# Formalizing a Subset of VDM-SL in HOL

Sten Agerholm  Kim Sunesen

IFAD, Forskerparken 10
DK-5230 Odense M, Denmark

May 20, 1999

**Project Id.:** PROSPER LTR 26241
**Deliverable Id.:** D1.2a.V1
**Document Id.:** IFAD-PROSPER-DOC-2
**Availability:** CONFIDENTIAL

p. 25 — 4.1.2 important

$x = p \Rightarrow p = x$

# Contents

# Chapter 1

# Introduction

This document describes the formalization of VDM-SL specifications in the HOL98 theorem prover. It presents detailed requirements for an algorithm for translating a large declarative subset of VDM-SL to concrete HOL98 commands and terms.

Our ultimate aim is to provide proof support for a large subset of the VDM Specification Language. However, translating VDM-SL specifications to HOL is not straightforward because of significant incompatibilities in the underlying theories. Therefore a substantial number of choices and restrictions must be made in designing the translation algorithm. For example, specifications in VDM-SL are traditionally expressed using the three-valued Logic of Partial Functions [9, 12, 10, 2] whereas our use of HOL98 requires a two-valued logic. We therefore restrict our consideration to VDM-SL specifications expressed within a two-valued subset. The subset, called $\text{VDM}_{dec}$, is defined in [6]. It is our experience that the majority of specifications developed in commercial practice adhere to a two-valued subset and that deviation from it is often considered an error to be resolved. Adherence to the two-valued subset can typically be ensured by verifying that the specification respects a number of logical conditions called *proof obligations* (see [6]). Current work is aimed at automating the proof of these obligations as a prelude to validation of the specification. The present translation strategy is based in part on the experiences obtained from the manual translations in [4], which we did for the VDM-SL specifications in [7], and in part on our previous experiences with the translation of VDM-SL specifications to PVS [1] and to Isabelle [3].

This document supplies enough detail to support manual translations, but its main purpose is to provide the path towards a detailed formal specification of the translation strategy in VDM-SL. The translation approach to formalizing VDM-SL specifications in HOL can be viewed as a shallow embedding of VDM-SL specifications in HOL [11]. The syntax of VDM-SL constructs is not embedded: we work with the "semantics" of the constructs directly. Thus a number of semantic decisions are built into the translator. We believe that this is satisfactory for our purposes, since many constructs are similar in VDM-SL

and HOL, and so the translation is direct in many cases.

The rest of this report is structured as follows. In Chapter 2, we give an overview of the general strategy for translating VDM-SL specifications into HOL98. In Chapter 3, we introduce $\text{VDM}_{bas}$, a basic subset of VDM-SL and show how to translate specifications in $\text{VDM}_{bas}$ into HOL98. In Chapter 4 we extend $\text{VDM}_{bas}$ with more advanced features from the full $\text{VDM}_{dec}$ language. In Chapter 5, we give initial views on ongoing technical issues, including the translation of recursive functions and the representation of restricted forms of union types and recursive types.

The reader is assumed to be familiar with VDM-SL [14], HOL [15], and the HOL98 system.

# Chapter 2

# Translation strategy

This chapter gives an overview of the translation of VDM-SL specifications (more precisely, specifcations in the $\text{VDM}_{dec}$ subset of VDM-SL [6]) into the higher order logic of the HOL98 system. The definitions in specifications are translated one by one in a compositional fashion to a representation in HOL to achieve a shallow embedding in the terminology of [11]. We also discuss how incompatibilities between VDM-SL and HOL are solved by imposing a number of restrictions on specifications.

## 2.1 Translation steps

We envisage that the translation process includes the following four steps:

**Parsing:** the $\text{VDM}_{dec}$ specification is parsed into an intermediate representation in terms of an abstract syntax tree (AST).

**Type checking & proof obligation generation:** the AST is type checked. During the type checking, proof obligations are generated in an intermediate representation which at the end of type checking is added to the AST. These can be pretty-printed using concrete VDM-SL syntax for immediate presentation. The proof obligation generation process is described in [6].

**Transformation:** Those $\text{VDM}_{dec}$ constructions that can be transformed into simpler constructions are transformed. These include the elimination of patterns, set bindings, cases expressions and implicit function definitions. At the end of the transformation process, the specification is in a subset of $\text{VDM}_{dec}$ called $\text{VDM}_{bas}$. The $\text{VDM}_{bas}$ language is described in Chapter 3. The transformation of $\text{VDM}_{dec}$ constructs into $\text{VDM}_{bas}$ is decribed in Chapter 4.

**Translation:** a two-phase activity. In the first phase, a dependency analysis is performed on the transformed specification produced by the previous step and forward referencing is resolved. Then a recursive descent of the AST

is performed while translating the $\text{VDM}_{bas}$ abstract syntax into a concrete HOL98 syntax representation, which can be read by HOL98 directly.

## 2.2 Incompatibilites

There are incompatibilities between VDM-SL and HOL98 in three main areas:

**Proof theory:** The traditional proof theory of VDM-SL is based on the non-classical, three-valued Logic of Partial Functions (LPF) [9, 12], whereas HOL supports a standard classical logic.

**Type theory:** The type system of VDM-SL supports subtyping and union types while the type system of HOL does not directly support subtyping and non-disjoint union types. Also, type equivalence in VDM-SL is structural whereas the type definition principle of HOL uses by-name equivalence.

**Under-determinedness:** In VDM-SL, under-determinedness is used to interpret looseness (or choice). This breaks the reflexivity of equality.

In the following sections, we present an approach to handle each of these incompatibilities via a number of restrictions on specifications, while still supporting a large and useful declarative subset of VDM-SL.

### 2.2.1 From partiality to totality

The higher order logic HOL is a logic of total functions and does not support partial functions directly as VDM-SL does. Hence, partiality must be replaced by totality.

**Partial operators:** The proof obligations generated for each VDM-SL specification as discussed in [6] allow us to translate partial operators in VDM-SL to their total counterparts in HOL. The proof obligations guarantee that the partial operators are never applied to values outside their domain, so here their definition is unimportant and can be anything.

**Non-recursive functions:** The body of a syntactically non-recursive function is translated into a HOL expression denoting a total function. Again, the proof obligations guarantee that such a function is not applied outside its domain.

**Recursive functions:** Syntactically recursive functions are not guaranteed to terminate and the proof obligations, as described in [6], do not presently capture termination. Instead we use HOL to prove termination of recursive functions as they cannot be introduced in a guaranteed consistent way without such a proof.

The use of HOL to prove termination affects the translation strategy. Consider the following VDM-SL specification:

```
A = nat
inv x == x <> 0;

f(x,y):  A * nat -> nat
f(x,y) = if x <> 0 and y<>0 then x + y else f(x,y)
pre y <> 0;
```

The domain of `f` is the set of pairs $(x, y)$ of natural numbers where both $x$ and $y$ are nonzero. A syntactic analysis of the definition of `f` cannot show termination, but a semantical analysis can since the else branch of the body of `f` is guaranteed never to be executed. Hence, the direct translation of the definition into

```
f (x,y) = if x <> 0 and y<>0 then x + y else f(x,y);
```

is not a total function and hence not definable in HOL. Instead we need to introduce the domain guards from the type invariant and the precondition explicitly in the translation

```
f (x,y) = if inv_A(x) and pre_f(x,y) then
            if x <> 0 and y<>0 then x + y else f(x,y)
          else ARBITRARY;
```

Hence, yielding a function which is guaranteed to terminate on the unrestricted domain `nat * nat`.

### 2.2.2 Subtypes

The use of union types and invariants is intrinsic to VDM-SL. Since HOL does not support subtypes directly, we translate type definitions and invariant definitions separately to HOL. All subtype checking is performed in proof obligations generated at the VDM-SL level, and the invariant predicates are inserted explicitly in the translation process (see Chapter 3.5 for further details).

### 2.2.3 Union types

VDM-SL's non-disjoint union types are not supported. We require that a union type is only used with record types or quote types, allowing a translation to an abstract datatype definition in HOL98.

### 2.2.4 Under-determinedness

VDM-SL provides a number of constructs with a loose interpretation, for example the let-be-such-that expression: `let x in set s be st p in e`. The concept of under-determinedness is used to give semantics to such constructs. As a consequence of the interpretation of under-determinedness in VDM-SL,

distinct occurrences of an expression do not necessarily denote the same value. In other words, equality on expressions is not reflexive, that is, for instance `let x in set {1,2,3} be st x <> 2 in x` is not provably equal to itself. This interpretation makes good sense for instance when using under-determinedness in connection with underspecified specifications where a chain of specifications is gradually refined towards an implementation, see [16] for details. We shall employ an interpretation using Hilbert's choice operator to formalize under-determinedness. The choice operator hence gives an undetermined choice but always the same one. In particular, reflexivity of equality is preseved.

The faithful support of the VDM-SL interpretation of under-determinedness in HOL appears to be complex. Moreover, it is not clear to what extent the difference in interpretation will be a problem in practice. The VDM-SL interpreter of IFAD interprets under-determinedness by means of a randomised choice.

# Chapter 3

# Translating a basic subset of VDM-SL

In this chapter, we define $\text{VDM}_{bas}$, a basic subset of VDM-SL. We show how $\text{VDM}_{bas}$ specifications can be translated into the higher-order logic of the HOL98 system. Section 3.1 describes the $\text{VDM}_{bas}$ language and Sections 3.2 to 3.5 describe the process of translating the language to HOL98.

