Evaluation of Underdetermined Explicit Definitions[⋆]

Peter Gorm Larsen

The Institute of Applied Computer Science
Forskerparken 10
DK – 5230 Odense M
Denmark

Abstract. The specification language VDM-SL is used in the model-oriented formal method known as the Vienna Development Method. There have been a number of different dialects of this language, but a standard for the language has now been defined. The draft standard includes a dynamic semantics which, in general, is not executable. A given specification denotes a (possibly infinite) set of models according to the dynamic semantics. This paper illustrates how all the valid results from evaluating a given expression in all the different models can be collected. It is important to be able to do this in order to investigate the amount of looseness which is present in the given specification, such that correctness properties of the expression as a whole can be derived. We present a specification of such a loose evaluation for an executable subset of VDM-SL. The subset is interesting because it also illustrates how underdeterminedness is combined with recursion in VDM-SL.

1 Introduction

Formal specification languages often provide means of specification not usually found in programming languages. One of these is the possibility of writing down expressions whose value is not fully determined. Such 'loose specification' may be a convenient way of expressing that a number of alternative implementations are allowed.

There are (at least) two different ways of interpreting loose specifications. In VDM-SL they are called: 'underdeterminedness'[1] (allowing several different deterministic implementations) and 'nondeterminism' (allowing non-deterministic implementations). This paper will only deal with loosely specified constructs which are interpreted as being underdetermined. Thus, the implementer must decide at implementation time which functionality to use. Since this choice is static the final implementation becomes deterministic. We focus on this part because it turns out that the underdeterminedness in explicit definitions may

[⋆] Interested readers may obtain a copy of the full version of the specification given in this paper via anonymous ftp. Site: hermes.ifad.dk (130.225.136.3) Directory: /pub/vdm/examples/loose.tar.gz or alternatively directly from the author.

[1] In the literature 'underdeterminedness' has also been called 'under-specification'.

give rise to complicated proof rules and proofs (see [5])[2]. The loose expression evaluation functions described here may help to simplify the task of proving properties about expressions which are loose, as a kind of decision procedure which can reduce the simple (executable) expressions to a corresponding "loose value".

In this paper a specification of loose evaluation of expressions is presented for a non-trivial subset of VDM-SL. The definition given here is operational and closely related to the dynamic semantics definition used for developing the interpreter from the IFAD VDM-SL Toolbox [4][3]. The main motivation for using an operational style here is to illustrate how small the changes are when one moves from using an arbitrary model to collecting all models. The definition presented here illustrates what the valid implementations should yield for an arbitrary expression in a given context (a specification). This information can be used in different ways:

- Imagine that one had evaluated[4] an expression in one of the models for the given specification. This would yield one of the values allowed by the specification. It would then be possible that the implementer (of the specification) had chosen a different model yielding a different result. The loose evaluation described here could then be used to find out whether another model exists where the result is identical to the implementation.
- Alternatively if one is proving properties about a (potentially loose) executable expression, trivial reductions of this expression to its (possibly loose) value could be carried out by this kind of loose evaluation[5].
- When one would like to prove general properties about (potentially loose) recursive functions, the information could be used to indicate whether the property holds. In cases where it does not hold, this approach will be able to save the specifier time.

It is also important to derive information regarding the presence of both external looseness (where different models yield different results) and internal looseness (where each model yields the same result). In most cases the specifier may want the looseness to be internal to some extent, and the definition given here provides the specifier with more confidence as to whether this actually holds.

After this introduction we explain what the problems with underdeterminedness are, and show how the semantics of undeterminedness combine with recursion in VDM-SL. We then proceed with an overview of the abstract syntax of the selected subset. This is followed by three sections about the evaluation of underdetermined explicit definitions. Finally some examples and some concluding remarks are given.

[2] These problems mainly have to do with the fact that referential transparency must be kept for loose functions when they are interpreted as being underdetermined.

[3] Notice that the 'meta-language' used for this definition is VDM-SL itself. Thus, this definition cannot be expanded to the full VDM-SL language because we would run into 'meta-circular' problems.

[4] For example by using the interpreter from the IFAD VDM-SL Toolbox [6, 2].

[5] In the same way as decision procedures are used in the PVS system [8].

2 The Problems with Underdeterminedness

As mentioned in the introduction above, there are different ways of interpreting looseness (see [9, 10] for general overviews of alternative models). When looseness is interpreted as being underdetermined it means that even functions which contain looseness in the body are deterministic in each model. Thus, in any model for a specification it will always hold that for all elements, x, in the domain of a function, f (which also means that it must satisfy the pre-condition of the function), $f(x) = f(x)$ yields true. However, the general substitution law (replacing equals for equals) does not in general hold in the presence of underdeterminedness. A simple expression like:

$$\mathsf{let}\ a \in \{1, 2\}\ \mathsf{in}\ a$$

has two different models. One where a is bound to 1 (and the entire expression yields 1) and one where a is bound to 2 (where the entire expression yields 2). Comparing underdetermined expressions like this:

$$(\mathsf{let}\ a \in \{1, 2\}\ \mathsf{in}\ a) = (\mathsf{let}\ a \in \{1, 2\}\ \mathsf{in}\ a)$$

there will be models where this expression yields false (and there will be models where it yields true).

