mathit{env}), \\
\qquad\qquad\qquad \ldots, \\
\qquad\qquad\qquad \mathit{val}_n = \mathit{ev}_n(\mathit{env})\ \textbf{in} \\
\qquad\qquad\quad \mathit{AnOp}(\mathit{val}_1,\ldots,\mathit{val}_n,\mathit{op}) \\
\quad \mathbb{B}\ \mathit{ev}_1 \in \mathit{EvalExpr}(\mathit{expr}_1),\ldots,\ \mathit{ev}_n \in \mathit{EvalExpr}(\mathit{expr}_n)\,\}
\end{array}
$$

where the function *AnOp* is the mathematical definition of the actual operation that occurs in *AnExpr*. *MkTag* is an abstract syntax level operator tagging syntactic constructs with the name of their 'types', *AnExpr* in this case.

7 Semantics of Let-Be-Such-That Expressions

Let-be-such-that expressions are one of the main sources of looseness in explicit expressions and therefore it is naturally interesting for the work presented here.

The concrete syntax of a *let-be-such-that expression* has the form:

let bind be st st in in

where **bind** is a binding of a general pattern to either a set value or a type, **st** is a predicate (a boolean expression), and **in** is an expression, involving the pattern identifiers from the pattern in **bind**. The expression denotes the value of the expression **in** in a context where the pattern from **bind** has been matched against either a value in the set from **bind** or against a value from the type in **bind**. However, only bindings where the predicate **st** yields **true** are considered.

Its formal semantics is defined as:

$$
\begin{array}{l}
\mathit{EvalLetBeSTExpr} : \mathit{LetBeSTExpr} \to \mathit{LEEval} \\
\mathit{EvalLetBeSTExpr}(\mathit{MkTag}(\text{`}\mathit{LetBeSTExpr}\text{'}, (\mathit{bind},\ \mathit{st},\ \mathit{in}))) \triangleq \\
\quad \mathit{PropE}(\{\, \lambda \mathit{env}.\textbf{let}\ \mathit{venv\_s} = \bigcup\{\, \mathit{bev}(\mathit{env}) \mid \mathit{bev} \in \mathit{EvalBind}(\mathit{bind})\,\}\ \textbf{in} \\
\qquad\qquad \textbf{let}\ \mathit{venv\_s}' = \{\, \mathit{venv} \mid \mathit{venv} \in \mathit{venv\_s} \cdot \\
\qquad\qquad\qquad\qquad \mathit{venv} \neq \underline{\mathrm{err}} \wedge \mathsf{not}\mathit{BotEnv}(\mathit{venv}) \wedge \\
\qquad\qquad\qquad\qquad \mathit{stev}(\underline{\mathrm{overwrite}}(\mathit{env},\ \mathit{venv})) = \mathit{True}()\,\}\ \textbf{in} \\
\qquad\quad \{\, \mathit{inev}(\underline{\mathrm{overwrite}}(\mathit{env},\ \mathit{venv})) \mid \mathit{venv} \in \mathit{venv\_s}'\,\} \\
\qquad \mathbb{B}\ \mathit{stev} \in \mathit{EvalExpr}(\mathit{st}),\ \mathit{inev} \in \mathit{EvalExpr}(\mathit{in})\,\})
\end{array}
$$

Notice how the definition of *EvalLetBeSTExpr* follows the general scheme described above. The main difference is that here it is necessary to propagate the looseness out one level using the auxiliary function **PropE**. Looseness can be present in all three components (**bind**, **st**, and **in**). All possible combinations of looseness from the two expressions **st** and **in** are taken into account by indexing over them in a set comprehension expression. The looseness in the binding is removed by taking the distributed union of all binding evaluators applied with a given environment. The resulting environments are then reduced to those where the

expression evaluator for the **st** predicate yields true[6]. The (simplified) type of the evaluation function for bindings is:

$$EvalBind : Bind \to \mathbb{P}(ENV \to \mathbb{P}(\mathbb{M}(Id,\ VAL)))$$

Finally, the expression evaluator for the **in** expression must be applied with all the different binding environments (extending the given environment). It is this set of values which must be propagated up one level.

8 Semantics of Lambda Expressions

The definition of the semantics of lambda expressions is crucial for the combination of looseness and recursion and therefore we will show its semantics here.