The purpose of the chapter is to discuss how the basic concepts of VDM-SL can be translated. In particular, the basic concepts include a nearly complete subset of the rich palette of VDM-SL operators and constructors for both basic types and compound types like sequence types, set types, and map types.

## 3.1 The language $\text{VDM}_{bas}$

The $\text{VDM}_{bas}$ language is a simple functional specification language with a rich language of operators and constructors.

Let $x, x_1, \ldots$ range over variables, $c$ over constants, $f, f_1, \ldots$ over function variables, $t_1, \ldots$ over type variables, and $A, a, a_1, \ldots$ over identifiers. Terms $e$, $e_1$, and $e_2$ of $\text{VDM}_{bas}$ take the following syntax

$$
\begin{array}{rcl}
e & ::= & x \mid c \mid C_j(e_1, \ldots, e_{k_j}) \mid \texttt{<}a\texttt{>} \mid \texttt{RESULT} \mid \\
& & op\ e \mid e_1\ op\ e_2 \mid f_i(e_1, \ldots, e_{n_i}) \\
& & e.a \mid e.\#c \mid e_1(e_2) \mid \\
& & \texttt{forall}\ x : \sigma\ \texttt{\&}\ e \mid \texttt{iota}\ x : \sigma\ \texttt{\&}\ e \mid \\
& & \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \\
& & \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \mid \\
& & \texttt{let}\ x : \sigma\ \texttt{be st}\ e_1\ \texttt{in}\ e_2
\end{array}
$$

The rich language of operators $op$ and value constructors $C_j(e_1, \ldots, e_{k_j})$, including enumerations and comprehensions is described in Sections 3.3 and 3.4

below.

Readers familiar with ISO Standard VDM-SL may not recognise RESULT, which is the symbol used in post-conditions of explicit function definitions to stand for the function body. Also unfamiliar is the general projection operator on tuples $e.\#c$, an extension to VDM-SL introduced for the purposes of the Prosper tools (the introduction is explained in [6] and [8].

For convenience, we shall use multiple bindings in quantifier and iota expressions as a short hand for the corresponding expansions

$$\begin{aligned}
\texttt{forall } x_1:\sigma_1,\ldots,x_k:\sigma_k \texttt{ \& } e &\stackrel{\text{def}}{=} \texttt{forall } x_1:\sigma_1 \texttt{ \& } \ldots \texttt{ forall } x_k:\sigma_k \texttt{ \& } e \\
\texttt{exists } x:\sigma \texttt{ \& } e &\stackrel{\text{def}}{=} \texttt{not forall } x:\sigma \texttt{ \& not } e \\
\texttt{exists } x_1:\sigma_1,\ldots,x_k:\sigma_k \texttt{ \& } e &\stackrel{\text{def}}{=} \texttt{exists } x_1:\sigma_1 \texttt{ \& } \ldots \texttt{ exists } x_k:\sigma_k \texttt{ \& } e
\end{aligned}$$

Similarly for convenience, for let bindings, we shall use the following short hand for its expansion.

$$\texttt{let } x_1 = e_1,\ldots,x_k = e_k \texttt{ in } e \stackrel{\text{def}}{=} \texttt{let } x_1 = e_1 \texttt{ in } \ldots \texttt{ let } x_k = e_k \texttt{ in } e$$

We shall only translate well-typed VDM-SL terms into HOL98, and thus in particular, we shall only translate well-typed $\text{VDM}_{bas}$ terms into HOL98. The type language of $\text{VDM}_{bas}$ is given by the following syntax

$$\begin{aligned}
\gamma \;::=\; & \sigma \mid \texttt{<}a_1\texttt{>} \mid \texttt{<}a_2\texttt{>} \mid \\
& \texttt{compose } a \texttt{ of } a_1:\sigma_1 \;\; a_2:\sigma_2 \texttt{ end} \\
\\
\sigma \;::=\; & \texttt{token} \mid \texttt{bool} \mid \texttt{char} \mid \texttt{nat} \mid \texttt{<}a\texttt{>} \mid t_i \mid \sigma_1 \texttt{*} \sigma_2 \mid \\
& \texttt{set of } \sigma \mid \texttt{seq of } \sigma \mid \texttt{map } \sigma_1 \texttt{ to } \sigma_2
\end{aligned}$$

The syntax of the type language is defined in two levels in order to syntactically rule out nested composite types and enumeration types within composite types. As explained in Section 3.3, we do this to get a more direct translation of enumeration types and composite type into HOL98. However, both nested composite types and enumeration types within composite types can be achieved indirectly via the declaration of type variables as explained below. The restriction is hence not severe and can be surrounded by introducing new type variables. In the following, $\gamma, \gamma_1, \ldots$ range over the set of type expressions and $\sigma, \sigma_1, \ldots$, range over the subset of type expressions defined by $\sigma$ above.

The meaning of function variables is given by a list of function declarations

of the form

$$
\begin{array}{l}
f_1 : \sigma_{11} * \ldots * \sigma_{1a_1} \texttt{->} \sigma_1 \\
f_1(x_{11}, \ldots, x_{1a_1}) == e_1 \\
\texttt{pre } e_1' \\
\texttt{post } e_1''; \\
\quad \vdots \\
f_n : \sigma_{n1} * \ldots * \sigma_{na_n} \texttt{->} \sigma_n \\
f_n(x_{n1}, \ldots, x_{na_n}) == e_n \\
\texttt{pre } e_n' \\
\texttt{post } e_n'';
\end{array}
$$

where for each $i = 1, \ldots, n$ the free variables of the expressions $e_i$, $e_i'$, and $e_i''$ are contained in the set of formal parameters $\{x_{i1}, \ldots, x_{ia_i}\}$. As in VDM-SL, each declaration consists of four parts; a type annotation, a function definition, a precondition, and a postcondition. We disallow recursive declarations, that is, for each $i, j = 1, \ldots, n$, $e_i$, $e_i'$, and $e_i''$ can only refer backwards to function variables $f_j$ such that $j < i$. Furthermore, we disallow composite types and enumeration types in the type annotation.

Similarly, the meaning of type variables is given by a list of type declarations

$$
\begin{array}{c}
t_1 = \gamma_1 \\
\vdots \\
t_m = \gamma_m
\end{array}
$$

where $\gamma_1, \ldots, \gamma_m$ are type expressions. Again, we disallow recursive declarations and forward references, that is, for each $i, j = 1, \ldots, m$, $\gamma_i$ can only refer backwards to type variables $t_j$ such that $j < i$.

As a subset of VDM-SL, $\text{VDM}_{bas}$ inherits the type system of VDM-SL, and hence, the notion of well-typedness. Throughout the report, we shall only be concerned with well-typed VDM-SL specifications. The translation function will utilise a partial function $\Gamma$ from variables to types to represent the environment of type definitions. For each $\Gamma$, we shall assume a function $\tau_\Gamma$ on well-typed expressions that given an expression $e$ computes its type. As usual, we define the update of a partial function $\Gamma$ by

$$
\Gamma[a \leftarrow b](x) = \begin{cases} b & \text{if } x = a \\ \Gamma(x) & \text{otherwise} \end{cases}
$$

We use $\emptyset$ to denote the partial function which is undefined everywhere.

Also, we need a function $FV$ to compute the set of free variables in an expression.

$$
\begin{array}{rcl}
FV(x) & = & \{x\} \\
FV(c) = FV(\texttt{<}a\texttt{>}) = FV(\texttt{RESULT}) & = & \emptyset \\
FV(C_j(e_1, \ldots, e_{k_j})) & = & \bigcup_{i=1,\ldots,k_j} FV(e_i)
\end{array}
$$

$$
\begin{aligned}
FV(f_i(e_1,\ldots,e_{n_i})) &= \bigcup_{j=1,..,n_i} FV(e_j) \\
FV(op\ e) &= FV(e) \\
FV(e_1\ op\ e_2) &= FV(e_1) \cup FV(e_2) \\
FV(e.a) = FV(e.\#c) &= FV(e) \\
FV(e_1(e_2)) &= FV(e_1) \cup FV(e_2) \\
FV(\texttt{forall}\ x:\sigma\ \texttt{\&}\ e) &= FV(e)\text{-}\{x\} \\
FV(\texttt{iota}\ x:\sigma\ \texttt{\&}\ e) &= FV(e)\text{-}\{x\} \\
FV(\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2) &= FV(e_1) \cup (FV(e_2)\text{-}\{x\}) \\
FV(\texttt{let}\ x:\sigma\ \texttt{be st}\ e_1\ \texttt{in}\ e_2) &= FV(e_1) \cup (FV(e_2)\text{-}\{x\}).
\end{aligned}
$$

## 3.2 Translating VDM$_{bas}$

We are now ready to translate VDM$_{bas}$ specifications into HOL98. The translation is based on two functions; ⟪-⟫ for types and ⟦-⟧ for expressions.