In addition, the looseness present in the body of a function may depend upon the argument. Thus for a function like:

$$\begin{array}{l} f : \mathbb{B} \to \mathbb{N} \\ f\,(b) \triangleq \\ \quad \mathsf{let}\ a \in \{1, 2\}\ \mathsf{in}\ a \end{array}$$

models will exist where $f(\mathsf{true})$ is different from $f(\mathsf{false})$. This means that the traditional unfolding rule for function definitions must be used with caution.

This is the kind of problem which the underdetermined interpretation of looseness gives, and which will be dealt with in this paper. In particular we will investigate how this influences the semantics of recursive functions which have looseness in their body.

3 The Semantics of Underdeterminedness in VDM-SL

Explicitly defined recursive functions in VDM-SL are given a least fixed point semantics (see [7]). However, in the presence of underdeterminedness more than one least fixed point will be present. Thus, we need some extra explanation about how this is dealt with.

In the denotational dynamic semantics of VDM-SL given in the standard [1], the functions which give meaning to expressions yield a loose expression evaluator. This is a set of expression evaluators, each of which is a deterministic function from environments to values. Thus, if an expression evaluator is given an environment (corresponding to a model) it will yield one value (corresponding to the denotation of the expression in this model). So when recursion is combined with looseness it is important that the least fixed point is found for each different expression evaluator. Thus, the looseness has already been distributed on the different deterministic expression evaluators, and for each of these a fixed point induction is made. The result is that a recursive function denotes a set of (incomparable) least fixed points corresponding to the different choices which can be taken in the body of the function.

This can be illustrated by a couple of small examples[6]. Let us first consider:

```
$fac' : \mathbb{N} \to \mathbb{N}$
$fac'(n) \triangleq$
  if $n = 0$
  then let $x \in \{1, 2\}$ in $x$
  else $n \times fac'(n-1)$
```

In the dynamic semantics in the standard, this function will denote a set with two expression evaluators (because the choice is only made at the bottom of the recursion):

$$\left\{ \lambda\, env \cdot \lambda\, n \cdot \begin{cases} n = 0 \Rightarrow 1 \\ n > 0 \Rightarrow n \times fac'(n-1), \end{cases} \right.$$
$$\left. \lambda\, env \cdot \lambda\, n \cdot \begin{cases} n = 0 \Rightarrow 2 \\ n > 0 \Rightarrow n \times fac'(n-1) \end{cases} \right\}$$

The result of taking the least fixed point of these two expression evaluators is that fac' denotes a set of 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 of each evaluator, env will gradually contain more and more defined versions of fac' ending with the least fixed point. Thus in this case, one obtains two incomparable least fixed points. This means that for example $fac'(4)$ 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:

```
$fac'' : \mathbb{N} \to \mathbb{N}$
$fac''(n) \triangleq$
  let $x \in \{1, 2\}$ in
  if $n = 0$
  then $x$
  else $x \times n \times fac''(n-1)$
```

Notice how the choice of x can be made on each recursive call of fac''. Therefore, the choice of x depends on the actual value of the formal parameter n (i.e. $fac''(2)$

[6] We chose to let the examples be centered around the well-known factorial function. They are chosen specially to illustrate the principles of the semantics of the combination of underdeterminedness and recursion.

may chose to bind x to 1 while $\mathit{fac}''(1)$ may chose to bind x to 2 in the same model).

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

$$
\begin{aligned}
\{\ & \lambda\, env \cdot \lambda\, n \cdot \begin{cases} n = 0 \Rightarrow 1 \\ n > 0 \Rightarrow n \times \mathit{fac}''(n-1), \end{cases} \\
& \lambda\, env \cdot \lambda\, n \cdot \begin{cases} n = 0 \Rightarrow 2 \\ n > 0 \Rightarrow n \times \mathit{fac}''(n-1), \end{cases} \\
& \lambda\, env \cdot \lambda\, n \cdot \begin{cases} n = 0 \Rightarrow 2 \\ n = 1 \Rightarrow 4 \\ n > 1 \Rightarrow n \times \mathit{fac}''(n-1), \end{cases} \\
& \ldots \\
& \lambda\, env \cdot \lambda\, n \cdot \begin{cases} n = 0 \Rightarrow 1 \\ n > 0 \Rightarrow 2 \times n \times \mathit{fac}''(n-1), \end{cases} \\
& \lambda\, env \cdot \lambda\, n \cdot \begin{cases} n = 0 \Rightarrow 2 \\ n > 0 \Rightarrow 2 \times n \times \mathit{fac}''(n-1) \end{cases} \}
\end{aligned}
$$

This infinite set contains all functions for which the least fixed points can be described by[7]: $\lambda\, n \cdot 2^{n+1-k} \times \mathit{fac}(n)$ where fac is the normal factorial function and $0 \leq k \leq n+1$. The first evaluator listed chooses 1 every time, whereas the second evaluator chooses 2 for $n = 0$ and 1 otherwise. The third evaluator chooses 2 for $n = 0$ and $n = 1$ and 1 otherwise and the last evaluators which are written in the set are ones where 2 is chosen (almost) all the time. In general there will be many evaluators which have the same least upper bound. In this particular case each evaluator which chooses 2 once and 1 all other times will have the least upper bound corresponding to twice the normal factorial function. However, notice that even though these have the same least upper bound they are incomparable least fixed points (they will have different graphs because of the different places where 2 was chosen). Here it means that $\mathit{fac}''(2)$ may yield 2 (2!), 4 ($2 \times 2!$), 8 ($2 \times 2 \times 2!$) and 16 ($2 \times 2 \times 2 \times 2!$) in the different models for fac''.

