Towards Proof Rules for Looseness in Explicit Definitions from VDM-SL

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

Abstract

The model-oriented formal method called VDM contains a specification language called VDM-SL. This language existed in a number of different dialects, but now a standard for the language has been prepared, including a dynamic semantics defined from a model-theoretic point of view. Thus, it is not at all clear that the defined semantics is appropriate for deriving proof rules which reflect the semantics. This paper focus on the possible ways of defining proof rules which reflect the semantics of explicit definitions which contain looseness.

The model-theoretic view which has been chosen for the definition of the semantics for VDM-SL, incorporates looseness by denoting the semantics of a (loose) specification as a set of models. The proof system should be designed such that properties which can be proved about a given specification should hold for all its models. This paper shows why it is an interesting challenge to develop a proof system which are able to do this.

1 Introduction

Formal specification languages have properties not usually found in programming languages. One of these is the possibility of specifying constructs that denote a choice. Such 'loose specification' arises because a specification generally needs to be stated at a higher level of abstraction than that of the final implementation. When loose specification is used, the question of how to interpret this looseness is often ignored. However, this interpretation is important, especially if a specification is to be proven to implement another specification. The implementation relation relies on the interpretation of looseness. Thus, this interpretation is especially interesting in connection with the proof rules for the specification language. However, the implementation relation and the proof rules have not yet been formulated for the entire standard VDM-SL language.

Work on the provision of a basic proof theory for reasoning about specifications and developments in VDM has been done by researchers in the UK and Australia (to appear in [Bicarregui&93] and in [Fitzgerald&93]). The extension of this to give a proof theory for the full standard language VDM-SL is an area of the author's research. This paper will focus on one aspect: proof rules for explicit definitions.

It is natural that implicit function and operation definitions can contain looseness. However, in VDM-SL it is also possible to let explicit function and operation definitions be loosely specified (see e.g. [Larsen&89], and [Larsen92]). In some other specification languages, e.g. VVSL [Middelburg90], this is disallowed. It is a matter of taste whether one wishes to include such a style of specification in a language. However, we think that it is important to investigate the consequences of such a decision in the proof rules. In RSL [RAISE94], looseness in explicit definitions is interpreted as nondeterminism and not as underdetermined which is done inside functions in VDM-SL[1]. Underdeterminedness can be obtained by means of axioms in RSL, but these simply correspond to a more general form of implicit definitions than the pre-post style used in VDM-SL. Thus, the problems dealt with in this paper does not appear in RSL. This kind of looseness is also avoided in Z [Hayes93].

To sum up, we can say that, since explicit expressions in VDM-SL can be loose, we need to investigate how this can be reflected in the proof rules for VDM-SL. In this paper we will present different ways of treating such looseness and indicate their limitations. Prior to presenting the different possible approaches we explain the semantics which have been chosen for looseness in VDM-SL.

2 The Semantics of Looseness in VDM-SL

There are (at least) two different ways of interpreting loose specification. In VDM-SL we have termed these: 'underdeterminedness'[2] (allowing several different deterministic implementations) and 'nondeterminism' (allowing non-deterministic implementations). The difference between the two lies in the time at which the looseness is resolved. If a loosely specified construct is interpreted as underdetermined, it is decided by the implementer at implementation time which functionality to use. Since this choice is static the final implementation becomes deterministic. However, if a loosely specified construct is interpreted as nondeterministic, it is possible to delay this choice until execution time. In this way the actual functionality is chosen dynamically at run-time. This means that the final implementation may be nondeterministic. Ideally, a specifier would like to be able to choose freely between these different interpretations for all functions and operations.

In VDM-SL we have chosen to interpret function definitions as underdetermined, while operation definitions are interpreted as nondeterministic[3]. The difference between functions and operations is that operations may operate on a (specified) state space, while functions may not refer to the state. Thus, functions are applicative while operations may be imperative. Both expressions and pattern matchings can be loosely specified in VDM-SL. Thus, when there is looseness in an expression or a pattern, its interpretation depends upon whether it is used inside a function or an operation.

Concerning the proof rules, clearly the underdetermined interpretation gives rise to most problems, because two loose expression can be proved to be equal

[1] The distinction between these is further described in Section 2.

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

[3] The complexity of the semantics with an arbitrary combination of loose specification is given in [Wieth89].

under certain conditions. Two loose expressions which are interpreted in a nondeterministic way can never be proved to be equal. Thus, the proof rules we will derive in the remaining part of this paper will reflect the underdetermined semantics of looseness to illustrate how the hardest problem can be dealt with.

3 The naïve approach

In [Jones&91],p.267 a naïve approach for treating proof rules for looseness in expressions is presented. A general let-be-such-that expression like:

$$\mathsf{let}\ x : A\ \mathsf{be\ st}\ P(x)\ \mathsf{in}\ E(x)$$

can contain looseness, because $P(x)$ does not necessarily restrict x to be a unique element from A. When such an expression is used in a specification it can be translated into:

$$\boxed{\mathsf{letbe\text{-}def}_1}\ \frac{\exists\, x : A \cdot P(x)}{(\mathsf{let}\ x : A\ \mathsf{be\ st}\ P(x)\ \mathsf{in}\ E(x)) = E(\varepsilon x : A \cdot P(x))}$$