The concrete syntax of a *lambda expression* has the form:

$\lambda\,\mathbf{pat}\colon\mathbf{type}\cdot\mathbf{body}$

where **pat** is a general pattern, **type** is a type expression and **body** is the body expression of the function. The scope of the pattern identifiers from **pat** is the body expression. This corresponds exactly to lambda expressions from (typed) lambda calculus except that the formal parameter can be a general pattern[7] instead of a simple identifier.

A simplified version (assuming that types are well-defined, values are not tagged, etc.) of the formal definition from [ISOVDM93] can be presented as:

$$\begin{array}{l}
EvalLambda : Lambda \to LEEval \\
EvalLambda(MkTag(\text{`Lambda'},\ (MkTag(\text{`Par'},\ (id,\ tp)),\ body))) \triangleq \\
\quad \textbf{let}\ quasi_eval = \lambda env\ .\ \textbf{let}\ d = EvalType(tp)(env)\ \textbf{in} \\
\qquad\qquad \lambda ob \in |\mathbf{d}|\ .\ \{\,\textbf{if}\ ob \in \|\mathbf{d}\| \\
\qquad\qquad\qquad \textbf{then}\ ev(\underline{\text{overwrite}}(env,\ \{\,id \mapsto ob\,\})) \\
\qquad\qquad\qquad \textbf{else}\ \bot \\
\qquad\qquad \mathbb{B}\ ev \in EvalExpr(body)\,\}\ \textbf{in} \\
\quad \{\,eeval \mid eeval \in EEval \cdot \\
\qquad \forall\ env \in ENV \cdot \\
\qquad\quad \delta_0(eeval(env)) = |\mathbf{EvalType(tp)(env)}| \wedge \\
\qquad\quad \forall ob \in |\mathbf{EvalType(tp)(env)}| \cdot eeval(env)(ob) \in quasi_eval(env)(ob))\,\}
\end{array}$$

A "quasi"-evaluator for the lambda expression is first produced. It has the type: $ENV \to \|\mathbf{d}\| \to \mathbb{P}(VAL)$ where d is the denotation of tp in the given environment. This is not a "real" expression evaluator because the looseness of the body has not been propagated out. The *EvalLambda* function yields all possible expression evaluators which yield proper functions with the domain corresponding to the given type and a body which yields one of the possible values returned by the functions in the quasi-evaluator. Note how this ensures that

[6] In addition it is ensured that the binding does not contain any errors or bindings to the bottom value (which is a proper value for the dynamic semantics definition).

[7] This generality is however not taken into account in the simplified version of the semantics presented below.

there is a functional dependency for all choices which are present in the body of the function such that models exist where different choices are taken for different arguments.

9 Semantics of Mutually Recursive Function Definitions

In this section we will show how a least fixed point semantics is obtained for recursive function definitions which contain looseness[8]. The *EvalExplDef* function is used by the function which gives meaning to the entire collection of function definitions from a specification.

$$EvalExplDef : \mathbb{M}(Name,\ ExplFnDef) \rightarrow LDef$$

$$\begin{array}{l}
EvalExplDef(def_m) \triangleq \\
\quad \textbf{let } id_s = \underline{\mathrm{dom}}(def_m) \textbf{ in} \\
\quad\quad \textbf{let } tev_m = \{\, id \mapsto \lambda env.\ EvalType(GetType(def_m(id))(env)) \mid id \in id_s \} \textbf{ in} \\
\quad\quad\quad \textbf{let } lev_m = \{\, id \mapsto EvalExpr(SelBody(def_m(id))) \mid id \in id_s \} \textbf{ in} \\
\quad\quad\quad\quad \textbf{let } ev_m_s = \{\, m \mid m \in \mathbb{M}(Id,\ EEval) \cdot \\
\qquad\qquad\qquad\qquad \underline{\mathrm{dom}}(m) = id_s \wedge \\
\qquad\qquad\qquad\qquad \forall id \in id_s \cdot m(id) \in lev_m(id) \} \textbf{ in} \\
\quad\quad\quad\quad\quad \{\, \lambda env.\ \textbf{let } d_m = \{\, id \mapsto tev_m(env) \mid id \in id_s \ \} \textbf{ in} \\
\qquad\qquad\qquad \textbf{if } \underline{\mathrm{err}} \in \{\, d_m(id) \mid id \in id_s \} \\
\qquad\qquad\qquad \textbf{then } \underline{\mathrm{err}} \\
\qquad\qquad\qquad \textbf{else let } Appr = \{ f \mid f : id_s \rightarrow VAL \cdot \\
\qquad\qquad\qquad\qquad\qquad \forall id \in id_s \cdot f(id) \in |\mathbf{d_m(id)}| \} \textbf{ in} \\
\qquad\qquad\qquad\qquad \textbf{let } tf = \lambda appr \in Appr.\ \{ \textbf{let } ev = ev_m(id), \\
\qquad\qquad\qquad\qquad\qquad\qquad env' = \underline{\mathrm{overwrite}}(env,\ appr) \\
\textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad id \mapsto ev(env') \\
\qquad\qquad\qquad\qquad\qquad \mathbb{B}\ id \in id_s \} \textbf{ in} \\
\qquad\qquad\qquad\qquad \textbf{if } IsCont_{Appr}(tf) \\
\qquad\qquad\qquad\qquad \textbf{then let } lfp_m = \mathsf{Y}(tf) \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad \underline{\mathrm{overwrite}}(env,\ lfp_m) \\
\qquad\qquad\qquad\qquad \textbf{else } \underline{\mathrm{err}} \\
\quad\quad\quad\quad\quad \mathbb{B}\ ev_m \in ev_m_s \}
\end{array}$$