4 Constructs in the Selected Subset

Having illustrated the semantics of the combination of underdeterminedness and recursion we will now present the abstract syntax we have selected to illustrate the principles of evaluating such expressions. The abstract syntax presented in this section is a simplified extract of the outer abstract syntax from the draft VDM-SL Standard.

[7] Note that the value of k may depend on n for each of the functions. Thus we really have an infinite set of functions despite the relatively "simple" expression used here to express it.

4.1 Definitions

In this paper we only consider value definitions and explicit function definitions.

$$\begin{aligned}\mathit{Definitions} :: \mathit{valuem} &: \mathit{ValueDef}^* \\ \mathit{fnm} &: \mathit{Name} \xrightarrow{m} \mathit{ExplFnDef}\end{aligned}$$

4.2 Value Definitions

The value definitions simply consist of a left hand side pattern and a right hand side expression.

$$\begin{aligned}\mathit{ValueDef} :: \mathit{pat} &: \mathit{Pattern} \\ \mathit{val} &: \mathit{Expr}\end{aligned}$$

4.3 Function Definitions

The abstract syntax of a function which is needed for the looseness analysis here is composed of the name of the function, the argument pattern and the body expression. Thus, information such as the type of the function is not needed in this context.

$$\begin{aligned}\mathit{ExplFnDef} :: \mathit{nm} &: \mathit{Name} \\ \mathit{pat} &: \mathit{Pattern} \\ \mathit{body} &: \mathit{Expr}\end{aligned}$$

4.4 Expressions

The expressions which are selected in the full version of this definition are restricted to let expressions (including let-be-such-that expressions), conditional expressions, apply expressions (only for function application), literals, names, bracketed expressions and (a few) binary expressions. However, in this paper we will only focus on two of these expression constructs to illustrate the principles. For this purpose, we have selected the let expression and the function application expression. The definition for the other expression constructs follow the same principle, and so have been omitted from this paper.

$$\mathit{Expr} = \mathit{LetExpr} \mid \mathit{ApplyExpr} \mid \ldots$$

$$\begin{aligned}\mathit{LetExpr} :: \mathit{lhs} &: \mathit{Pattern} \\ \mathit{rhs} &: \mathit{Expr} \\ \mathit{body} &: \mathit{Expr}\end{aligned}$$

$$\begin{aligned}\mathit{ApplyExpr} :: \mathit{fct} &: \mathit{Name} \\ \mathit{arg} &: \mathit{Expr}\end{aligned}$$

4.5 Patterns and Binds

The patterns selected in this paper are pattern identifiers (notice here how the position of the pattern has been incorporated as well[8]), and set patterns (for set enumeration and set union patterns). For bindings we are only considering set bindings because we only investigate an executable subset of VDM-SL here. We have also incorporated matching values in the full version of this definition, but we will not focus on this here.

$$Pattern = PatternName \mid SetPattern \mid MatchVal$$

$$PatternName :: nm : [(Name \times Position)]$$

$$Position = \mathbb{N} \times \mathbb{N}$$

$$SetPattern = SetEnumPattern \mid SetUnionPattern$$

$$SetEnumPattern :: Elems : Pattern^*$$

$$\begin{aligned} SetUnionPattern :: lp &: Pattern \\ rp &: Pattern \end{aligned}$$

5 Semantic Domains

In order for the loose evaluation to take place, the context in which the expression is to be evaluated must be initialized and maintained. This is taken care of by the definitions presented in this section.

5.1 The Evaluation Stack

The semantic domains describe the type of the structure which will be used for specifying the operational (loose) semantics for the abstract syntax.

$$ENVL = ENV^*$$

The main structure in the semantic domains is the environment, *ENVL* which is organized as a stack of so-called function application environments *ENV*. When a function is called, it must establish a local environment containing its own definitions such as the formal parameters.

$$ENV = BlkEnv^*$$

Expressions can define a local environment called a block environment *BlkEnv*. For example a let expression will produce a local environment for which the scope

[8] We will see later why this is essential, even though any other unique identification of the pattern name could be used.

is the body of the let-expression. The function application environment is therefore organized as a stack of block environments where the first block environment pushed on the stack contains the instantiation of the formal parameters. When the value of an identifier is looked up this will happen in a top-down manner down through the block environments.

$BlkEnv = NameVal^*$

Finally a block environment is a sequence of *NameVals*, each containing a unique identifier and its associated value. A *BlkEnv* could alternatively be modelled as a map. We have chosen to use a sequence in order to keep it in line with the operational dynamic semantics presented in [4].