The function ⟪-⟫ for translating types defines a fairly direct translation. Further explanations are given in Section 3.3 and Section 3.4 below. Its inductive definition is

$$
\begin{aligned}
\langle\!|\texttt{token}|\!\rangle &= \texttt{ind} \\
\langle\!|\texttt{bool}|\!\rangle &= \texttt{bool} \\
\langle\!|\texttt{char}|\!\rangle &= \texttt{ascii} \\
\langle\!|\texttt{nat}|\!\rangle &= \texttt{num} \\
\langle\!|\texttt{<}a\texttt{>}|\!\rangle &= a \\
\langle\!|t_i|\!\rangle &= t_i \\
\langle\!|\texttt{compose}\ a\ \texttt{of}\ a_1:\sigma_1\ a_2:\sigma_2\ \texttt{end}|\!\rangle &= \texttt{mk\_}a\ \texttt{of}\ \langle\!|\sigma_1|\!\rangle\ \texttt{\#}\ \langle\!|\sigma_2|\!\rangle \\
\langle\!|\sigma_1\texttt{*}\sigma_2|\!\rangle &= \langle\!|\sigma_1|\!\rangle\ \texttt{\#}\ \langle\!|\sigma_2|\!\rangle \\
\langle\!|\texttt{<}a_1\texttt{>|<}a_2\texttt{>}|\!\rangle &= a_1\ \texttt{|}\ a_2 \\
\langle\!|\texttt{set of}\ \sigma|\!\rangle &= \langle\!|\sigma|\!\rangle \rightarrow \texttt{bool} \\
\langle\!|\texttt{seq of}\ \sigma|\!\rangle &= \langle\!|\sigma|\!\rangle\ \texttt{list} \\
\langle\!|\texttt{map}\ \sigma_1\ \texttt{to}\ \sigma_2|\!\rangle &= (\langle\!|\sigma_1|\!\rangle,\langle\!|\sigma_2|\!\rangle)\texttt{fmap}
\end{aligned}
$$

The function ⟦-⟧ for translating well-typed VDM$_{bas}$ expressions also defines a fairly direct translation which needs little justification. The functions *constant*, *constructor*, *operator* and $proj_m$, which translate constants, value constructors, operators and tuple projection respectively, are discussed in Section 3.3 and Section 3.4 below. The inductive definition of ⟦-⟧ is given by

$$
\begin{aligned}
[\![x]\!]_\Gamma &= x \\
[\![c]\!]_\Gamma &= constant(c) \\
[\![C_j(e_1,\ldots,e_{k_j})]\!]_\Gamma &= constructor(C_j([\![e_1]\!]_\Gamma,\ldots,[\![e_{k_j}]\!]_\Gamma))
\end{aligned}
$$

$$
\begin{array}{rcl}
[\![\texttt{<}a\texttt{>}]\!]_\Gamma & = & a \\
[\![\texttt{RESULT}]\!]_\Gamma & = & \texttt{RESULT} \\
[\![op\ e]\!]_\Gamma & = & operator(op)\ [\![e]\!]_\Gamma \\
[\![e_1\ op\ e_2]\!]_\Gamma & = & [\![e_1]\!]_\Gamma\ operator(op)\ [\![e_2]\!]_\Gamma \\
[\![f_i(e_1,\ldots,e_{n_i})]\!]_\Gamma & = & f_i([\![e_1]\!]_\Gamma,\ldots,[\![e_{n_i}]\!]_\Gamma) \\
[\![e.a]\!]_\Gamma & = & A\_a([\![e]\!]_\Gamma), \text{ where } \tau_\Gamma(e) = A \\
[\![e.\#c]\!]_\Gamma & = & proj_m([\![e]\!]_\Gamma), \\
 & & \text{where } c \text{ denotes the natural number } m \\
[\![e_1(e_2)]\!]_\Gamma & = & \begin{cases} [\![e_1]\!]_\Gamma([\![e_2]\!]_\Gamma) & \text{if } \tau_\Gamma(e_1) = \sigma_1 \to \sigma_2 \\ \texttt{FAPPLY}\,[\![e_1]\!]_\Gamma\,[\![e_2]\!]_\Gamma & \text{if } \tau_\Gamma(e_1) = \text{map } \sigma_1 \text{ to } \sigma_2 \\ \texttt{SEQAPPLY}\,[\![e_1]\!]_\Gamma\,[\![e_2]\!]_\Gamma & \text{if } \tau_\Gamma(e_1) = \text{seq of } \sigma \end{cases} \\
[\![\texttt{forall}\ x:\sigma\ \texttt{\&}\ e]\!]_\Gamma & = & !x:(\!|\sigma|\!).\,[\![e]\!]_{\Gamma[x\leftarrow\sigma]} \\
[\![\texttt{iota x:}\sigma\ \texttt{\&}\ e]\!]_\Gamma & = & \texttt{@x:}(\!|\sigma|\!).\,[\![e]\!]_{\Gamma[x\leftarrow\sigma]} \\
[\![\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!]_\Gamma & = & \texttt{COND}\ ([\![e]\!]_\Gamma,[\![e_1]\!]_\Gamma,[\![e_2]\!]_\Gamma) \\
[\![\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2]\!]_\Gamma & = & \texttt{let}\ x\ \texttt{=}\ [\![e_1]\!]_\Gamma\ \texttt{in}\ [\![e_2]\!]_{\Gamma[x\leftarrow\tau_\Gamma(e_1)]} \\
[\![\texttt{let x:}\sigma\ \texttt{be st}\ e_1\ \texttt{in}\ e_2]\!]_\Gamma & = & \texttt{let x = @x:}(\!|\sigma|\!).\ [\![e_1]\!]_{\Gamma[x\leftarrow\sigma]}\ \texttt{in}\ [\![e_2]\!]_{\Gamma[x\leftarrow\sigma]}
\end{array}
$$

The final definition above shows the translation of a loose binding (as introduced in Section 2.2.4) in terms of the Hilbert epsilon operator.

The translation of $\text{VDM}_{bas}$ specifications is then accomplished by the translation of type and function declarations. A type declaration

$$t_i = \gamma_i$$

is translated to

$$\texttt{vdm\_datatype`}\,t_i = (\!|\gamma_i|\!)\texttt{`;}$$

Presently, we have not defined an SML function `vdm_datatype`. In the manual translations done in [4], we translate enumeration types and composite types using the function `Hol_datatype`

$$\texttt{Hol\_datatype`}\,t_i = (\!|\gamma_i|\!)\texttt{`;}$$

and all other types using anti-quotations

$$\texttt{val}\ t_i\ \texttt{=}\ \texttt{ty\_antiq(Type`:}(\!|\gamma_i|\!)\texttt{`);}$$

A function declaration

$$
\begin{array}{l}
f_i : \sigma_{i1} \texttt{*} \ldots \texttt{*} \sigma_{ia_i} \texttt{->}\ \sigma_i \\
f_i(x_{ii},\ldots,x_{ia_i}) \texttt{==}\ e_i \\
\texttt{pre}\ \ e_i' \\
\texttt{post}\ e_i'';
\end{array}
$$

is translated to

$$\texttt{pred}\ `pre\_f_i(x_{i1} : \langle\!|\sigma_{i1}|\!\rangle, \ldots, x_{ia_i} : \langle\!|\sigma_{ia_i}|\!\rangle) = (\llbracket e_i' \rrbracket_\Gamma : \texttt{bool})`;$$

$$\texttt{pred}\ `post\_f_i(x_{i1} : \langle\!|\sigma_{i1}|\!\rangle, \ldots, x_{ia_i} : \langle\!|\sigma_{ia_i}|\!\rangle, \texttt{RESULT} : \langle\!|\sigma_i|\!\rangle) = (\llbracket e_i'' \rrbracket_\Gamma : \texttt{bool})`;$$

$$\texttt{func}\ `f_i(x_{i1} : \langle\!|\sigma_{i1}|\!\rangle, \ldots, x_{ia_i} : \langle\!|\sigma_{ia_i}|\!\rangle) = (\llbracket e_i \rrbracket_\Gamma : \langle\!|\sigma_i|\!\rangle)`;$$

where $\Gamma = \emptyset[x_{i1} \leftarrow \sigma_{i1}, \ldots, x_{ia_i} \leftarrow \sigma_{ia_i}]$. The SML functions `pred` and `func` are defined using the `Tfl` function `Rfunction`.

Presently, they are defined by

```
fun pred qtm = Rfunction `measure (\x.7)` qtm;

fun func qtm = Rfunction `measure (\x.7)` qtm;
```

The measures are arbitrary and simply reflect the fact that the functions defined using `pred` and `func` are non-recursive, and hence always terminate. We use special functions for the definitions of predicates and general non-recursive functions because they play a special role when reasoning about specifications. In the case studies on reasoning about VDM-SL specifications reported in [4], we manually separated definitions into lists of predicates, projections, non-recursive functions, and recursive functions. Later, we will update the definition of `pred func` to add the returned HOL98 definitions to special lists of predicates and functions.

For the non-recursive functions of $\text{VDM}_{bas}$, there is no reason for defining the pre- and post-conditions of a function before the function itself. But later, when we consider recursive functions and implicitly defined functions this will be important. Thus, we enforce the reordering at this point to ensure a conservative extension of the translation to larger subsets of VDM-SL.

## 3.3 Translating basic types

This section lays down the basic requirements for the translation of the $\text{VDM}_{bas}$ basic types into HOL98. For each type we briefly discuss its representation in HOL98. We then list the constants, constructors and operators of the type and give an indication of their translation in HOL. The basic types are the token type `token`, the Boolean type `bool`, the character type `char`, the natural numbers `nat`, the positive natural numbers `nat1`, the integers `int`, the rational numbers `rat`, the real numbers `real`, and the quote type.