First we take the set of function identifiers occurring in the definition. Then, for every identifier we take an appropriate type evaluator corresponding to the functional type of the identifier. This gives us the mapping *tev_m*. Next *EvalExpr* is used to deal with the bodies of all the function definitions. Thus, *lev_m* is a mapping which for every identifier assigns its loose expression evaluator. In order to find the meaning for the whole collection of mutually recursive definitions, it is necessary to first create all possible combinations of the expression evaluators for the different functions. As a result the set of maps *ev_m_s* is constructed. It consists of maps from identifiers to deterministic expression evaluators. For each of these "deterministic" maps a least fixed point iteration will be carried out.

For every "deterministic" map *ev_m* we define an appropriate evaluator. For a given environment *env* we have to first evaluate the type evaluator mapping *tev_m*, obtaining the

[8] Mutually recursive value definitions follow the same strategy.

"domain mapping" *d_m*. If for every identifier *id* $d\_m(id)$ gives us a sensible domain we can define an appropriate approximation set *Appr*. If the evaluation process for *tev_m* fails, an error is returned as a result.

The set *Appr* consists of all mappings from the set of identifiers *id_s* to the universe of values *VAL*, which respect the types of the identifiers. The expression $|d\_m(id)|$ used in the definition of *Appr* denotes the set of values of the VDM domain $d\_m(id)$ corresponding to the type of the identifier *id*. The set *Appr* has a structure of a cpo with an ordering $\sqsubseteq$ defined as follows:

$$\text{for} \quad f, g \in Appr \quad \text{it holds that} \quad f \sqsubseteq g \quad \text{iff} \quad \forall id \in id\_s \cdot f(id) \sqsubseteq_{d\_m(id)} g(id).$$

In the above definition $\sqsubseteq_{d\_m(id)}$ denotes the ordering on the VDM domain corresponding to the type of the identifier *id*. The least element in the cpo *Appr* is the function "$\lambda id \in id\_s.\ \bot$".
Having the "approximation" cpo *Appr* we define a functional $tf : Appr \to Appr$ corresponding to the given "deterministic" evaluator map *ev_m*. If *tf* is continuous with respect to the ordering on *Appr* then the least fixed point *lfp_m* is calculated. Otherwise an error is returned. Finally, the original environment *env* is updated with *lfp_m*.

The whole "trick" is that this construction is indexed by all possible interpretations of lambda expressions occurring as the body of the function definitions. Therefore the least fixed point is found for each individual deterministic expression evaluator.