Here the usual choice operator from first order logic is used. In [Jones&91] two extra axioms are defined for the choice operator:

$$\boxed{\varepsilon\text{-def}}\ \frac{\exists\, x : A \cdot P(x)}{P(\varepsilon x : A \cdot P(x))}$$

$$\boxed{\varepsilon\text{-form}}\ \frac{\exists\, x : A \cdot P(x)}{\varepsilon x : A \cdot P(x) : A}$$

As a consequence of the "ε-form" and the "=-self-I" rule[4] the following can be proved:.

$$\boxed{\varepsilon\text{-deterministic}}\ \frac{\exists\, x : A \cdot P(x)}{\varepsilon x : A \cdot P(x) = \varepsilon x : A \cdot P(x)}$$

However, if for example two functions f and g are defined with the same loose expression, models will exist where they yield different results. Let us consider a simple example which can be used to illustrate this[5]:

$$f : () \to \mathbb{N}$$
$$f\ () \triangleq$$
$$\quad \mathsf{let}\ x : \mathbb{N}\ \mathsf{be\ st}\ x \in \{1, 2\}\ \mathsf{in}\ x$$

[4] $\boxed{=\text{-self-I}}\ \frac{a : A}{a = a}$

[5] These two functions can be slightly more elegantly described in VDM-SL by using the "be-such-that" expression as a set binding (and then not having a type binding at all). However, at this point we abstract away from this because the let-be-such-that expressions used in [Jones&91] always are typed in this way.

$g : () \to \mathbb{N}$

$g\,() \triangleq$
  $\mathsf{let}\ x : \mathbb{N}\ \mathsf{be\ st}\ x \in \{1,2\}\ \mathsf{in}\ x$

With the three proof rules defined above and a rule for function unfolding it is possible to derive for instance that $f() = g()$ which do not hold in all models. Thus, this model can describe looseness appropriately for one function alone, but cannot deal with combinations of loose functions.

4 Tagging with static context information

To repair an example like the one mentioned in the previous section, one can consider tagging the choice expression with the origin of the expression which contains the looseness. In this way it cannot be derived that $f() = g()$. However, it is not sufficient simply to tag an expression with the name of the function in which it is used, because the same syntactic expression can be used several times and it may not denote the same value in all models[6]. Thus, for each expression in a VDM-SL specification a unique identification must be present and used for tagging the corresponding choice expression.

The "letbe-def$_1$" rule above would now look something like:

$$\boxed{\mathsf{letbe\text{-}def_2}}\ \frac{\exists\, x : A \cdot P(x)}{(\mathsf{let}\ x : A\ \mathsf{be\ st}\ P(x)\ \mathsf{in}\ E(e))_{\mathcal{POS}} = E(\varepsilon x : A \cdot P(x)_{\mathcal{POS}})}$$

where $\mathcal{POS}$ indicates the position of the let-be-such-that expression in the concrete specification. Since the position of the body of the example functions f and g are different, it will neither be possible to prove that $f() = g()$ or $f() \neq g()$. This corresponds to the standard semantics because neither of these propositions holds in all models. However, if we modify f slightly we get another problem:

$f' : \mathbb{B} \to \mathbb{N}$

$f'\,(b) \triangleq$
  $\mathsf{let}\ x : \mathbb{N}\ \mathsf{be\ st}\ x \in \{1,2\}\ \mathsf{in}\ x$

Even though that the argument b is not used in the body of the function it will not hold in all models that $f'(\mathsf{true}) = f'(\mathsf{false})$[7], but this can be proved with the "letbe-def$_2$" rule defined above. Thus, this approach can deal with combinations of parameterless loose functions, but it cannot correctly cope with loose functions which have parameters.

[6] An example of this could be an expression like: $(\mathsf{let}\ x : \mathbb{N}\ \mathsf{be\ st}\ x \in \{1,2\}\ \mathsf{in}\ x) + (\mathsf{let}\ x : \mathbb{N}\ \mathsf{be\ st}\ x \in \{1,2\}\ \mathsf{in}\ x)$ which also has a model where it denotes 3.

[7] Two of the models will map true and false to the same value (either 1 or 2) while the two other models will map true and false to different values.

5 Tagging with dynamic context information

To repair the problems with simply using static context information, one can add the dynamic dependencies, i.e. the parameters of the function in which the expression occur.

The "letbe-def$_2$" rule above would now look something like:

$$\boxed{\text{letbe-def}_3}\ \frac{\exists\, x : A \cdot P(x)}{(\mathsf{let}\ x : A\ \mathsf{be\ st}\ P(x)\ \mathsf{in}\ E(e))_{(\mathcal{POS},[args])} = E(\varepsilon x : A \cdot P(x)_{(\mathcal{POS},[args])})}$$

With this rule one will neither be able to prove $f'(\mathsf{true}) = f'(\mathsf{false})$ nor $f'(\mathsf{true}) \neq f'(\mathsf{false})$ which is correct because they do not hold in all models.

Thus, this approach can deal with combinations of arbitrary loose functions which have parameters. Therefore, one can now try to investigate how proof rules can be derived for the most general expressions with arbitrary patterns. This will be presented in the form of skeletons which can be used as a basis for an automatic translation for each component of a given specification to its corresponding proof rules.