### 3.3.1 The token type

The token type `token` consists of a countably infinite set of distinct values. We translate `token` into the `ind` type of HOL98:

$$\langle\!|\texttt{token}|\!\rangle \quad = \quad \texttt{ind}$$

The only operations that can be carried out on tokens are equality and inequality, as given in the following table. Both of these are also supported by `ind`.

| Operator | Name | Type |
|---|---|---|
| `s = t` | Equal | token * token → bool |
| `s <> t` | Not equal | token * token → bool |

### 3.3.2 The Boolean type

The `bool` type of $\text{VDM}_{bas}$ (VDM-SL) is translated into the `bool` type of HOL98:

$$\langle\!|\texttt{bool}|\!\rangle \quad = \quad \texttt{bool}$$

The operators which we need to translate are given in the following table. The `boolTheory` supports all of these though the syntax is different in most cases.

| Operator | Name | Type |
|---|---|---|
| `not b` | Negation | bool → bool |
| `a and b` | Conjunction | bool * bool → bool |
| `a or b` | Disjunction | bool * bool → bool |
| `a => b` | Implication | bool * bool → bool |
| `a <=> b` | Biimplication | bool * bool → bool |
| `a = b` | Equality | bool * bool → bool |
| `a <> b` | Inequality | bool * bool → bool |

### 3.3.3 The character type

The character type `char` is translated to the `ascii` type of HOL98:

$$\langle\!|\texttt{char}|\!\rangle \quad = \quad \texttt{ascii}$$

We have not yet checked whether the `ascii` type supports all the values supported by `char` (Appendix A). The only operations that can be carried out on characters are equality and inequality. Both of these are also supported by `ascii`.

| Operator | Name | Type |
|---|---|---|
| `c1 = c2` | Equal | char * char → bool |
| `c1 <> c2` | Not equal | char * char → bool |

### 3.3.4 The numeric types

There are five basic numeric types: positive naturals, naturals, integers, rationals and reals. Except for three, all the numerical operators can have mixed

operands of the three types. The exceptions are integer division, modulo and the remainder operation.

Presently, no theories for integers, rationals, or reals have been fully ported to HOL98. In the manual translations of examples reported in [4], we have therefore replaced the the VDM$_{bas}$ type `real` by `nat`. It is recognised that this will not be adequate in future:

The natural number type `nat` is translated to the HOL98 type `num`:

$$(\!|\texttt{nat}|\!) \quad = \quad \texttt{num}$$

The positive natural number type `nat1` is not supported by VDM$_{bas}$. In Section 3.5, we extend VDM$_{bas}$ with type invariants. With type invariants `nat1` is naturally translated as `nat` with an additional invariant stating the non-zeroness.

The operators which we need to translate are given in the following table. The theories connected to `num` do not support all of these and will have to be extended.

| Operator | Name | Type |
|---|---|---|
| `-x` | Unary minus | real → real |
| `abs x` | Absolute value | real → real |
| `floor x` | Floor | real → int |
| `x + y` | Sum | real * real → real |
| `x - y` | Difference | real * real → real |
| `x * y` | Product | real * real → real |
| `x / y` | Division | real * real → real |
| `x div y` | Integer division | int * int → int |
| `x rem y` | Remainder | int * int → int |
| `x mod y` | Modulus | int * int → int |
| `x**y` | Power | real * real → real |
| `x < y` | Less than | real * real → bool |
| `x > y` | Greater than | real * real → bool |
| `x <= y` | Less or equal | real * real → bool |
| `x >= y` | Greater or equal | real * real → bool |
| `x = y` | Equal | real * real → bool |
| `x <> y` | Not equal | real * real → bool |

### 3.3.5 Quote types

For an identifier $a$, a quote type `<a>` has exactly one element, which is also called `<a>`. Any quote type `A = <a>` can be translated to HOL98 using the `Hol_datatype` definition

```
Hol_datatype `A = a`;
```

For each quote type `<a>`, only the equality and inequality operators are supported. Both of these are supported by theories supporting the `Hol_datatype`.

| Operator | Name | Type |
|---|---|---|
| `q = r` | Equal | `<a> * <a>` $\rightarrow$ bool |
| `q <> r` | Not equal | `<a> * <a>` $\rightarrow$ bool |

## 3.4 Translating compound types

This section lays down the basic requirements for the translation of the $\text{VDM}_{bas}$ compound types into HOL98. For each type we briefly discuss its representation in HOL98. We then list the constants, constructors and operators of the type and give an indication of their translation in HOL. The compound types are composite types, product types, enumeration types, sequence types, set types and mapping types.

### 3.4.1 Composite types

Composite types of VDM-SL are translated using the `Hol_datatype` of HOL98.

$$\langle\!|\texttt{compose } a \texttt{ of } a_1 : \sigma_1 \;\; a_2 : \sigma_2 \texttt{ end}|\!\rangle \;\; = \;\; \texttt{mk\_}a \texttt{ of } \langle\!|\sigma_1|\!\rangle \texttt{ \# } \langle\!|\sigma_2|\!\rangle$$

Consider a declaration of a composite type

$$a \texttt{ = compose } a \texttt{ of } a_1 : \sigma_1 \;\; a_2 : \sigma_2 \texttt{ end}$$

Note that the first $a$ is a type variable while the second $a$ is a tag. The tag is used when constructing values. A value of type $a$ is constructed using `mk_a`$(x_1, x_2)$. Such a constructor is introduced using `Hol_datatype` of HOL98. The following table lists the composite type operators to be supported supported.

| Operator | Name | Type |
|---|---|---|
| `r.i` | Field select | $a * Id \rightarrow \sigma_i$ |
| `r1 = r2` | Equality | $a * a \rightarrow$ bool |
| `r1 <> r2` | Inequality | $a * a \rightarrow$ bool |
| `is_a(r1)` | Is | $Id * a \rightarrow$ bool |

We support the field selector by generating a projection function for each field selector. Thus, the translation of the type declaration above, additionally, yields projection functions

$$\texttt{proj `}a\_a_1\texttt{(mk\_a } (x_1, x_2)) = x_1\texttt{`;}$$

$$\texttt{proj `}a\_a_2\texttt{(mk\_a } (x_1, x_2)) = x_2\texttt{`;}$$

The SML function `proj` is defined using the `Tfl` function `Rfunction`. The reason for choosing the `Tfl` package is that it immediately supports the used pattern matching. Presently, `proj` is defined by

```
fun proj qtm = Rfunction `measure (\x.7)` qtm;
```

The measure is arbitrary and simply reflects the fact that projections are non-recursive, and hence always terminate. As before, we use a special function `proj` for projections because projections play a special role when reasoning about specifications.

The following is a popular shorthand for declaring composite types in VDM-SL:

```
A :: x : X
     y : Y
```

It is a shorthand for

```
A = compose A of x:X y:Y end
```

### 3.4.2 Product types

Product types of VDM-SL are translated to product types of HOL98.

$$\langle\!|\sigma_1 * \sigma_2|\!\rangle \quad = \quad \langle\!|\sigma_1|\!\rangle \ \# \ \langle\!|\sigma_2|\!\rangle$$

Consider the type declaration $\sigma = \sigma_1 * \sigma_2$. A value of type $\sigma$ is constructed using $\texttt{mk\_}(x_1, x_2)$. The only operators working on products are equality and inequality. These are both supported by HOL98.

| Operator | Name | Type |
|---|---|---|
| `t1 = t2` | Equality | $\sigma_1 * \sigma_2 \rightarrow$ bool |
| `t1 <> t2` | Inequality | $\sigma_1 * \sigma_2 \rightarrow$ bool |

General projections $e.\#n$ are not directly supported in HOL98. Instead, they are expanded to sequences of applications of the `FST` and `SND` functions which are available in the pair theory of HOL98.

### 3.4.3 Enumeration types

An enumeration type is a union of quote types. We translate enumeration types to disjoint sums of identifiers:

$$\langle\!|\texttt{<}a_1\texttt{>} \mid \texttt{<}a_2\texttt{>}|\!\rangle \quad = \quad a_1 \ | \ a_2$$

Since HOL98 does not support identifiers enclosed in brackets < and >, we have to get rid of these.

### 3.4.4 Sequence types

We translate VDM-SL sequences to HOL98 lists:

$$\langle\!|\texttt{seq of } \sigma|\!\rangle \quad = \quad \langle\!|\sigma|\!\rangle \ \texttt{list}$$

The sequence constructors supported by VDM-SL are

**Sequence enumeration:** `[e1, e2,..., en]` constructs a sequence of the enumerated elements. The empty sequence will be written as `[]`. Sequence enumeration is supported by the list theory of HOL98.

**Sequence comprehension:** `[e | id in set S & P]` constructs a sequence by evaluating the expression `e` on all the bindings for which the predicate `P` evaluates to true. The expression `e` will use the identifier `id`. `S` is a set of numbers and `id` will be matched to the numbers in the normal order (the smallest number first). Sequence comprehension is not supported by the list theory of HOL98. However, a preliminary example definitions have been developed for use in the examples [5].