$NameVal = UniqueId \times VAL$

$UniqueId = (Name \times Position \times ([Name \times VAL]))$

The unique names which are used here deserve a little additional explanation. The unique name contains the actual name (from a pattern identifier), the position (of the pattern identifier[9]) and if the binding takes place inside the body of a function, the function name and the argument (value) to the function are added (otherwise nil is used). This corresponds to the dynamic tagging information used in [5].

5.2 Semantic Values

$LVAL = (VAL \times Model)\text{-set}$

The "loose value" returned by the loose expression evaluation functions is a set of pairs. The first element of such a pair is the value of the expression in the model described by the second element of the pair. Thus, one gets information about both external looseness and internal looseness in the sense that more than one element will be present if any looseness is involved. If the value part is the same for each pair we simply have internal looseness while external looseness is used if different values are present.

$Model = UniqueId \xrightarrow{m} VAL$

A model contains a precise description of all the choices made underway in the loose evaluation of an expression. In this paper the values considered are simply numerical values, Boolean values and set values.

$VAL = NUM \mid BOOL \mid SET$

$NUM :: v : \mathbb{Z}$

[9] As already mentioned this could in principle simply be any other unique identification of the occurrence of the name.

$BOOL :: v : \mathbb{B}$

$SET :: v : VAL$-set

5.3 The State Definition

The state of this specification contains the environment described above, *env-l*, and additionally some information about the context of the expression (i.e. the value definitions, *val-m*, and function definitions, *fn-m*, it may make use of). The *curfn* component indicates whether we are currently evaluating an expression in the body of some function[10]. If this is the case, information about the function name and the argument value needs to be added to the unique identifiers which are created when new pattern identifiers are encountered. Finally the *fnparms* component contains the unique identification of the parameters of the functions. This information is important because the binding of these formal parameters does not constitute any input to the models. Thus, the binding of the formal parameters must **not** be taken into account in the choices taken in a given model. The reason for this will be clear at the end of the next section.

```
state $Sigma$ of
   $env\text{-}l$     : $ENVL$
   $val\text{-}m$     : $UniqueId \xrightarrow{m} LVAL$
   $fn\text{-}m$      : $Name \xrightarrow{m} (Pattern \times Expr)$
   $curfn$     : $(Name \times VAL)^*$
   $fnparms$ : $UniqueId$-set
end
```

In order to show how the *fnparms* are taken into account we will show the definition of the *LooseLookUp* operation which looks up a value in the environment.

```
$LooseLookUp : Name \xrightarrow{o} LVAL$

$LooseLookUp\,(nm) \triangleq$
   (let $topenv = TopEnvL\,()$ in
    for $env$
    in $topenv$
    do for mk- $(id, val)$
       in $env$
       do if $SelName\,(id) = nm$
          then return $\{$mk-$(val,$ if $id \in fnparms$
                                 then $\{\mapsto\}$
                                 else $\{id \mapsto val\})\}$
          else skip;
   $LookUpValueDefs(nm)$ )
```

[10] Note that we could also be evaluating an expression from a value definition.

Note that only the current function application environment, *TopEnvL*, is used for look-up. For each block environment the unique identifier and its associated value is investigated. When the name from the unique identifier is identical to the name which is being looked up we return a (singleton) loose value. Notice how the returned model is affected by whether or not the unique identifier is one of the function parameters.

6 Loose Evaluation Expression Operations

The loose expression evaluation operations which are presented in this section take a syntactic expression and yield a "loose value". This loose value is a set of pairs with a return value and its corresponding model as explained in Section 5.2.

The general strategy behind the loose evaluation of expressions is that whenever a syntactic expression contains more than one subexpression the loose values from these subexpressions are combined such that only "consistent" models are considered. In addition it should be noticed that the *PatternMatch* operation which performs matching of a syntactic pattern to a value yields the set of all possible bindings of the pattern identifiers in case of looseness.

To illustrate this principle we will show the definition of let expressions and function application. The other definitions follow the same underlying idea:

```
$LooseEvalLetExpr : LetExpr \overset{o}{\rightarrow} LVAL$

$LooseEvalLetExpr$ (mk-$LetExpr$ $(pat, expr, in\text{-}e)$) $\triangleq$
  (dcl $lval : LVAL := \{\}$;
   let $val\text{-}lv = LooseEvalExpr\,(expr)$ in
   for all mk-$(val\text{-}v, m) \in val\text{-}lv$
   do let $env\text{-}s = PatternMatch\,(pat, val\text{-}v)$ in
      if $env\text{-}s \neq \{\}$
      then for all $env \in env\text{-}s$
           do ($PushBlkEnv(env)$ ;
              let $in\text{-}lv = LooseEvalExpr\,(in\text{-}e)$ in
              ($PopBlkEnv()$ ;
               $lval := lval \cup Consistent\,(in\text{-}lv, m)$))
      else error;
   return $lval$ )
```

Here in *LooseEvalLetExpr* it can be seen how the right-hand-side expression, *expr*, is evaluated first. For each of the loose value pairs (value and corresponding model) the value is matched against the pattern and given that this succeeds ($env\text{-}s \neq \{\}$) the in-expression is evaluated in a context where each of the bindings from the pattern matching is visible (by *PushBlkEnv*). The loose value pairs which are consistent with the model for the loose value pair of the first expression are added to the resulting loose value for the entire let-expression.