5.1 General rules for loose values

However, before we look at this, we first present three other rules for ε values which are more directly applicable than the "ε-def" and "ε-form" rules above[8]:

$$\boxed{\varepsilon\text{-I}}\ \frac{\forall\, e : A \cdot (P(e) \ \Rightarrow\ Q(e)), \exists\, e : A \cdot P(e)}{Q(\varepsilon x : A \cdot P(x)_{\text{any tagging}})}$$

$$\boxed{\varepsilon\text{-E}}\ \frac{a : A, P(a), \exists!\, x : A \cdot P(x)}{(\varepsilon x : A \cdot P(x)_{\text{any tagging}}) = a}$$

$$\boxed{\varepsilon\text{-difference}}\ \frac{\exists\, x : A \cdot P(x), \exists\, x : A \cdot Q(x), \neg\, \exists\, x : A \cdot P(x) \wedge Q(x)}{\varepsilon x : A \cdot P(x)_{\,Tag_1} \neq \varepsilon x : A \cdot Q(x)_{\,Tag_2}}$$

The first of these rules simply says that if some property, Q, holds for all elements in the loose value, the property will also hold for the loose value[9]. The second rule says that if the loose value has only one possible value, then that is going to be the value of the choice expression. Finally the third rule says that if two choice values have disjoint predicates these choice values will be different in all models.

[8] In [Fitzgerald93] it is shown that these rules follow from a version of the "ε-def" and the ε-form" which are extended with tags

[9] The existential quantification assumption is needed because a loose value must at least have one possibility to choose from.

These rules can be formulated slightly more elegantly by using a set notation (allowing infinite sets[10]) instead:

$$\boxed{\varepsilon\text{-I}}\ \frac{\forall\, e \in s \cdot (P(e) \ \Rightarrow\ Q(e)), \exists\, e : A \cdot P(e)}{Q(\varepsilon x \in \{e \mid e \in s \cdot P(x)\}_{\text{any tagging}})}$$

$$\boxed{\varepsilon\text{-E}}\ \frac{a : A}{(\varepsilon x \in \{a\}_{\text{any tagging}}) = a}$$

$$\boxed{\varepsilon\text{-difference}}\ \frac{s_1 \cap s_2 = \{\,\}}{(\varepsilon id_1 \in s_1)_{Tag_1} \neq (\varepsilon id_2 \in s_2)_{Tag_2}}$$

Instead of a type and a predicate we have here used a set notation because the predicate simply is used to create a subtype of the given type. The main advantage is that the notation from set theory simplify the appearance[11] of the assumptions in both the "ε-E" rule and the "ε-difference" rule.

Below we will use the set notation consequently[12].

5.2 Let expressions

Let us now consider a specification which contains an ordinary let expression with a general pattern (which can be loose):

$$\mathsf{let}\ pat = defexpr\ \mathsf{in}\ inexpr$$

then one can produce a proof rule following a skeleton like:

$$\boxed{\mathsf{let}\text{-def}}\ \frac{\exists\, \text{patids} : \text{types inferred} \cdot pat' = defexpr}{\begin{array}{l}(\mathsf{let}\ pat = defexpr\ \mathsf{in}\ inexpr)_{(\mathcal{POS},[args])} = \\ \varepsilon id \in \{inexpr \mid \text{patids} : \text{types inferred} \cdot pat' = defexpr\}_{(\mathcal{POS},[args])}\end{array}}$$

where "patids" will be the pattern identifiers which can be extracted from the pattern, *pat*, and "types inferred" will be the corresponding types which can be inferred from a static semantics analysis[13]. *pat'* is an expression which in the concrete syntax is identical to the pattern *pat* except that possible "don't care" patterns have been replaced with new and unused identifiers (which then

[10] In model-theoretic terms this simply correspond to the carrier sets of domains from the domain universe of VDM-SL where an ordering also is present.

[11] The existence of a single value satisfying a given predicate can be expressed as a singleton set.

[12] Alternatively we could have used the subtype notation from [Jones&91] (which we have used so far) or the domain notation from the domain universe.

[13] Here the "possibly well-formedness" (see [Bruun&91]) check will be performed on the *defexpr* to derive its possible type. This derived type is then used to derive the type of the individual pattern identifiers which is inferred to be used here. Here it is important to make sure that the inferred types are sufficiently fine-grained, such that *defexpr* is defined for all possible values from the inferred types.

also are used in the list of pattern identifiers in the existential quantification). Notice that under "normal" circumstances, i.e. where *pat* is not a loose pattern (contains neither a set union pattern nor a sequence concatenation pattern) this choice value can be reduced by the ε-E rule right away.

A simple example

Let us take a simple example which can be used to illustrate the let-def skeleton:

$h : (\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$

$h\,(pair) \triangleq$
  let $mk\text{-}(n, m) = pair$ in $n + m$

would give rise to a proof rule like:

$$\boxed{\text{let-def(h)}_1}\ \frac{\exists\, n : \mathbb{N}, m : \mathbb{N} \cdot mk\text{-}(n, m) = pair}{\begin{array}{r}(\text{let } mk\text{-}(n, m) = pair \text{ in } n + m)_{(\mathcal{POS},[pair])} = \\ \varepsilon id \in \{n + m \mid n : \mathbb{N}, m : \mathbb{N} \cdot mk\text{-}(n, m) = pair\}_{(\mathcal{POS},[pair])}\end{array}}$$

where the loose value can be reduced because it only contains a set comprehension expression which is simply a singleton set.