The following table lists the sequence operators supported by VDM-SL. Many of the operators are supported by the `list` theory in HOL98.

| Operator | Name | Type |
|---|---|---|
| `hd l` | Head | seq1 of $A \to A$ |
| `tl l` | Tail | seq1 of $A \to$ seq of $A$ |
| `len l` | Length | seq of $A \to$ nat |
| `elems l` | Elements | seq of $A \to$ set of $A$ |
| `inds l` | Indexes | seq of $A \to$ set of nat1 |
| `l1 ^ l2` | Concatenation | (seq of $A$) $*$ (seq of $A$) $\to$ seq of $A$ |
| `conc ll` | Distributed concatenation | seq of seq of $A \to$ seq of $A$ |
| `l ++ m` | Sequence modification | seq of $A *$ map nat to $A \to$ seq of $A$ |
| `l(i)` | Sequence application | seq of $A *$ nat1 $\to A$ |
| `l1 = l2` | Equality | (seq of $A$) $*$ (seq of $A$) $\to$ bool |
| `l1 <> l2` | Inequality | (seq of $A$) $*$ (seq of $A$) $\to$ bool |

### 3.4.5 Set types

We translate VDM-SL sets to predicates in HOL98 using the `pred_set` theory.

$$\langle\!|\texttt{set of } \sigma|\!\rangle \;=\; \langle\!|\sigma|\!\rangle \to \texttt{bool}$$

The set constructors supported by VDM-SL are

**Set enumeration:** `{e1, e2, ..., en}` constructs a set of the enumerated elements. The empty set is denoted by {}. This constructor is supported by the `pred_set` theory in HOL98.

**Set comprehension:** `{e | bd1, bd2, ..., bdm & P}` constructs a set by evaluating the expression `e` on all the bindings for which the predicate `P` evaluates to true. A binding is either a set binding or a type binding. A set binding `bdn` has the form `pat1, ..., patp in set s`, where `pati` is a pattern (normally simply an identifier), and `s` is a set constructed by an expression. A type binding is similar, in the sense that `in set` is replaced by a colon and `s` is replaced with a type expression. The set comprehension constructor is supported by the `pred_set` theory in HOL98.

**Set range expression:** {e1, ..., en} constructs a set of the enumerated integers from e1 to e2 both inclusive for integer expressions e1 and e2. If e2 is smaller than e1 the set range expression denotes the empty set. This constructor is not supported by the pred_set theory in HOL98. Again, a preliminary example definition has been developed for use in the examples [5].

The first two constructions are supported by the pred_set theory whereas the last is not. It is however straightforward to add. The following table lists the set operators supported by VDM-SL. Most of the operators are supported by the pred_set theory.

| Operator | Name | Type |
|---|---|---|
| e in set s1 | Membership | $A$ * set of $A$ → bool |
| e not in set s1 | Not membership | $A$ * set of $A$ → bool |
| s1 union s2 | Union | set of $A$ * set of $A$ → set of $A$ |
| s1 inter s2 | Intersection | set of $A$ * set of $A$ → set of $A$ |
| s1 \ s2 | Difference | set of $A$ * set of $A$ → set of $A$ |
| s1 subset s2 | Subset | set of $A$ * set of $A$ → bool |
| s1 psubset s2 | Proper subset | set of $A$ * set of $A$ → bool |
| s1 = s2 | Equality | set of $A$ * set of $A$ → bool |
| s1 <> s2 | Inequality | set of $A$ * set of $A$ → bool |
| card s1 | Cardinality | set of $A$ → nat |
| dunion ss | Distributed union | set of set of $A$ → set of $A$ |
| dinter ss | Distributed intersection | set of set of $A$ → set of $A$ |
| power s1 | Finite power set | set of $A$ → set of set of $A$ |

### 3.4.6 Mapping types

We use the HOL98 support for finite mappings written by Collins and Syme [13].

$$(\!|\texttt{map}\ \sigma_1\ \texttt{to}\ \sigma_2|\!) \quad = \quad ((|\sigma_1|),(|\sigma_2|))\texttt{fmap}$$

The map constructors supported by VDM-SL are

**Map enumeration:** {a1 |-> b1, a2 |-> b2, ..., an |-> bn} constructs a mapping of the enumerated maplets. The empty map will be written as {|->}. This constructor is not supported by the fmap theory. A preliminary example definition has been developed for use in the examples [5].

**Map comprehension:** {ed |-> er | bd1, ..., bdn & P} builds a mapping by evaluating the expressions ed and er on all the possible bindings for which the predicate P evaluates to true. bd1, ..., bdn are bindings of free identifiers from the expressions ed and er to sets or types. This constructor is not supported by the pred_set theory in HOL98. Again, a preliminary example definition has been developed for use in the examples [5].

The following table lists the mapping operators supported by VDM-SL. The `fmap` theory of Collins and Syme supports most of these.

| Operator | Name | Type |
|---|---|---|
| `dom m` | Domain | (map $A$ to $B$) $\rightarrow$ set of $A$ |
| `rng m` | Range | (map $A$ to $B$) $\rightarrow$ set of $B$ |
| `m1 munion m2` | Merge | (map $A$ to $B$) $*$ (map $A$ to $B$) $\rightarrow$ map $A$ to $B$ |
| `m1 ++ m2` | Override | (map $A$ to $B$) $*$ (map $A$ to $B$) $\rightarrow$ map $A$ to $B$ |
| `merge ms` | Distributed merge | set of (map $A$ to $B$) $\rightarrow$ map $A$ to $B$ |
| `s <: m` | Domain restrict to | (set of $A$) $*$ (map $A$ to $B$) $\rightarrow$ map $A$ to $B$ |
| `s <-: m` | Domain restrict by | (set of $A$) $*$ (map $A$ to $B$) $\rightarrow$ map $A$ to $B$ |
| `m :> s` | Range restrict to | (map $A$ to $B$) $*$ (set of $B$) $\rightarrow$ map $A$ to $B$ |
| `m :-> s` | Range restrict by | (map $A$ to $B$) $*$ (set of $B$) $\rightarrow$ map $A$ to $B$ |
| `m(d)` | Map apply | (map $A$ to $B$) $* A \rightarrow B$ |
| `m1 comp m2` | Map composition | (map $B$ to $C$) $*$ (map $A$ to $B$) $\rightarrow$ map $A$ to $C$ |
| `m ** n` | Map iteration | (map $A$ to $A$) $*$ nat $\rightarrow$ map $A$ to $A$ |
| `m1 = m2` | Equality | (map $A$ to $B$) $*$ (map $A$ to $B$) $\rightarrow$ bool |
| `m1 <> m2` | Inequality | (map $A$ to $B$) $*$ (map $A$ to $B$) $\rightarrow$ bool |

## 3.5 Invariants

In this section, we begin the gradual extension of the language $\text{VDM}_{bas}$ by extending the type system of $\text{VDM}_{bas}$ with type invariants.

Let $\gamma_i$ be a type expression and $d_i$ be an expression then we extend a type declaration to consist of two parts; a type definition and a type invariant

$$t_i = \gamma_i$$
$$\texttt{inv}\ x_i == d_i;$$

The type invariant (or just invariant) is a predicate on type $\gamma_i$. The invariant is optional, but for convenient uniformity we assume that all types have an invariant associated with it. The assumption is made without loss of generality since in case no invariant is present the trivial invariant which is always `true` can be added to achieve a type following the scheme.

In VDM-SL, invariants are an intrinsic part of the type system. In the proposed translation, we explicitly keep track of the invariant associated with a type. Hence in general, a VDM-SL type is not represented directly by a HOL98 type. Instead, we represent VDM-SL types according to the equation

$$\text{VDM-SL type} = \text{HOL98 type} + \text{invariant}$$

The function $(\!|\text{-}|\!)$ for translating types is unaffected by the enrichment with type invariants. In fact, type invariants only need to be made explicit in bindings (and later when recursive functions are considered.)

A type declaration

$$t_i = \gamma_i$$
$$\texttt{inv}\ x_i == d_i;$$

is translated to

$$\texttt{vdm\_datatype`}\, t_i = (\!|\gamma_i|\!)\,\texttt{`};$$

$$\texttt{pred`}\, inv\_t_i\ (x_i : (\!|\gamma_i|\!)) = ([\![d_i]\!] : \texttt{bool}) \texttt{/\textbackslash}\ inv(\gamma_i)\ x_i\,\texttt{`};$$

where vdm_datatype is discussed in Section 3.2 above.

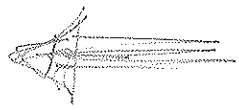
### 3.5.1 Deriving invariants

In this section, we show how to extract, for a given VDM-SL type, the possibly implicitly associated invariant from the explicitly declared invariants. We define a function *inv* that given a VDM-SL type computes the associated invariant in terms of a HOL98 term. The inductive definition of *inv* is

$$\begin{aligned}
inv(\texttt{token}) &= \texttt{\textbackslash x. T}\\
inv(\texttt{bool}) &= \texttt{\textbackslash x. T}\\
inv(\texttt{char}) &= \texttt{\textbackslash x. T}\\
inv(\texttt{nat}) &= \texttt{\textbackslash x. T}\\
inv(\texttt{<}a\texttt{>}) &= \texttt{\textbackslash x. T}\\
inv(t_i) &= \texttt{\textbackslash x. inv_}t_i\,\texttt{x}\\
inv(\texttt{compose}\ A\ \texttt{of}\ a_1 : \sigma_1\ \ a_2 : \sigma_2\ \texttt{end}) &= \texttt{\textbackslash x.}\ inv(\sigma_1)\,(A_a_1\,x)\ \texttt{/\textbackslash}\ inv(\sigma_2)\,(A_a_2\,x)\\
inv(\sigma_1 \texttt{*} \sigma_2) &= \texttt{\textbackslash x.}\ inv(\sigma_1)\,(\texttt{FST}x)\ \texttt{/\textbackslash}\ inv(\sigma_2)\,(\texttt{SND}x)\\
inv(\texttt{<}a_1\texttt{>} \mid \texttt{<}a_2\texttt{>}) &= \texttt{\textbackslash x. T}\\
inv(\texttt{set of}\ \sigma) &= \texttt{\textbackslash as.(!a. a IN as ==>}\ inv(\sigma)\,\texttt{a) /\textbackslash}\\
&\quad \texttt{(FINITE as)}\\
inv(\texttt{seq of}\ \sigma) &= \texttt{\textbackslash as.!a. a IN ELEMS as ==>}\ inv(\sigma)\,\texttt{a}\\
inv(\texttt{map}\ \sigma_1\ \texttt{to}\ \sigma_2) &= \texttt{\textbackslash f.(!a. a IN FDOM f ==>}\ inv(\sigma_1)\,\texttt{a) /\textbackslash}\\
&\quad \texttt{(!a. a IN FDOM f ==>}\ inv(\sigma_2)\,\texttt{(FAPPLY f a))}
\end{aligned}$$