In order to show the close relationship with the operational dynamic semantics presented in [4] we present the corresponding evaluation of a let expression from that definition:

```
$EvalLetExpr : LetExpr \overset{o}{\rightarrow} VAL$

$EvalLetExpr$ (mk-$LetExpr$ ($pat$, $expr$, $in\text{-}e$)) $\triangleq$
  (let $val = EvalExpr\ (expr)$ in
   let $env\text{-}s = PatternMatch\ (pat, val)$ in
   if $env\text{-}s \neq \{\}$
   then let $env \in env\text{-}s$ in
          ($PushBlkEnv(env)$ ;
           let $in\text{-}val = EvalExpr\ (in\text{-}e)$ in
           ($PopBlkEnv()$ ;
            return $in\text{-}val$ ))
   else error)
```

This definition is almost the same as the one used for *LooseEvalLetExpr*. Naturally we did not have a loose value *lval* to collect all the possible values (and corresponding models) because this definition is only interested in an arbitrary model. This means that the first loop in *LooseEvalLetExpr* is not applicable here and the second loop is replaced by a let-be-such-that expression which selects an arbitrary model. In *LooseEvalLetExpr* it is therefore necessary to add the consistent new values in each execution of the loop whereas *EvalLetExpr* can simply return the value right away.

```
$LooseEvalApplyExpr : ApplyExpr \overset{o}{\rightarrow} LVAL$

$LooseEvalApplyExpr$ (mk-$ApplyExpr$ ($fct\text{-}e$, $arg\text{-}e$)) $\triangleq$
  (dcl $lval : LVAL := \{\}$;
   let $arg\text{-}lv = LooseEvalExpr\ (arg\text{-}e)$,
       mk-$(pat, body) = LookUpFn\ (fct\text{-}e)$ in
   ($PushEmptyEnv()$ ;
    for all mk-$(arg\text{-}v, m) \in arg\text{-}lv$
    do let $env\text{-}s = PatternMatch\ (pat, arg\text{-}v)$ in
         ($InstallCurFn(fct\text{-}e, arg\text{-}v, PatternIds\ (pat))$ ;
          for all $env \in env\text{-}s$
          do ($PushBlkEnv(env)$ ;
               let $ap\text{-}lv = LooseEvalExpr\ (body)$ in
               ($PopBlkEnv()$ ;
                $lval := lval \cup Consistent\ (ap\text{-}lv, m)$)));
    $LeaveCurFn()$ );
   $PopEnvL()$ ;
   return $lval$ )
```

In *LooseEvalApplyExpr* the argument expression is first evaluated and the function definition is found from the context. For all possible matches of the argument value against the formal parameter pattern, the global state is updated (by *InstallCurFn*) with information about the current function application (to be used to create unique identifiers) and the pattern identifiers from the formal parameter are installed as well, such that bindings to these are not made a part of the resulting model. The body is then evaluated in a context where these bindings are visible and the consistent models are added to the resulting loose evaluator. Here it can also be seen how the stack of environments are used by pushing and popping environments at the outermost levels (using *PushEmptyEnv* and *PopEnvL*). This is then taken into account by the corresponding look-up operation, which will only search for a binding in the top-most function application environment such that static scoping is used.

Since the subexpressions are evaluated separately, it is necessary to combine the looseness from them. However, this must naturally be restricted such that values resulting from the same model (or a consistent model) are combined. This is dealt with by *Consistent*:

$$Consistent : LVAL \times Model \to LVAL$$

$$\begin{aligned}&Consistent\,(lval, bind) \triangleq\\ &\quad \{\mathsf{mk\text{-}}(val, b \overset{m}{\uplus} bind) \mid \mathsf{mk\text{-}}(val, b) \in lval \cdot\\ &\qquad\qquad \forall\, id \in (\mathsf{dom}\ b \cap \mathsf{dom}\ bind) \cdot b\,(id) = bind\,(id)\}\end{aligned}$$

The *Consistent* function is a very important auxiliary function because it precisely specifies which parts of a loose value are "consistent" with a given model. The point is that when one is not taking the binding of the formal parameters of functions into the model then the consistency check is simply that those unique identifiers which are present in a loose value pair and also in the given model must match the same chosen value (i.e. the choices may not be inconsistent). This is a sufficient condition because the identifiers uniquely identify in which context the choice is made. If the binding of formal parameters was incorporated in the models, the *Consistent* function here would never be able to cope with recursive functions, because "incompatible" bindings of the formal parameters would be present in the same model.