Below we present an example of a proof which uses this proof rule:

from	$n' : \mathbb{N}, m' : \mathbb{N}$	
1	$n' + m' : \mathbb{N}$	add-lem(h)
2	$mk\text{-}(n', m') : (\mathbb{N} \times \mathbb{N})$	two-tuple-form(h)
3	$\exists\, n : \mathbb{N}, m : \mathbb{N} \cdot mk\text{-}(n, m) = mk\text{-}(n', m')$	$\exists$-equal-lem(h)
4	$h(mk\text{-}(n', m')) = (\text{let } mk\text{-}(n, m) = mk\text{-}(n', m') \text{ in } n + m)_{(\mathcal{POS},[mk\text{-}(n', m')])}$	h-defn$_1$(2)
5=4	$\varepsilon id \in \{n + m \mid n : \mathbb{N}, m : \mathbb{N} \cdot mk\text{-}(n, m) = mk\text{-}(n', m')\}_{(\mathcal{POS},[mk\text{-}(n', m')])}$	let-def(h)$_1$(3,4)
6=4	$\varepsilon \in \{n' + m'\}$	setcomp-sin-lem(5=4)
infer	$h(mk\text{-}(n', m')) = n' + m'$	ε-E(1,6=4)

Notice that the proof is displayed in the 'natural deduction' style except that a simplification resembling [Dijkstra&90] is used. The simplification we have used is that when a justified expression is an equality and one wish to rewrite the right-hand-side of it this can be done by adding an equal sign and a line number (where the equality come from) and then simply write the new right-hand-side. This simplification is very handy when the *lhs* and/or the *rhs* become large.

In the proof above the following extra rules and lemmas are used:

$$\boxed{\text{add-lem}}\ \frac{a : \mathbb{N}, b : \mathbb{N}}{a + b : \mathbb{N}}$$

This lemma simply says that adding two natural numbers give a new natural number.

$$\boxed{\text{two-tuple-form}}\ \frac{a : A, b : B}{mk\text{-}(a, b) : (A \times B)}$$

This rule is simply a formation rule for two-tuples.

$$\boxed{\exists\text{-equal-lem}}\ \frac{a : A, b : B}{\exists\, mk\text{-}(x, y) : (A \times B) \cdot mk\text{-}(a, b) = mk\text{-}(x, y)}$$

This lemma explains that knowledge about existence of two specific values can be used to introduce an existential quantification with equality.

$$\boxed{\text{h-defn}_1}\ \frac{pair : (\mathbb{N} \times \mathbb{N})}{h(pair) = (\mathsf{let}\ mk\text{-}(n, m) = pair\ \mathsf{in}\ n + m)_{(\mathcal{POS}, [pair])}}$$

This rule corresponds to the translation for the explicit function definition for h.

$$\boxed{\text{setcomp-sin-lem}}\ \frac{a : A, b : B, f(a, b) : C}{\{f(a', b') \mid a' : A, b' : B \cdot a = a' \wedge b = b'\} = \{f(a, b)\}}$$

This lemma simply says that a set comprehension which uniquely describes the member value by equality is a singleton set.

To prove the lemmas we appeal to the basic theories of LPF, type constructors and basic types.

A more complicated example

Let us now take a more complicated example which can be used to illustrate the let-def skeleton:

$$h : \mathbb{N}^+ \to \mathbb{N}^*$$

$$h\,(l) \triangleq \mathsf{let}\ l' \frown [\text{-}] \frown l'' = l\ \mathsf{in}\ l'' \frown l'$$

This function removes an arbitrary element from a finite non-empty sequence of natural numbers and shift the preceding and succeeding sequences around. It would give rise to a proof rule like:

$$\boxed{\text{let-def(h)}_2}\ \frac{\exists\, l' : \mathbb{N}^*, l'' : \mathbb{N}^*, id : \mathbb{N} \cdot l' \frown [id] \frown l'' = l}{\begin{array}{l}(\mathsf{let}\ l' \frown [\text{-}] \frown l'' = l\ \mathsf{in}\ l'' \frown l')_{(\mathcal{POS}, [l])} = \\ \varepsilon id \in \{l'' \frown l' \mid l' : \mathbb{N}^*, l'' : \mathbb{N}^*, id : \mathbb{N} \cdot l' \frown [id] \frown l'' = l\}_{(\mathcal{POS}, [l])}\end{array}}$$

where the loose value cannot directly be reduced to a singleton set because looseness is present.
Below we present an example of a proof which uses this proof rule:

from $list : \mathbb{N}^+$		
1	$\exists\, l' : \mathbb{N}^*, l'' : \mathbb{N}^*, id : \mathbb{N} \cdot l' \frown [id] \frown l'' = list$	seq-conc-lem(h)
2	$h(list) = (\textsf{let}\ l' \frown [\text{-}] \frown l'' = list\ \textsf{in}\ l'' \frown l')_{(\mathcal{POS},[list])}$	h-defn$_2$(h)
3=2	$\varepsilon id \in \{l'' \frown l' \mid l' : \mathbb{N}^*, l'' : \mathbb{N}^*, id : \mathbb{N} \cdot l' \frown [id] \frown l'' = list\}_{(\mathcal{POS},[list])}$	let-def$_2$(h)(1,2)
4	$\forall\, l \in \{l'' \frown l' \mid l' : \mathbb{N}^*, l'' : \mathbb{N}^*, id : \mathbb{N} \cdot l' \frown [id] \frown l'' = list\} \cdot$	
	elems $l \subseteq$ **elems** $list$	elems-conc-lem(h)
infer **elems** $h(list) \subseteq$ **elems** $list$		ε-I(1,4,3=2)