### 3.5.2 Translation with invariants

As mentioned, the type translation is unaffected by the enrichment with type invariants. Instead, we represent type invariants explicitly and therefore the function for translating expressions [-] needs to be altered. It does, however, only need to be altered for expressions using bindings. There are three such cases. For each $\Gamma$ and $e_1$, let $\eta$ denote the type given by the equation $\texttt{set of}\ \eta = \tau_\Gamma(e_1)$.



$$\begin{aligned}
[\![\texttt{forall x:}\sigma\ \texttt{\&}\ e]\!]_\Gamma &= \texttt{!x:}(\!|\sigma|\!)\texttt{.}\,(inv(\sigma)\ \texttt{x}) \texttt{==>}\ [\![e]\!]_{\Gamma[x\leftarrow\sigma]}\\
[\![\texttt{iota x:}\sigma\ \texttt{\&}\ e]\!]_\Gamma &= \texttt{@x:}(\!|\sigma|\!)\texttt{.}\,(inv(\sigma)\ \texttt{x}) \texttt{/\textbackslash}\ [\![e]\!]_{\Gamma[x\leftarrow\sigma]}\\
[\![\texttt{let x:}\sigma\ \texttt{be st}\ e_1\ \texttt{in}\ e_2]\!]_\Gamma &= \texttt{let x = @x:}(\!|\sigma|\!)\texttt{.}\,inv(\sigma)\ \texttt{x} \texttt{/\textbackslash}\ [\![e_1]\!]_{\Gamma[x\leftarrow\sigma]}\\
&\quad \texttt{in}\,[\![e_2]\!]_{\Gamma[x\leftarrow\sigma]}
\end{aligned}$$

In each case, the binding is explicitly strengthened by introducing the derived invariant.

# Chapter 4

# Extending the basic subset

In this chapter, we extend the $\text{VDM}_{bas}$ language with constructs from $\text{VDM}_{dec}$ the larger VDM-SL language. All of these constructs can be transformed to simpler constructs already defined within $\text{VDM}_{bas}$ prior to translation into HOL98. The extensions covered are pattern matching, set bindings, cases expressions and implicit function definitions.

## 4.1 Pattern matching

Pattern matching plays an important role in VDM-SL specifications. It is used frequently to gain access to the values in the fields of a record and it is the only way to gain access to the values of the components of a tuple. In this chapter, we first extend our basic language $\text{VDM}_{bas}$ with pattern matching and then show how constructs with pattern matching can be transformed into constructs without pattern matching, prior to translation into HOL98.

Below we augment our language to support pattern matching in connection with local variables in let expressions, binders in quantifier expressions, and function declarations. The syntax for patterns is given by

$$
\begin{array}{rcl}
p & ::= & x \mid c \mid \text{-} \mid (e) \mid \\
 & & \texttt{mk\_}(p_1, \ldots, p_k) \mid mk\_A(p_1, \ldots, p_k) \mid \\
 & & [p_1, \ldots, p_k] \mid \{p_1, \ldots, p_k\} \mid \\
 & & p_1 \ \texttt{union} \ p_2 \mid p_1 \ \hat{} \ p_2
\end{array}
$$

In this chapter, we do not consider set enumeration, union, and concatenation patterns since they involve looseness.

The so-called "don't care", -, pattern can be replaced by a fresh variable. For convenience, we therefore assume in the following that "don't care" patterns do not appear.

In the following, we need a function *PID* to compute the set of pattern

identifiers of a pattern.

$$\begin{aligned}
PID(x) &= \{x\} \\
PID(c) = PID(\text{-}) = PID((e)) &= \{\} \\
PID(\texttt{mk_}(p_1,\ldots,p_k)) = PID(mk_A(p_1,\ldots,p_k)) &= \bigcup_{i=1,\ldots,k} PID(p_i) \\
PID(\texttt{[}p_1,\ldots,p_k\texttt{]}) = PID(\{p_1,\ldots,p_k\}) &= \bigcup_{i=1,\ldots,k} PID(p_i) \\
PID(p_1 \texttt{ union } p_2) = PID(p_1 \hat{}\, p_2) &= PID(p_1) \cup PID(p_2).
\end{aligned}$$

Also, we need to extend the function $FV$ for computing free variables in expression to also compute free variables in patterns.

$$\begin{aligned}
FV(x) = FV(c) = FV(\text{-}) &= \{\} \\
FV((e)) &= FV(e) \\
FV(\texttt{mk_}(p_1,\ldots,p_k)) = FV(mk_A(p_1,\ldots,p_k)) &= \bigcup_{i=1,\ldots,k} FV(p_i) \\
FV(\texttt{[}p_1,\ldots,p_k\texttt{]}) = FV(\{p_1,\ldots,p_k\}) &= \bigcup_{i=1,\ldots,k} FV(p_i) \\
FV(p_1 \texttt{ union } p_2) = FV(p_1 \hat{}\, p_2) &= FV(p_1) \cup FV(p_2).
\end{aligned}$$

In the following, we shall assume that for each pattern $p$ there is no overlap between the pattern identifiers and the free variables of $p$, that is,

$$PID(p) \cap FV(p) = \emptyset.$$

We have chosen to enforce this restriction for two reasons. First, the restriction makes the elimination of pattern matching is more straightforward. Second, the restriction seems to have no practical influence since occurrences of such overlaps seem uncommon in practice.

### 4.1.1 Let expressions

In let expressions, pattern matching can be used in the binding of local variables. The syntax for let expressions with pattern matching is

$$\begin{aligned}
e \;::=\; & \texttt{let } p \texttt{ = } e_1 \texttt{ in } e_2 \mid \\
& \texttt{let } p\texttt{:}\sigma \texttt{ = } e_1 \texttt{ in } e_2
\end{aligned}$$

There are two kinds of patterns in the syntax. The first is without type annotation. The second is with type annotation. Below, we show how to eliminate type annotated patterns. For patterns without type annotations the transformations are the same except that the type annotations are removed in the resulting expression.

We now proceed to show how to transform let expressions into let expressions with only variable patterns, and hence just ordinary variable bindings. The

transformation is inductive and built on a case analysis of the shape of the pattern $p : \sigma$ in the let expression. There are three case to consider. Below, we go through each case and give transformation from which it is straightforward to produce the full transformation by an inductive argument.

**Simple patterns** Variable, constant, and expression patterns are straightforward to handle. For variable patterns $x{:}\sigma$ no transformation is needed. Constant and expression patterns do not bind any variables. Hence, such bindings can be removed, that is, expressions of the form

$$\texttt{let } c : \sigma \texttt{ = } e_1 \texttt{ in } e_2$$

and

$$\texttt{let } (e) : \sigma \texttt{ = } e_1 \texttt{ in } e_2$$

are both simply replaced by the expression

$$e_2$$

**Product and composite patterns** Next, we show how to eliminate composite patterns in let expressions. A product type is just an anonymous composite type. As explained in Section 3.4.2, the translation generates projections for product types. Therefore, product and composite patterns can be handled in essentially the same way. Assume that the composite type $A$ is defined the declaration

$$A \texttt{ = compose } A \texttt{ of } a_1 : \sigma_1 \ldots a_k : \sigma_k \texttt{ end}$$

followed by some invariant. Then, the expression

$$\texttt{let mk\_}A(p_1, \ldots, p_k) : \sigma \texttt{ = } e_1 \texttt{ in } e_2$$

is replaced by the expression

$$\begin{array}{lll} \texttt{let} & x : \sigma & = \quad e_1, \\ & p_1 & = \quad x.a_1, \\ & \vdots & \\ & p_k & = \quad x.a_k \\ \texttt{in} & & \\ & e_2 & \end{array}$$

where $x$ is a fresh variable, that is, $x \notin PID(p) \cup FV(p) \cup FV(e)$.

**Sequence enumeration patterns** The transformation for sequence enumeration patterns is simlar to that for transforming product and composite patterns. The expression

$$\texttt{let } [p_1, \ldots, p_k] : \sigma \texttt{ = } e_1 \texttt{ in } e_2$$

is replaced by the expression

$$\begin{array}{lll} \texttt{let} & x:\sigma & = & e_1, \\ & p_1 & = & x(\texttt{1}), \\ & \vdots & & \\ & p_k & = & x(\texttt{k}) \\ \texttt{in} & & & \\ & e_2 & & \end{array}$$

where $x$ is a fresh variable, that is, $x \notin PID(p) \cup FV(p) \cup FV(e)$.