7 Loose Pattern Matching

The operation *PatternMatch* takes a syntactic pattern and a semantic value as input, and returns the set of possible block environments. In each block environment the identifiers defined in the input pattern are bound to the corresponding value from the input semantic value. An empty return set indicates that no matching can be performed.

$$PatternMatch : Pattern \times VAL \xrightarrow{o} BlkEnv\text{-}\mathsf{set}$$

```
$PatternMatch\,(pat\text{-}p, val\text{-}v) \triangleq$
  cases true:
    $(\mathsf{is\text{-}}PatternName\,(pat\text{-}p))$      $\to$ let $\mathsf{mk\text{-}}PatternName\,(id) = pat\text{-}p$ in
                                        return $\{MkBlkEnv\,(id, val\text{-}v)\}$,
    $(\mathsf{is\text{-}}SetEnumPattern\,(pat\text{-}p))$  $\to MatchSetEnumPattern(pat\text{-}p, val\text{-}v)$,
    $(\mathsf{is\text{-}}SetUnionPattern\,(pat\text{-}p))$ $\to MatchSetUnionPattern(pat\text{-}p, val\text{-}v)$
  end
```

Note that the *MkBlkEnv* operation takes into account whether we are currently matching inside a function body, and if so adds the relevant information to the unique identifier which is used in the block environment. This can be seen by:

```
$MkBlkEnv : (Name \times Position) \times VAL \overset{o}{\rightarrow} BlkEnv$

$MkBlkEnv$ (mk- $(nm, pos)$, $val\text{-}v$) $\triangleq$
  let $fninfo = FnInfo\ ()$ in
  return [mk- (mk- $(nm, pos, fninfo)$, $val\text{-}v$)]
```

where *FnInfo* extracts the current function name and current argument value (if any).

To illustrate how more complicated patterns can be dealt with we show the definition of matching with set enumeration patterns and set comprehension patterns[11].

```
$MatchSetEnumPattern : SetEnumPattern \times VAL \overset{o}{\rightarrow} BlkEnv$-set

$MatchSetEnumPattern$ (mk-$SetEnumPattern$ $(elems\text{-}lp)$, $val\text{-}v$) $\triangleq$
  if is-$SET$ $(val\text{-}v)$
  then let mk-$SET$ $(val\text{-}sv) = val\text{-}v$ in
       if card $val\text{-}sv$ = card elems $elems\text{-}lp$
       then let $perm\text{-}slv = Permute\ (SetToSeq\ (val\text{-}sv))$ in
            return $\bigcup \{MatchLists\ (elems\text{-}lp, tmp\text{-}lv) \mid tmp\text{-}lv \in perm\text{-}slv\}$
       else return $\{\}$
  else return $\{\}$
```

This operation returns the set of all possible binding environments. The main condition for matching a set enumeration pattern is that the value is a set value and then that the cardinality of the value corresponds to the number of elements in the set enumeration pattern. We first create the set of all permutations of the input semantic value. The distributed union of all matches of the input pattern list with the elements from the permutation is returned when a successful match is found.

In the case of a set union pattern, we first create all pairs of set values, for which the union is equal to the original input set value, but are still disjoint. For each pair, we create two sets of binding environments. These are combined, and inserted into the resulting set of binding environments after duplicate entries are removed[12]. *UnionMatch* also ensures that pattern identifiers which occur mul-

[11] We have omitted the definition of some of the auxiliary operations (e.g., *Permute*, and *SetToSeq*) which are used here, but the intended meaning should be clear from the names of the operations.

[12] Here it is worth noting that the disjointness criterion which has been used for both set enumeration patterns and for set union patterns no longer conforms to the semantics given in the standard. However, we have decided to retain this here because we have not yet been able to find any examples where one would like such patterns to be non-disjoint.

tiple times are consistently bound to the same value everywhere. The operation is defined as:

```
$MatchSetUnionPattern : SetUnionPattern \times VAL \overset{o}{\rightarrow} BlkEnv$-set
$MatchSetUnionPattern\ (\text{mk-}SetUnionPattern\ (lp\text{-}p, rp\text{-}p), val\text{-}v) \triangleq$
  (dcl $envres\text{-}sl : BlkEnv$-set $:= \{\}$;
   if is-$SET\ (val\text{-}v)$
   then let mk-$SET\ (val\text{-}sv) = val\text{-}v$ in
        (for all mk-$(setl\text{-}sv, setr\text{-}sv) \in$
              $\{\text{mk-}(setl\text{-}sv, setr\text{-}sv) \mid setl\text{-}sv, setr\text{-}sv \in \mathcal{F}\ val\text{-}sv \cdot$
                                        $(setl\text{-}sv \cup setr\text{-}sv = val\text{-}sv) \wedge$
                                        $(setl\text{-}sv \cap setr\text{-}sv = \{\})\}$
         do let $envl\text{-}s = PatternMatch\ (lp\text{-}p, \text{mk-}SET\ (setl\text{-}sv))$,
                $envr\text{-}s = PatternMatch\ (rp\text{-}p, \text{mk-}SET\ (setr\text{-}sv))$ in
            if $envl\text{-}s \neq \{\} \wedge envr\text{-}s \neq \{\}$
            then let $tmpenv = \{CombineBlkEnv\ (e_1, e_2)$
                                 $\mid e_1 \in envl\text{-}s, e_2 \in envr\text{-}s\}$ in
                 $envres\text{-}sl := envres\text{-}sl \cup UnionMatch\ (tmpenv)$
            else skip;
         return $envres\text{-}sl$ )
   else return $\{\}$ )
```

Finally it is worth noting that except for the treatment of position information for pattern identifiers this definition is identical to the one from [4].