Where the following extra rules and lemmas are used:

$$\boxed{\text{seq-conc-lem}}\ \frac{l : A^+}{\exists\, l_1 : A^*, l_2 : A^*, a : A \cdot l_1 \frown [a] \frown l_2 = l}$$

This lemma simply says that if one has a non-empty sequence there exists an element which can be used to split the sequence into two potentially empty sequences (and this element).

$$\boxed{\text{h-defn}_2}\ \frac{l : \mathbb{N}^+}{h(l) = (\textsf{let}\ l' \frown [\text{-}] \frown l'' = l\ \textsf{in}\ l'' \frown l')_{(\mathcal{POS},[mk\text{-}(n',m')])}}$$

This rule corresponds to the translation for the explicit function definition for h.

$$\boxed{\text{elems-conc-lem}}\ \frac{list : A^+}{\begin{array}{c}\forall\, l \in \{l'' \frown l' \mid l' : \mathbb{N}^*, l'' : \mathbb{N}^*, id : \mathbb{N} \cdot l' \frown [id] \frown l'' = list\} \cdot \\ \textsf{elems}\ l \subseteq \textsf{elems}\ list\end{array}}$$

This lemma relies on the intuitive fact that if an element is removed from a sequence, the elements of the new sequence will be a subset of the elements of the original sequence.

The principle which is used in the skeleton for the let-def rule can be applied for other constructs which introduce looseness explicitly. Below we list the corresponding skeletons for let-be-such-that expressions, cases expressions and quantified expressions, where the most general form for patterns are taken into account in all cases.

5.3 Let-be-such-that expressions

Let us now consider a specification which contains a let-be-such-that expression with a general pattern (which can be loose):

let *pat* : *type* **be st** *predexpr* **in** *inexpr*

then one can produce a proof rule following a skeleton like:

$$\boxed{\textsf{letbe-def}}\ \frac{\exists\,\text{patids}:\text{types inferred}\cdot predexpr}{\begin{array}{c}(\textsf{let}\ pat:type\ \textsf{be st}\ predexpr\ \textsf{in}\ inexpr)_{(\mathcal{POS},[args])} = \\ \varepsilon id \in \{inexpr \mid \text{patids}:\text{types inferred}\cdot predexpr\}_{(\mathcal{POS},[args])}\end{array}}$$

where "patids" will be the pattern identifiers which can be extracted from the pattern, pat, and "types inferred" will be the corresponding types which can be inferred from a static semantics analysis (here it is worth noticing that if no matching is possible an empty set (or type) will be inferred and thus it will be impossible to satisfy the assumption with the existential quantification).

5.4 Cases expressions

Let us now consider a specification which contains a cases expression with general patterns (which can be loose):

$$\begin{array}{l}\textsf{cases}\ match: \\ \quad pat_1 \rightarrow expr_1, \\ \quad pat_2 \rightarrow expr_2, \\ \quad \ldots \quad \rightarrow \ldots, \\ \quad pat_n \rightarrow expr_n \\ \textsf{end}\end{array}$$

Notice that the last pattern, pat_n, can be an **others** clause which corresponds semantically to a "don't care"-pattern[14].

Then one can produce two proof rules by means of the two following skeletons:

$$\boxed{\textsf{cases-def1}}\ \frac{\exists\,\text{patids}_1:\text{types inferred}\cdot pat_1' = match}{\begin{array}{c}(\textsf{cases}\ match: pat_1 \rightarrow expr_1, \ldots, pat_n \rightarrow expr_n\ \textsf{end}\ )_{(\mathcal{POS},[args])} = \\ \varepsilon id \in \{expr_1 \mid \text{patids}_1:\text{types inferred}\cdot pat_1' = match\}_{(\mathcal{POS},[args])}\end{array}}$$

where "patids$_1$" will be the pattern identifiers which can be extracted from the pattern, pat_1, and "types inferred" will be the corresponding types which can be inferred from a static semantics analysis. pat_1' is an expression which in the concrete syntax is identically to the pattern pat_1 except that possible "don't care" patterns have been replaced with new and unused identifiers (which then also are used in the list of pattern identifiers in the existential quantification).

$$\boxed{\textsf{cases-def2}}\ \frac{\neg\,\exists\,\text{patids}_1:\text{types inferred}\cdot pat_1' = match, \text{“}1 \neq n\text{”}}{\begin{array}{c}(\textsf{cases}\ match: pat_1 \rightarrow expr_1, \ldots, pat_n \rightarrow expr_n\ \textsf{end}\ )_{(\mathcal{POS},[args])} = \\ (\textsf{cases}\ match: pat_2 \rightarrow expr_2, \ldots, pat_n \rightarrow expr_n\ \textsf{end}\ )_{(\mathcal{POS},[args])}\end{array}}$$

where "patids$_1$" will be the pattern identifiers which can be extracted from the pattern, pat_1, and "types inferred" will be the corresponding types which can be inferred from a static semantics analysis. pat_1' is an expression which in the

[14] Here it is worth noticing that the semantics of VDM-SL is such that if no **others** clause is present the matching patterns should cover all possible matching values. The cases expression yield bottom for all values which do not match any of the patterns.

concrete syntax is identically to the pattern pat_1 except that possible "don't care" patterns have been replaced with new and unused identifiers (which then also are used in the list of pattern identifiers in the existential quantification). The quoted part with $1 \neq n$ in the hypothesis is simply intended to indicate that there must be more alternatives if this rule is to be applied.