10 Examples

The key motivation for using an under-determined interpretation of looseness is that functional values then act as real mathematical functions which always (deterministically) return the same result when they are applied to the same argument. This also holds even if the body of the function is loose; this will simply give rise to different evaluators (i.e. different models). In order to illustrate how the choices made in the body of a lambda expression are "implicitly parameterised" by its argument, we will present two examples of adding underdeterminedness to the well-known factorial function. The examples are written in the actual VDM-SL syntax, which in many aspects is very similar to our meta-language syntax. We hope it would not lead to confusion.

$$\begin{array}{l}
\mathbf{fac'}: \mathbf{N} \to \mathbf{N} \\
\mathbf{fac'}\,(\mathbf{n}) \triangleq \\
\quad \textbf{if } \mathbf{n} = 0 \\
\quad \textbf{then let } \mathbf{x_n}\{1,2\} \textbf{ in } \mathbf{x} \\
\quad \textbf{else } \mathbf{n} \times \mathbf{fac'}\,(\mathbf{n}-1)
\end{array}$$

The function **fac′** denotes a set of two evaluators (because the choice is only made in the base case, i.e., at the "bottom" of the recursion):

$$\left\{ \begin{array}{l} \lambda\,\rho \cdot \lambda\,\mathbf{n} \cdot \left\{ \begin{array}{lcl} \mathbf{n} = 0 & \Rightarrow & 1 \\ \mathbf{n} > 0 & \Rightarrow & \mathbf{n} \times \rho(\mathbf{fac'})(\mathbf{n}-1), \end{array} \right. \\ \lambda\,\rho \cdot \lambda\,\mathbf{n} \cdot \left\{ \begin{array}{lcl} \mathbf{n} = 0 & \Rightarrow & 2 \\ \mathbf{n} > 0 & \Rightarrow & \mathbf{n} \times \rho(\mathbf{fac'})(\mathbf{n}-1) \end{array} \right. \end{array} \right\}$$

The result of taking the least fixed point of these two evaluators is that $\mathbf{fac}'$ denotes a set of 2 least fixed points, one corresponding to the normal factorial function and the other corresponding to twice the normal one. In the fixed point induction process for each evaluator, the environment ρ will gradually contain more and more defined versions of $\mathbf{fac}'$ ending with the least fixed point. Thus, in this case, one obtains two incomparable least fixed points. This means that, for example, $\mathbf{fac}'(4)$ would yield 24 (4!) in one model and 48 ($2 \times 4!$) in the other model.

Let us now consider a slightly more complicated example:

$$
\begin{array}{l}
\mathbf{fac}'' \colon \mathbf{N} \to \mathbf{N} \\
\mathbf{fac}''\,(\mathbf{n}) \triangleq \\
\quad \textbf{let } \mathbf{x_n}\{1,2\} \textbf{ in} \\
\quad \textbf{if } \mathbf{n} = 0 \\
\quad \textbf{then } \mathbf{x} \\
\quad \textbf{else } \mathbf{x} \times \mathbf{n} \times \mathbf{fac}''\,(\mathbf{n}-1)
\end{array}
$$

Notice how the choice of $\mathbf{x}$ can be made on each recursive application of $\mathbf{fac}''$. Therefore, the choice of $\mathbf{x}$ depends on the actual value of the formal parameter $\mathbf{n}$ (i.e. $\mathbf{fac}''(2)$ may choose to bind $\mathbf{x}$ to 1 while $\mathbf{fac}''(1)$ may choose to bind $\mathbf{x}$ to 2 in the same model).

In the denotational semantics, this function will denote an infinite set of expression evaluators of the form:

$$
\left\{
\begin{array}{l}
\lambda\,\rho \cdot \lambda\,\mathbf{n} \cdot \left\{ \begin{array}{lcl} \mathbf{n} = 0 & \Rightarrow & 1 \\ \mathbf{n} > 0 & \Rightarrow & \mathbf{n} \times \rho(\mathbf{fac})(\mathbf{n}-1), \end{array} \right. \\
\lambda\,\rho \cdot \lambda\,\mathbf{n} \cdot \left\{ \begin{array}{lcl} \mathbf{n} = 0 & \Rightarrow & 2 \\ \mathbf{n} > 0 & \Rightarrow & \mathbf{n} \times \rho(\mathbf{fac})(\mathbf{n}-1), \end{array} \right. \\
\lambda\,\rho \cdot \lambda\,\mathbf{n} \cdot \left\{ \begin{array}{lcl} \mathbf{n} = 0 & \Rightarrow & 2 \\ \mathbf{n} = 1 & \Rightarrow & 4 \\ \mathbf{n} > 1 & \Rightarrow & \mathbf{n} \times \rho(\mathbf{fac})(\mathbf{n}-1), \end{array} \right. \\
\ldots \\
\lambda\,\rho \cdot \lambda\,\mathbf{n} \cdot \left\{ \begin{array}{lcl} \mathbf{n} = 0 & \Rightarrow & 1 \\ \mathbf{n} > 0 & \Rightarrow & 2 \times \mathbf{n} \times \rho(\mathbf{fac})(\mathbf{n}-1), \end{array} \right. \\
\lambda\,\rho \cdot \lambda\,\mathbf{n} \cdot \left\{ \begin{array}{lcl} \mathbf{n} = 0 & \Rightarrow & 2 \\ \mathbf{n} > 0 & \Rightarrow & 2 \times \mathbf{n} \times \rho(\mathbf{fac})(\mathbf{n}-1) \end{array} \right. \quad \right\}
\end{array}
$$