8 Examples

Let us first return to the examples presented in Section 3. *fac′* looks like:

```
$fac' : \mathbb{N} \rightarrow \mathbb{N}$
$fac'\ (n) \triangleq$
  if $n = 0$
  then let $x \in \{1, 2\}$ in $x$
  else $n \times fac'\ (n - 1)$
```

For *fac′* with two expression evaluators, calculating $fac'(4)$ yields 24 and 48 with two different models corresponding to the normal factorial of 4 and two times this just as expected. The loose value, obtained from the evaluation of *LooseEvalExpr* applied with the abstract syntax representation of $fac'(4)$ looks like:

$$\{mk\text{-}(24, \{x_{fac'(0)} \mapsto 1\}), mk\text{-}(48, \{x_{fac'(0)} \mapsto 2\})\}$$

The function *fac″* looks like:

```
$fac'' : \mathbb{N} \to \mathbb{N}$
$fac''\,(n) \triangleq$
  let $x \in \{1, 2\}$ in
  if $n = 0$
  then $x$
  else $x \times n \times fac''\,(n - 1)$
```

For fac'' calculating $fac''(2)$ yields 8 (2^3) elements in the loose value:

$$\begin{array}{l}
\{mk\text{-}(2, \{x_{fac''(0)} \mapsto 1, x_{fac''(1)} \mapsto 1, x_{fac''(2)} \mapsto 1\}),\\
\;\; mk\text{-}(4, \{x_{fac''(0)} \mapsto 1, x_{fac''(1)} \mapsto 1, x_{fac''(2)} \mapsto 2\}),\\
\;\; mk\text{-}(4, \{x_{fac''(0)} \mapsto 1, x_{fac''(1)} \mapsto 2, x_{fac''(2)} \mapsto 1\}),\\
\;\; mk\text{-}(4, \{x_{fac''(0)} \mapsto 2, x_{fac''(1)} \mapsto 1, x_{fac''(2)} \mapsto 1\}),\\
\;\; mk\text{-}(8, \{x_{fac''(0)} \mapsto 1, x_{fac''(1)} \mapsto 2, x_{fac''(2)} \mapsto 2\}),\\
\;\; mk\text{-}(8, \{x_{fac''(0)} \mapsto 2, x_{fac''(1)} \mapsto 1, x_{fac''(2)} \mapsto 2\}),\\
\;\; mk\text{-}(8, \{x_{fac''(0)} \mapsto 2, x_{fac''(1)} \mapsto 2, x_{fac''(2)} \mapsto 1\}),\\
\;\; mk\text{-}(16, \{x_{fac''(0)} \mapsto 2, x_{fac''(1)} \mapsto 2, x_{fac''(2)} \mapsto 2\}\}
\end{array}$$

Here it can be seen that we get one with 2 as result, three with 4 as result, three with 8 as result, and one with 16 as result. This corresponds exactly to the expression evaluators from Section 3 as expected.

In addition to these, 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:

```
$Add : \mathbb{N}\text{-set} \to \mathbb{N}$
$Add\,(s) \triangleq$
  if $s = \{\}$
  then $0$
  else let $e \in s$ in
       $e + Add\,(s \setminus \{e\})$
```

For example using this definition as context it is possible to calculate by the looseness analysis definition described in this paper that $Add(\{3, 4, 5\})$ yields 12 in all 6 (3!) possible models (corresponding to the different orders in which the elements can be chosen). This indicates that the looseness which is present is internal looseness. This is the form of looseness which is used in most "real" specifications, but it is still important to be able to convince oneself that the looseness involved simply is internal.

Here the loose value looks like:

$$\begin{array}{l}
\{mk\text{-}(12, \{e_{Add(\{3\})} \mapsto 3, e_{Add(\{3,4\})} \mapsto 4, e_{Add(\{3,4,5\})} \mapsto 5\}),\\
\;\; mk\text{-}(12, \{e_{Add(\{3\})} \mapsto 3, e_{Add(\{3,5\})} \mapsto 5, e_{Add(\{3,4,5\})} \mapsto 4\}),\\
\;\; mk\text{-}(12, \{e_{Add(\{4\})} \mapsto 4, e_{Add(\{3,4\})} \mapsto 3, e_{Add(\{3,4,5\})} \mapsto 5\}),\\
\;\; mk\text{-}(12, \{e_{Add(\{4\})} \mapsto 4, e_{Add(\{4,5\})} \mapsto 5, e_{Add(\{3,4,5\})} \mapsto 3\}),\\
\;\; mk\text{-}(12, \{e_{Add(\{5\})} \mapsto 5, e_{Add(\{3,5\})} \mapsto 3, e_{Add(\{3,4,5\})} \mapsto 4\}),\\
\;\; mk\text{-}(12, \{e_{Add(\{5\})} \mapsto 5, e_{Add(\{4,5\})} \mapsto 4, e_{Add(\{3,4,5\})} \mapsto 3\})\}
\end{array}$$

An equivalent definition of this function also has 6 models which all yield 12 with the same arguments.

$$
\begin{array}{l}
Add2 : \mathbb{N}\textsf{-set} \to \mathbb{N} \\
Add2\,(s) \ \triangleq \\
\quad \textsf{cases}\ s : \\
\qquad \{\} \qquad\quad \to 0, \\
\qquad \{e\} \qquad \to e, \\
\qquad \{e\} \cup s' \to e + Add2\,(s') \\
\quad \textsf{end}
\end{array}
$$

Here the function is just using the set enumeration and set union patterns, for which the definition was presented in the previous section.