5.5 Quantified expressions

Let us now consider a specification which contains a number of quantified expressions with a general pattern (which can be loose):

quantifier $pat : type \cdot predexpr$

For such a quantified expression one can use a skeleton like:

$$\boxed{\text{quantifier-def}} \quad \frac{\text{quantifier patids} : \text{type inferred} \cdot predexpr : \mathbb{B}}{(\text{quantifier } pat : type \cdot predexpr)_{(\mathcal{POS},[args])} = (\text{quantifier patids} : \text{type inferred} \cdot predexpr)_{(\mathcal{POS},[args])}}$$

where "patids" will be the pattern identifiers which can be extracted from the pattern, *pat*, and "types inferred" will be the corresponding types which can be inferred from a static semantics analysis. The quantifier can either be a universal quantifier ($\forall$), a existential quantifier ($\exists$), or a unique existential quantifier ($\exists!$). Here it is worth noting that looseness in patterns in the model theory (potentially) correspond to several models. At first sight this may not correspond to ones intuition but considering the unique existential quantification it becomes clear that this is the most natural solution (see the example in the section about quantifiers in [Larsen92]).

5.6 Combining looseness

Having defined skeletons for expressions which can introduce explicit looseness it becomes interesting to investigate whether it is possible to define proof rules which enables one to combine expressions which contain looseness. Thereby one can achieve a ε-calculus which can be used to manipulate loose values. However, it turns out that there is a problem with creating tags for the new loose values. Below we illustrate how this approach have problems with such combinations.

Let us consider an example where looseness is combined:

let $a : \mathbb{N}$ be st $a \in \{1, 2\}$ in
let $b : \mathbb{N}$ be st $b \in \{3, a + 2\}$ in $a + b$

it would be desirable to be able to reduce such an expression to a choice value like: $(\varepsilon id \in \{4, 5, 6\})_{(\mathcal{POS},[args])}$ where the looseness has been propagated to the outermost level. However, doing such a propagation requires some way of keeping a record of the choices made underway (e.g. that a may be bound to either 1 or 2). The tagging with dynamic context information cannot cope with this.

In addition, one would like for example to be able to define skeletons for binary operators like:

$$\boxed{\text{loose-bin-comb1}}\ \frac{\forall\, id \in s \cdot id \text{ binoprt } id : A}{(\varepsilon id \in s)_t \text{ binoprt } (\varepsilon id \in s)_t = (\varepsilon id \in \{e \text{ binoprt } e \mid e \in s\})_t}$$

and

$$\boxed{\text{loose-bin-comb2}}\ \frac{t_1 \neq t_2, \forall\, id_1 \in s_1, id_2 \in s_2 \cdot id_1 \text{ binoprt } id_2 : A}{\begin{array}{l}(\varepsilon id_1 \in s_1)_{t_1} \text{ binoprt } (\varepsilon id_2 \in s_2)_{t_2} = \\ (\varepsilon id \in \{e_1 \text{ binoprt } e_2 \mid e_1 \in s_1, e_2 \in s_2\})_{Combine(t_1,t_2)}\end{array}}$$

where "binoprt" is an arbitrary binary (infix) operator. The first rule states that if two identical loose values are combined the choice must only be performed once. The second rule express that for two different loose values all possible combinations must be considered in the new loose value.

The problem about rules like these is that one would like to be able to combine the different tags such that the traditional rules from set theory still hold. Consider a small example making use of the definitions of f and g from Section 3:

$$3 \times f() + g()$$

According to the dynamic semantics for VDM-SL this expression may yield 4, 5, 7, or 8 in the different legal models. Notice however that it cannot yield 6. The problem is now to let the tags reflect the chosen model for each of the functions even if the expression is rewritten using traditional rules from set theory forming e.g. $2 \times f() + g() + f()$. This turns out to be impossible unless one focuses on which choices that are made during "evaluation" instead of on the actual result of an expression.

One could also consider letting the *Combine* function (used in "loose-bin-comb2") simply maintaining a set of tags. One could then disallow combining looseness of choice values which had common elements unless both choice values are singleton sets (i.e. choice values which already rely on a combination of the semantics of some construct with another construct). This would save the example above by disallowing the user to combine the result of using traditional rules from set theory. However, this idea also does not work in all cases. Consider an expression like:

$$f() + \textsf{if } f() = 1 \textsf{ then } 2 \textsf{ else } 1$$

In all models this expression denotes 3, but the idea about letting *Combine* maintain a set of tags would indicate that this expression yields either 2 or 4. The problem is that even though the choice value corresponding to the if-then-else expression only rely on $f()$ it is in fact not using the models for $f()$ in its returned value. Thus, we can conclude that examples can be found where even a set of tags is insufficient for *Combine* to become unique.

The approach which tags loose values with dynamic context information can deal with combinations of arbitrary loose functions which have parameters. One can define a number of skeletons which correctly model the semantics for loose

expressions. However, it is not suitable for combinations of loose expressions inside the same function. In the next section we will investigate whether an approach which maintains all possible binding environments (and uses these as tags) are more suited to deal with this combination of loose expressions.