### 4.1.2 Let be such expressions

As for let expressions and explicit function declarations, pattern matching can be used in let-be-st expressions. In this section, we show how also these patterns can be transformed into simpler $\text{VDM}_{bas}$ expressions.

In let-be-st expressions, pattern matching can be used in the binding of local variables. The syntax for let-be-st expressions with pattern matching is

$$e \ ::= \ \texttt{let}\ p:\sigma\ \texttt{be st}\ e_1\ \texttt{in}\ e_2$$

We now proceed to show how to transform let-be-st expressions into let-be-st expressions with only variable patterns, and hence just ordinary variable bindings.

The expression

$$\texttt{let}\ p:\sigma\ \texttt{be st}\ e_1\ \texttt{in}\ e_2$$

is replaced by the expression

$$\begin{array}{ll} \texttt{let} & x:\sigma\ \texttt{be st} \\ & \quad \texttt{let x = p in}\ e_1 \\ \texttt{in} & \\ & \quad \texttt{let x = p in}\ e_2 \end{array}$$

where $x$ is a fresh variable, that is, $x \notin PID(p) \cup FV(p) \cup FV(e_1) \cup FV(e_2)$.

### 4.1.3 Quantifier expressions

Pattern matching can be used in the bindings of quantifier expressions. The syntax for binders $b$ augmented with patterns $p$ is given by

$$b \ ::= \ p:\sigma$$

Let $p$ be a pattern, let $\{x_1, \ldots, x_n\}$ be the set of pattern identifiers $PID(p)$ of $p$, and for each $i = 1, \ldots n$, let $\sigma_i$ be the type of the variable $x_i$.

The expression

$$\texttt{forall}\ p:\sigma\ \texttt{\&}\ e$$

is replaced by the expression

$$\texttt{forall}\ x_1:\sigma_1, \ldots, x_n:\sigma_n, x:\sigma\ \texttt{\&}\ x = p \texttt{=>}\ e$$

where $x$ is a fresh variable, that is, $x \notin PID(p) \cup FV(p) \cup FV(e)$.

### 4.1.4 Iota expressions

As for quantifier expressions, pattern matching can be used in the bindings of iota expressions. The syntax for binders $b$ augmented with patterns $p$ is the same as for quantifier expressions. Let $p$ be a pattern, let $\{x_1, \ldots, x_n\}$ be the set of pattern identifiers $PID(p)$ of $p$, and for each $i = 1, \ldots n$, let $\sigma_i$ be the type of the variable $x_i$.

The expression

$$\texttt{iota}\ p : \sigma\ \texttt{\&}\ e$$

is replaced by the expression

$$\texttt{iota}\ x : \sigma\ \texttt{\&}\ \texttt{exists}\ x_1 : \sigma_1, \ldots, x_n : \sigma_n\ \texttt{\&}\ x\ \texttt{=}\ p\ \texttt{and}\ e$$

where $x$ is a fresh variable, that is, $x \notin PID(p) \cup FV(p) \cup FV(e)$.

### 4.1.5 Function and type declarations

Patterns can be used as formal parameters in function declarations and type declarations. In this case, the patterns can be moved into patterns in let expressions, by introducing new variable names and extending the body with a let expression matching the new variables against the corresponding patterns. Thus, the function declaration

$$\begin{array}{l} f(p_1, \ldots, p_n) == e \\ \texttt{pre}\ e' \\ \texttt{post}\ e'' \end{array}$$

is replaced by the declaration

$$\begin{array}{l} f(x_1, \ldots, x_n) == \texttt{let}\ p_1\ \texttt{=}\ x_1, \ldots, p_n\ \texttt{=}\ x_n\ \texttt{in}\ e \\ \texttt{pre}\ \texttt{let}\ p_1\ \texttt{=}\ x_1, \ldots, p_n\ \texttt{=}\ x_n\ \texttt{in}\ e' \\ \texttt{post}\,\texttt{let}\ p_1\ \texttt{=}\ x_1, \ldots, p_n\ \texttt{=}\ x_n\ \texttt{in}\ e'' \end{array}$$

where $x_1, \ldots, x_n$ are fresh variables, that is, $x_1, \ldots, x_n \notin PID(p) \cup FV(p) \cup FV(e)$.

The same principle applies for invariants. Hence, the type declaration

$$\begin{array}{l} t = \gamma \\ \texttt{inv}\, p == d \end{array}$$

is replaced by the declaration

$$\begin{array}{l} t = \gamma \\ \texttt{inv}\, x == \texttt{let}\ p\ \texttt{=}\ x\ \texttt{in}\ d \end{array}$$

where $x$ is a fresh variable, that is, $x \notin PID(p) \cup FV(p) \cup FV(d)$.

## 4.2 Set bindings

Set bindings play an important role in defining an expressive executable subset of VDM-SL, cf. [17]. Due to the finiteness of sets in VDM-SL, set bindings can be straightforwardly enumerated whereas type bindings in general range over infinitely many values.

In this chapter we augment our language $\text{VDM}_{bas}$ with set bindings and show how they can be transformed out into other more basic $\text{VDM}_{bas}$ constructs. In this chapter, we first extend our basic language $\text{VDM}_{bas}$ with pattern matching and then show how constructs with pattern matching can be transformed into constructs without pattern matching, prior to translation into HOL98.

Set bindings occur in quantification, iota, and let-be-such expressions. They take the syntax:

$$\begin{aligned} e \ ::= \ & \texttt{forall}\ p\ \texttt{in set}\ e_1\ \texttt{\&}\ e_2 \mid \\ & \texttt{iota}\ p\ \texttt{in set}\ e_1\ \texttt{\&}\ e_2 \mid \\ & \texttt{let}\ x\ \texttt{in set}\ e_1\ \texttt{be st}\ e_2\ \texttt{in}\ e_3. \end{aligned}$$

Following we show how to interpret each of these new constructions as syntactic sugar for existing constructions. In the following, let $e_1$, $e_2$, and $e_3$ range over expressions, $p$ over patterns, and $\sigma$ over type expressions of our augmented language. Also, let $\sigma$ be a type expression denoting the type of the expression $e_1$.

The expression

$$\texttt{forall}\ p\ \texttt{in set}\ e_1\ \texttt{\&}\ e_2$$

is replaced by the expression

$$\texttt{forall}\ p{:}\sigma\ \texttt{\&}\ p\ \texttt{in set}\ e_1\ \texttt{=>}\ e_2.$$

The expression

$$\texttt{iota}\ p\ \texttt{in set}\ e_1\ \texttt{\&}\ e_2$$

is replaced by the expression

$$\texttt{iota}\ p{:}\sigma\ \texttt{\&}\ p\ \texttt{in set}\ e_1\ \texttt{and}\ e_2.$$

The expression

$$\texttt{let}\ p\ \texttt{in set}\ e_1\ \texttt{be st}\ e_2\ \texttt{in}\ e_3$$

is replaced by the expression

$$\texttt{let}\ p{:}\sigma\ \texttt{be st}\ p\ \texttt{in set}\ e_1\ \texttt{and}\ e_2\ \texttt{in}\ e_3.$$

## 4.3 Cases expressions

As for many languages, cases expressions are ubiquitous in VDM-SL specifications. In VDM-SL, cases expressions take the syntax:

$$e \ ::= \ \texttt{cases}\ e\texttt{:}\ \ p_1\ \texttt{->}\ e_1, \ldots, p_n\ \texttt{->}\ e_n, \texttt{others}\ \texttt{->}\ e_{n+1}\ \texttt{end}$$

Cases expressions can be straightforwardly translated into nested if expressions as for example done in [3].

```
if exists p1:σ1 & p1 = e then
     let p1 = e in e1
else
⋮
if exists pn:σn & pn = e then
     let pn = e in en
else
     en+1
```

We suspect however that it will be valuable to use special case splitting tactics when reasoning about cases expressions, and that this is bound to be awkward once the cases expressions are replaced by nested if expressions. We intend to let experience decide the matter through examples and case studies.

## 4.4 Implicit function definitions

Implicit function definitions, which are specified using pre- and postconditions only and have no function body, can be replaced with explicit function definitions using the let-be-st expression. Hence, a function definition

```
f(p1 : σ1, ..., pm : σm) y1 : η1 ... yn : ηn
pre e'
post e'';
```

is replaced by the definition

```
f : σ1 * ... * σm -> η1 * ... * ηn
f(p1, ..., pm) == let mk_(y1, ..., yn) : η1 * ... * ηn
                  be st post_f(p1, ..., pm, y1, ..., yn)
                  in mk_(y1, ..., yn)
pre e'
post let RESULT = mk_(y1, ..., yn) in e'';
```

# Chapter 5

# Ongoing technical issues

In this chapter, we discuss three areas of translation in which there remain technical issues to be resolved: the translation of recursive function definitions, general union types and recursive types.

## 5.1 Recursive functions

We have not yet considered the translation of mutually recursive functions because they are not immediately supported by the `Tfl` package which we have used throughout, because we have not had the time to try out the `mutrec` package, and finally, because we hope for a more tight integration of the packages `phb`, `Tfl`, and `mutrec` in the future.