Each element, $\mathbf{f}$, in this infinite set of functions satisfies $\mathbf{f}(\mathbf{n}) = 2^{\mathbf{k}} \times \mathbf{n}!$ for some $\mathbf{k_n}\{0, \ldots, \mathbf{n}+1\}$, where "!" is used as a primitive operator as in traditional mathematics. Note that the value of $\mathbf{k}$ may depend on $\mathbf{n}$ for each of the functions. Thus we really have an infinite set of functions despite the relatively 'simple' expression used here. The first distinguishable evaluator listed above chooses 1 every time, whereas the second evaluator chooses 2 for $\mathbf{n} = 0$ and 1 otherwise. The third evaluator chooses 2 for $\mathbf{n} = 0$ and $\mathbf{n} = 1$ and 1 otherwise and the last evaluators which are written in the collection are ones where 2 is chosen (almost) all the time. So there will, for example, be models where $\mathbf{fac}''(2)$ is 2 (2!), 4 ($2 \times 2!$), 8 ($2 \times 2 \times 2!$) and 16 ($2 \times 2 \times 2 \times 2!$), respectively. Note that there is an infinite number of different models (evaluators), but for a given argument value $\mathbf{v}$ there will only be a finite set of possible results (the combination of choices made for all arguments less than $\mathbf{v}$).

Finally, let us consider an example where only internal looseness is present. Below is the

specification of a recursive function which takes a set of natural numbers and yields the sum of these numbers.

$$\textbf{Add}: \textbf{N-set} \to \textbf{N}$$

$$\begin{array}{l}\textbf{Add}\,(\textbf{s}) \triangleq \\ \quad \textbf{if } \textbf{s} = \{\} \\ \quad \textbf{then } 0 \\ \quad \textbf{else let } \textbf{e}_\textbf{n}\textbf{s} \textbf{ in} \\ \qquad \textbf{e} + \textbf{Add}\,(\textbf{s} \setminus \{\textbf{e}\})\end{array}$$

No matter which order the elements from the set, **s**, are chosen, **Add** will yield the same result. Therefore all expression evaluators will yield the same result and we will only have one least fixed point in this case.

11 Concluding Remarks

From a standards point of view it is interesting how the VDM-SL standard (and also the forthcoming Z standard [BSIZ92]), focusing on semantic issues, have become more precise than standards normally have been in the past. We believe that specifications languages like these will be a valuable tool to enhance the quality and preciseness of future standards. This has already been shown for a few programming language standards (e.g. [ISO/Modula2]).

This formal definition is now complete, and the final version in the ISO standard is not expected to change very much. It is worth noting however, that the VDM-SL language is large and complex and therefore the size of the dynamic semantics is over 100 pages. The standard provides meaning to flat VDM-SL specifications. Thus, future work is scheduled as a new standardisation work item on deciding on the structuring mechanism to adopt for VDM-SL (a number of different proposals already exists).

In this paper we have provided an overview of the formal semantics of ISO VDM-SL. It has focused in particular on the semantics for the combination of under-determinedness and recursion in specifications. This combination seems to be particularly interesting since it turns out that it is possible to automate the analysis of such specifications [Larsen94a] and to find corresponding proof rules [Larsen&94b].

Acknowledgements

We would like to acknowledge all the input received from the standardisation panel and from the review boards. We would also like to thank John Fitzgerald, Paul Mukherjee, Søren Prehn and Andrzej Tarlecki for constructive comments on the contents of this paper.

Bibliography

[Andrews&88] Derek Andrews. Report from the BSI Panel for the Standardization of VDM. In R. Bloomfield, L. Marshall and R. Jones, editors, *VDM '88 VDM – The Way Ahead*, pages 74–78, VDM-Europe, Springer-Verlag, September 1988.