6 Maintaining all possible binding environments

The underlying strategy behind this approach is to record all possible bindings in which an expression can be evaluated. When looseness is introduced by either an expression or a pattern it is spread to another expression. In the approaches presented above the value of the expression into which the looseness had spread was "calculated" with the different possible models for the looseness. The idea behind this approach is to keep the expression to which the looseness is spreading, and then additionally record the different possible bindings which the looseness corresponds to. Let us first illustrate this principle on a few examples:

$$\mathsf{let}\ a = 3\ \mathsf{in}\ a + 2$$

For a simple example like this one would like to get something like:

$$\varepsilon\ a + 2\ \mathsf{with}\ \{\{a_{(\mathcal{POS},[args])} \mapsto 3\}\}$$

where the syntax of the loose values now first take the resulting expression followed by a new keyword, with, which again is followed by a set of possible binding environments (i.e. in this case a singleton set because no looseness is present). Notice that it is the pattern identifier a which is tagged. Let us now consider how it would look with a loose pattern:

$$\mathsf{let}\ l' \curvearrowright l'' = [1]\ \mathsf{in}\ l'$$

Here one would would get:

$$\begin{array}{l}\varepsilon\ l'\ \mathsf{with}\ \{\{l'_{(\mathcal{POS},[args])} \mapsto [\,], l''_{(\mathcal{POS},[args])} \mapsto [1]\},\\ \qquad\qquad \{l'_{(\mathcal{POS},[args])} \mapsto [1], l''_{(\mathcal{POS},[args])} \mapsto [\,]\}\}\end{array}$$

Let us now consider an example where looseness is combined (we take the example from page 11 again):

$$\begin{array}{l}\mathsf{let}\ a : \mathbb{N}\ \mathsf{be\ st}\ a \in \{1,2\}\ \mathsf{in}\\ \mathsf{let}\ b : \mathbb{N}\ \mathsf{be\ st}\ b \in \{3, a+2\}\ \mathsf{in}\ a + b\end{array}$$

Here one would first get:

$$\varepsilon\ \mathsf{let}\ b : \mathbb{N}\ \mathsf{be\ st}\ b \in \{3, a+2\}\ \mathsf{in}\ a + b\ \mathsf{with}\ \begin{array}{r}\{\{a_{(\mathcal{POS}_1,[args])} \mapsto 1\},\\ \{a_{(\mathcal{POS}_1,[args])} \mapsto 2\}\}\end{array}$$

and in the next step one would like to get:

$$\begin{array}{l}\varepsilon\ (\varepsilon\ a + b\ \mathsf{with}\ \{\{b_{(\mathcal{POS}_2,[args])} \mapsto 3\}, \{b_{(\mathcal{POS}_2,[args])} \mapsto a+2\}\})\ \mathsf{with}\\ \{\{a_{(\mathcal{POS}_1,[args])} \mapsto 1\}, \{a_{(\mathcal{POS}_1,[args])} \mapsto 2\}\}\end{array}$$

which one would like to be able to reduce to:

$$\varepsilon\ a + b \text{ with } \{\{a_{(\mathcal{POS}_1,[args])} \mapsto 1, b_{(\mathcal{POS}_2,[args])} \mapsto 3\},\\ \{a_{(\mathcal{POS}_1,[args])} \mapsto 2, b_{(\mathcal{POS}_2,[args])} \mapsto 3\},\\ \{a_{(\mathcal{POS}_1,[args])} \mapsto 2, b_{(\mathcal{POS}_2,[args])} \mapsto 4\}\}$$

This corresponds exactly to the desired result with the previous approach. In the same way this strategy can be used to the other examples of combining looseness which where mentioned in the previous section.

This completes the motivation for being interested in investigating how an approach which maintains all possible bindings is able to deal with looseness in explicit definitions. We will now try to explain how the corresponding proof rules will look and elaborate upon the limitation of this approach.

Let us now consider a specification which contains an ordinary let expression with a general pattern (which can be loose):

$$\text{let } pat = defexpr \text{ in } inexpr$$

then one can produce a proof rule following a skeleton like:

$$\boxed{\text{let-def'}}\ \frac{\exists\, \text{patids} : \text{types inferred} \cdot pat' = defexpr}{\begin{array}{c}(\text{let } pat = defexpr \text{ in } inexpr)_{(\mathcal{POS},[args])} =\\ \varepsilon inexpr \text{ with } PatternMatch(pat_{(\mathcal{POS},[args])}, defexpr)\end{array}}$$

where *PatternMatch* is a function which given a pattern, *pat*, and an expression, *defexpr*, yields the set of all possible binding environments arising from such a matching, i.e. a set of maps from pattern identifiers (tagged with dynamic information) to their corresponding value. This function will be quite similar to the *EvalPattern* from [Larsen92] and to the *PatternMatch* function from [Lassen&91]. This function must also have access to the environment which currently is in context.

However, as it is visible in [Larsen92] *PatternMatch* may in general yield an infinite set of possible bindings. Thus, if one wanted to support such an approach with a tool it would probably be necessary to restrict its use to an executable subset (like the one used by the IFAD VDM-SL interpreter[15]). One could envisage a looseness analysis tool which given a specification and an arbitrary expression (using constructs from the specification) would yield a set of all possible results which the expression may evaluate to, and under which assumptions (which binding choices) each result is obtained. Possibly one could even let the expression be so general that it simply was symbolically executed. However, as the results of [Kneuper89] also show it is very difficult to reduce the produced symbolic expressions.