A function declaration

$$
\begin{array}{l}
f : \sigma_1 * \ldots * \sigma_n \texttt{ -> } \sigma \\
f(x_1, \ldots, x_n) == e \\
\texttt{pre } e' \\
\texttt{post } e'';
\end{array}
$$

where $f$ appears in the expression $e$ is translated to

$$\texttt{pred `}\mathit{pre\_f}(x_1 : \langle\!|\sigma_1|\!\rangle, \ldots, x_n : \langle\!|\sigma_n|\!\rangle) = (\llbracket e' \rrbracket : \texttt{bool})\texttt{`;}$$

$$\texttt{pred `}\mathit{post\_f_i}(x_1 : \langle\!|\sigma_1|\!\rangle, \ldots, x_n : \langle\!|\sigma_n|\!\rangle, \texttt{RESULT} : \langle\!|\sigma|\!\rangle) = (\llbracket e'' \rrbracket : \texttt{bool})\texttt{`;}$$

$$
\begin{array}{l}
\texttt{rfunc `}f(x_1 : \langle\!|\sigma_1|\!\rangle, \ldots, x_n : \langle\!|\sigma_n|\!\rangle) = \\
\qquad \texttt{(if } \mathit{inv}(\sigma_1) \texttt{ and } \ldots \texttt{ and } \mathit{inv}(\sigma_n) \texttt{ and pre\_f}(x_1, \ldots, x_n) \\
\qquad \texttt{then } \llbracket e \rrbracket \\
\qquad \texttt{else @r:}(\langle\!|\sigma|\!\rangle\texttt{.T}) : \langle\!|\sigma|\!\rangle\texttt{)`} \\
\qquad \texttt{`measure } \backslash(x_1, \ldots, x_n).d\texttt{`;}
\end{array}
$$

where $d$ is HOL98 expression of type num. Presently, $d$ must be supplied manually. Consider the function

```
Gateway: seq of Message * Category -> Ports
Gateway(ms,cat) ==
  if ms = []
  then mk_Ports([],[])
  else let rest_p = Gateway(tl ms,cat)
       in
         ProcessMessage(hd ms,cat,rest_p);
```

from the gateway example in [7]. In the manual translation presented in [4], we translate the function as follows

```
rfunc `Gateway(ms,cat) =
    COND (ms = []) (mk_Ports([],[]))
                   (let rest_p = Gateway(TL ms,cat)
                    in
                      ProcessMessage(HD ms,cat,rest_p))`
     `measure \(x,y). LENGTH x`;
```

Note that we have not added the extra guarding imposed by the type invariant on the parameter type since we can show termination without these. Also, note that the function `function` of the `Tfl` package can in fact derive good termination conditions for such functions when no measure is supplied.

Presently, `rfunc` is a somewhat ad hoc extension of the `Rfunction` of `Tfl`. The implementation reflects that we have spend little time on the proving of termination.

```
local
  val VDM_TAIL_LENGTH = mk_thm([],Term
    `!l. ~(l = []) ==> ((LENGTH (TL l)) < (LENGTH l))`)
  val vdm_termination_ss =
    mk_simpset [HOL_ss,rewrites[VDM_TAIL_LENGTH]]
in
fun rfunc dqtm mqtm =
     let val out = Rfunction mqtm dqtm in
          if #tcs(out) <> [] then
             let val _ = say "Oops! let's try proving tcs ourselves\n"
                 val termination =
                    prove_termination out
                       (SIMP_DEC_TAC vdm_termination_ss)
                 val rules = RW_RULE [termination] (#rules out)
                 val induction = PROVE_HYP termination (#induction out)
             in
                  {rules = rules,induction = induction,tcs = []}
             end  handle _   =>
                  let
                       val _ = say "Oops! couldn't prove tcs\n"
                  in out
```

```
                end
          else
        out
    end
end;
```

## 5.2 Union types

In VDM-SL, the union type corresponds to set union. Thus, the union type is a non-disjoint union. Such general union types are not supported by the higher order logic HOL. But, HOL does support disjoint sums (unions).

No decision has been made so far. But it seems reasonable to first restrict the translation to disjoint sums, and then see what happens in practice. Disjoint sums can be translated almost directly into HOL98 using the `Hol_datatype`.

Later, if the restriction to disjoint sums is too severe, we will consider more advanced translation using explicit tagging of the components of non-disjoint unions.

## 5.3 Recursive types

The use of recursive types is usually entangled with the use of union types.

In a first translation, we will support recursive types to the extent they are supported by the `Hol_datatype`, that is, recursive types (in connection with disjoint sums) which are neither mutually recursive nor nested.

Alternatively, we will investigate the support for mutually recursive types given by the `mutrec` theory. But, we hope for a more tight integration of the packages `phb`, `Tfl`, and `mutrec` in the future.

# Chapter 6

# Conclusions

Despite the incompatibilities between VDM-SL and HOL, we have been able to identify a large and useful subset of VDM-SL which can be formalized in HOL. We have described the requirements for a translation algorithm which maps a specification in this subset to a sequence of HOL98 commands formalizing the specification in the HOL98 system. Our next task is to write a precise specification of the algorithm in VDM-SL and then execute and test this specification using the VDM tools of IFAD. Finally it will be implemented in C++ or Java.

In the translation algorithm we have made a number of assumptions and choices. Some of the most important ones are:

- Proof obligations ensure that specifications are two-valued.
- Termination of recursive type and function definitions is handled using HOL definition mechanisms rather than proof obligations in VDM-SL.
- Union types are restricted to non-disjoint union types.
- Looseness is formalized using Hilbert's choice operator.
- Invariants and their types are translated separately to HOL, because HOL does not support subtypes.

The `Tfl` and `phb` packages of HOL98 have been used extensively. For example, `Tfl` are used to introduce recursive functions so it would be useful with more powerful support for these, e.g. mutually recursive functions are not yet supported. It would also be useful to have some support for type abbreviations, which are currently made using type antiquotation and by expanding types. We would also like some support for cases expressions which are used extensively in VDM-SL, and standard if-then-else expressions would also be useful. Finally, support for record types à la VDM-SL would be useful.

# Appendix A

# Characters

The following table lists the values of the character type `char`.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | d | e | f | g | h | i | j | k | l | m |
| n | o | p | q | r | s | t | u | v | w | x | y | z |
| A | B | C | D | E | F | G | H | I | J | K | L | M |
| N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| , | : | ; | = | ( | ) | \| | ~ | [ | ] | | | |
| { | } | + | / | < | | | | | | | | |
| > | . | & | ^ | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | | |
| _ | ' | ` | " | @ | ~ | | | | | | | |

Table A.1: The values of the type `char`

# Bibliography


[1] S. Agerholm. Translating specifications in VDM-SL to PVS. In J. von Wright, J. Grundy, and J. Harrison, editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'96)*, LNCS 1125. Springer-Verlag, 1996.

[2] S. Agerholm and J. Frost. An Isabelle-based theorem prover for VDM-SL. In *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, LNCS 1275. Springer-Verlag, August 1997.

[3] S. Agerholm and J. Frost. Towards an integrated CASE and theorem proving tool for VDM-SL. In *FME'97*, LNCS 1313. Springer-Verlag, September 1997.

[4] S. Agerholm and K. Sunesen. HOL formalization of VDM-SL examples and proof obligations. Technical Report IFAD-PROSPER-DOC-4, IFAD, Forskerparken 10, DK-5230 Odense M., Denmark, 1998.

[5] S. Agerholm and K. Sunesen. Hol theory files for VDM-SL examples. Technical Report IFAD-PROSPER-DOC-6, IFAD, Forskerparken 10, DK-5230 Odense M., Denmark, 1998.

[6] S. Agerholm and K. Sunesen. A two-valued subset of VDM-SL. Technical Report IFAD-PROSPER-DOC-1, IFAD, Forskerparken 10, DK-5230 Odense M., Denmark, 1998.

[7] S. Agerholm and K. Sunesen. VDM-SL examples and proof obligations. Technical Report IFAD-PROSPER-DOC-3, IFAD, Forskerparken 10, DK-5230 Odense M., Denmark, 1998.

[8] S. Agerholm and K. Sunesen. VDM-SL Toolbox Extensions. Technical Report IFAD-PROSPER-DOC-7, IFAD, Forskerparken 10, DK-5230 Odense M., Denmark, 1999.

[9] H. Barringer, J. H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[10] J. C. Bicarregui, J. S. Fitzgerald, P. A. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: A Practitioner's Guide.* FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X.

[11] R. J. Boulton, A. D. Gordon, M. J. C. Gordon, J. R. Harrison, J. M. J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Theorem Provers in Circuit Design: Theory, Practice and Experience: Proceedings of the IFIP TC10/WG 10.2 International Conference*, IFIP Transactions A-10, pages 129–156. North-Holland, June 1992.

[12] J. H. Cheng. A logic for partial functions. Ph.D. Thesis UMCS-86-7-1, Department of Computer Science, University of Manchester, Manchester M13 9PL, England, 1986.

[13] G. Collins and D. Syme. A theory of finite maps. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and its Applications.* Springer-Verlag, September 1995. LNCS 971.

[14] J. Fitzgerald and P. G. Larsen. *Modelling Systems: Practical Tools and Techniques in Software Development.* Cambridge University Press, 1998.

[15] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem-proving Environment for Higher-Order Logic.* Cambridge University Press, 1993.

[16] P. G. Larsen. *Towards Proof Rules for VDM-SL.* PhD thesis, Technical University of Denmark, Department of Computer Science, March 1995. ID-TR:1995-160.

[17] 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, March 1991.