The examples presented in this section (and all the complicated examples from [5]) have actually been tested by the interpreter from the IFAD VDM-SL Toolbox[13]. This gives us a reasonable level of confidence in the correctness of the definition.

9 Concluding Remarks

In this paper we have shown how underdetermined explicit definitions can be evaluated and how information about all possible results (and the corresponding models) for an executable subset of VDM-SL can be obtained. We have shown how complex this is, in particular when one combines the underdeterminedness with recursion. However, we have also shown that only minor adjustments needs to be made for the evaluation part of the definition from [4] to specify such a looseness analysis tool.

In addition, this paper shows that it is possible to use the definition here as a kind of decision procedure which can reduce the simple (executable) expressions to corresponding "loose values" when one wishes to prove properties about specifications which contain underdetermined explicit definitions. We believe that this is a significant improvement if one wishes to carry out such proofs. Even in cases where the definition shown here yields unexpected results it would be valuable. In this way it could be shown that some property the specifier would like to prove does not hold, which he otherwise would need to spend time to discover.

An investigation of the combination of recursion and underdeterminedness like the one presented here has not been done before. It is also worth noting that the main motivation for writing this specification was that in the process of developing proof rules for this combination an error was discovered in the semantics given in the standard (e.g. also in [3]). Thus, in earlier versions of the standard this combination was not treated in a consistent way.

The approach taken here has a few limitations. First of all, it only works if all models do not yield any undefined results, which is to be expected. In

[13] The test suite which has been used ensures that all constructs in the entire specification have been covered.

addition, the definition presented here cannot deal with constructs which are not executable. All executable constructs except lambda expressions can be dealt with by this approach. As was seen with *fac''* in Section 3, one can get infinite sets which cannot be handled in VDM-SL. Thus, one would need to handle lambda expressions with a closure model, which means that the looseness inside the lambda expression will not be visible until the function is applied to some argument. The only problem about this is that in all other cases the cardinality of the loose value indicates the number of combinations of choices which can be taken, but this will not be the case if this definition was extended with a lambda expression.

Finally, in order for this to be useful for "real" applications it would be necessary to expand this definition to cover as much as possible from standard VDM-SL and to closely integrate it with a proof tool for VDM-SL. We believe that this is possible, but that will be a part of future research where a proof tool supporting full VDM-SL is the final goal.

Acknowledgements

Tha author would like to thank Hanne Christensen, René Elmstrøm, Bo Stig Hansen, Mike Hinchey, Poul Bøgh Lassen, Paul Mukherjee, Nico Plat, Marcel Verhoef and Henrik Voss for valuable comments on earlier versions of this paper. In addition the author would like to thank his employer IFAD and the Danish Academy of Technical Sciences for financial support to carry out his PhD studies.

References

1. Information Technology Programming Languages – VDM-SL. First Committee Draft Standard: CD 13817-1, November 1993. ISO/IEC JTC1/SC22/WG19 N-20.
2. René Elmstrøm, Peter Gorm Larsen and Poul Bøgh Lassen. The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. *ACM Sigplan Notices*, Summer 1994.
3. 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.
4. Peter Gorm Larsen and Poul Bøgh Lassen. An Executable Subset of Meta-IV with Loose Specification. In S. Prehn and W.J. Toetenel, editors, *VDM '91: Formal Software Development Methods*. Springer-Verlag, March 1991.
5. Peter Gorm Larsen. Towards Proof Rules for Looseness in Explicit Definitions from VDM-SL. In *Proceedings of the "International Workshop on Semantics of Specification Languages (SoSL)"*, 25–27 October 1993, Utrecht, Springer-Verlag 1994.
6. Poul Bøgh Lassen. IFAD VDM-SL Toolbox. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*. Springer-Verlag, 1993.
7. David A. Schmidt. *Denotational Semantics – A Methodology for Language Development*. Allyn and Bacon, Inc. 1986.

8. Natarajan Shankar, Sam Owre and John Rushby. A Tutorial on Specification and Verification Using PVS. In Peter Gorm Larsen, editor, *Tutorial Material – Formal Methods Europe '93*, April 1993.
9. Harald Søndergaard and Peter Sestoft. Referential Transparency, Definiteness and Unfoldability. *Acta Informatica*, 27:505–517, 1990.
10. Harald Søndergaard and Peter Sestoft. Non-determinism in Functional Languages. *The Computer Journal*, 35(5):514–523, October 1992.