7 Turning an explicit definition into an implicit one

Another approach which one also might attempt is to transform an explicit definition into an implicit one which is semantically equivalent. The advantage

[15] See [Larsen&91], or [Lassen93].

about the implicit function definitions is that it simply describes the intended relationship between input and output. Thus, when looseness is present it will not be possible to say explicitly what the output will be, but simply what it will fulfill. The problem with looseness in explicitly definitions is exactly that such definitions in an explicitly manner yield a result and that such explicit loose definitions can be directly combined.

However, this is not always possible, because explicitly defined functions in VDM-SL are given a least fixed point semantics while implicitly defined functions are given an all fixed point semantics treating the definition as a kind of equation[16]. In this section we will try to indicate under which restrictions this idea can be used.

For total first order functions (which do not take other functions as arguments or returns other functions) the least fixed point is the only fixed point and thereby all such explicitly defined functions can be turned into an equivalent implicit function. However, partial functions (which may return bottom even for domain values which satisfy the pre-condition) cannot be transformed to an implicit equivalents. Implicitly defined functions cannot return bottom when the pre-condition is satisfied. Higher order explicitly defined total functions could in principle be turned into an implicit equivalent. However, this requires that the argument (or result) functions are total as well. Otherwise, the least fixed point is not the only fixed point.

8 Concluding Remarks

Looseness in explicit definitions has not previously been given a systematic and formal treatment in the form of proof rules reflecting the intended semantics. Thus, this is what we consider the main result of the work reported in this paper. Unfortunately, this research indicates that all the approaches have limitations (in particular if one wishes to provide tool support for it). However, even with these limitations we believe that the proof rules presented here is a step forward in obtaining proof rules for full VDM-SL.

We think that the proof rule skeletons shown in this paper are very complex in order to reflect the semantics of VDM-SL. However, we also believe that the extensive amount of tagging can be hidden for a user by a supporting verification tool. Such a tool can be used to keep track of all the tags and in this way ensure that the proof rules only are applied in a consistent way. An expert user should then be allowed to look at the tags during a proof to understand why certain rules cannot be used for a given derived expression. More work will be put into this in the future to investigate the possibilities of making such complicated proof rules usable by tool support.

Acknowledgements

We would like to thank Cliff Jones and Richard Moore for arranging a one-weeks visit to Manchester University in November 1992 where the core ideas of the research reported here was developed. Furthermore we would like to thank

[16] Since no functional is present they cannot be treated in a least fixed point manner.

the anonymous referees, Kees de Bruin, Bo Stig Hansen and especially John Fitzgerald for their influencing suggestions to improve this paper.

References

[**Bicarregui&93**] Juan Bicarregui, John Fitzgerald, Peter Lindsay, Richard Moore and Brian Ritchie. *Proof in VDM: A Practitioner's Guide.* Springer-Verlag, December 1993.

[**Bruun&91**] Hans Bruun, Flemming Damm and Bo Stig Hansen. An Approach to the Static Semantics of VDM-SL. In *VDM '91: Formal Software Development Methods*, VDM Europe, Springer-Verlag, October 1991.

[**Dijkstra&90**] Edsger W. Dijkstra, Carel S. Scholten. *Predicate Calculus and Program Semantics*, chapter 4, pages 21–29. Springer-Verlag, 1990.

[**Fitzgerald&93**] John Fitzgerald and Richard Moore. Experiences in Developing a Proof Theory for VDM Specifications. In *Proceedings of the International Workshop on Semantics of Specification Languages, Utrect, October 1993*, Springer Verlag, 1994. 17 pages.

[**Fitzgerald93**] John Fitzgerald. *Private Communication.* March 1993.

[**Hayes93**] Ian Hayes. *Private Communication.* June 1993.

[**Jones&91**] Cliff Jones, Kevin Jones, Peter Linsay and Richard Moore, editors. *mural: A Formal Development Support System.* Springer-Verlag, 1991. 421 pages.

[**Kneuper89**] Ralf Kneuper. *Symbolic Execution as a Tool for Validation of Specifications.* PhD thesis, Department of Computer Science, Univeristy of Manchester, 1989. 154 pages. Technical Report Series UMCS-89-7-1.

[**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, IFIP, North-Holland, 1989.

[**Larsen&91**] Peter Gorm Larsen and Poul Bøgh Lassen. An Executable Subset of Meta-IV with Loose Specification. In *VDM '91: Formal Software Development Methods*, VDM Europe, Springer-Verlag, October 1991.

[**Larsen92**] Peter Gorm Larsen. *The Dynamic Semantics of the VDM Specification Language.* Technical Report, The Institute of Applied Computer Science, July 1992. 177 pages.

[**Lassen&91**] Poul Bøgh Lassen, Kees de Bruin and Peter Gorm Larsen. *The Dynamic Semantics of IFAD VDM-SL.* June 1993. Internal document (IFAD)

[**Lassen93**] Poul Bøgh Lassen. IFAD VDM-SL Toolbox. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, page 681, Springer-Verlag, Berlin Heidelberg, April 1993. 1 page.

[**Middelburg90**] Kees Middelburg. *Syntax and Semantics of VVSL – A Language for Structured VDM Specifications.* PhD thesis, University of Amsterdam, 1989. 395 pages.

[**RAISE94**] Chris George and Søren Prehn. *The RAISE Justification Handbook. The BCS Practitioners Series*, To be published by Prentice-Hall, 1994. 205 pages.

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