[Arentoft&88] Michael Meincke Arentoft and Peter Gorm Larsen. *The Dynamic Semantics of the BSI/VDM Specification Language*. Master's thesis, Technical University of Denmark, DK-2800 Lyngby, Denmark, August 1988.

[BSIZ92] (editors) Stephen Brien and John Nicholls. *Z Base Standard*. Version 1.0, Oxford University, Computer Laboratory, Programming Research Group, November 1992.

[Hayes&89] I.J. Hayes, C.B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 330–338, November 1989.

[ISO/Modula2] *JCT1/SC22/WG13 ISO/IEC DIS 10514. Information technology – Programming languages – Modula-2.*

[ISOVDM93] *Information Technology Programming Languages – VDM-SL.* Technical Report, First Committee Draft Standard: CD 13817-1, November 1993. ISO/IEC JTC1/SC22/WG19 N-20.

[Larsen&89] Peter Gorm Larsen, Michael Meincke Arentoft, Brian Monahan and Stephen Bear. Towards a Formal Semantics of The BSI/VDM Specification Language. In Ritter, editor, *Information Processing 89*, pages 95–100, North-Holland, August 1989.

[Larsen94a] Peter Gorm Larsen. Evaluation of Underdetermined Explicet Expressions. In M. Naftalin, T. Denvir, M. Bertran, editor, *FME'94: Industrial Benefit of Formal Methods*, pages 233–250, Springer-Verlag, October 1994.

[Larsen&94b] Peter Gorm Larsen and Bo Stig Hansen. Semantics for Underdetermined Expressions. *Accepted for publication by "Formal Aspects of Computing"*, January 1995.

[Leisenring69] A.C. Leisenring. *Mathematical Logic and Hilbert's ε-Symbol*. Gordon and Breach Science Publishers, New York, 1969.

[Monahan85] Brian Q. Monahan. A Semantic Definition of the STC VDM Reference Language. November 1985. 176 pages. Doc. no. 9.

[Parkin94] Graeme I. Parkin. Vienna Development Method Specification Language (VDM-SL). *Computer Standard & Interfaces*, 16:527–530, 1994. Special issue with the general title: The programming language standards scene, ten years on.

[Plat&92] Nico Plat and Peter Gorm Larsen. An Overview of the ISO/VDM-SL Standard. *Sigplan Notices*, 27(8):76–82, August 1992.

[Schmidt86] David A. Schmidt. *Denotational Semantics – A Methodology for Language Development*. Allyn and Bacon, Inc. 1986.

[Sen87] Dev Sen. Objectives of the British Standardization of a Language to Support the VDM. In Bjørner, Jones, Airchinnigh and Neuhold, editors, *VDM '87 VDM – A Formal Method at Work*, pages 321–323, VDM-Europe, Springer-Verlag LNCS 252, 1987.

[Søndergaard&87] Harald Søndergaard and Peter Sestoft. *Non-Determinacy and Its Semantics.* Technical Report 86/12, DIKU, Datalogisk Institut, Københavns Universitet, Sigurdsgade 41, DK-2200 København N, 1987.

[Søndergaard&88] Harald Søndergaard and Peter Setoft. *Referential Transparancy and Allied Notions.* Technical Report 88/7, DIKU, Datalogisk Institut, Københavns Universitet, Sigurdsgade 41, DK-2200 København N, 1988.

[Søndergaard&90] Harald Søndergaard and Peter Sestoft. Referential Transparency, Definiteness and Unfoldability. *Acta Informatica*, 27:505–517, 1990.

[Søndergaard&92] Harald Søndergaard and Peter Sestoft. Non-determinism in Functional Languages. *The Computer Journal*, 35(5):514–523, October 1992.

[Tarlecki&90] Andrzej Tarlecki and Morten Wieth. A Naive Domain Universe for VDM. In Dines Bjørner, C.A.R. Hoare and Hans Langmaack, editors, *VDM '90 VDM and Z – Formal Methods in Software Development*, pages 552–579, VDM Europe, Springer-Verlag, April 1990.

[Wieth89] Morten Wieth. Loose Specification and its Semantics. In G.X. Ritter, editor, *Information Processing 89*, pages 1115–1120, North-Holland, August 1989.