Tradução de Tipos de Especificação para Protótipo [1]

Paulo H. M. Borba
Silvio R. de L. Meira

Departamento de Informática
Universidade Federal de Pernambuco
Caixa Postal 7851, 50739, Recife, PE, Brasil

Resumo

Introduzimos um método de transformação de especificações formais em protótipos funcionais. Em seguida, descrevemos a tradução de tipos da linguagem de especificação de VDM para a linguagem de programação SML, terminando com a especificação e protótipo da parte de tradução de tipos de um sistema de transformação de especificações em VDM para protótipos em SML.

Abstract

First, we introduce a method for transforming formal specifications into functional prototypes. Then, we describe the type translation from the VDM specification language to the SML programming language. We conclude by giving the specification and prototype of the type translation subsystem of a VDM to SML transformation system.

1 Introdução

Métodos Formais estão sendo cada vez mais usados para desenvolvimento de Software e Hardware [Bar89]. A utilização destes métodos é essencial para a produção de sistemas cujo funcionamento incorreto implica em riscos à vida e à propriedade. Além disso, existem evidências de que estes métodos diminuem o custo de produção dos sistemas e aumentam a qualidade dos mesmos. Isto ocorre pois os erros e ambigüidades são descobertos mais cedo no processo de desenvolvimento. Os erros são encontrados ainda na especificação, assim não é preciso refazer partes do sistema devido a erros encontrados durante a fase de testes.

[1] Trabalho parcialmente financiado pelo CNPq e SID Informática, Projeto ESTRA.

Também nos métodos formais, para assegurar que o sistema especificado e a ser desenvolvido é o desejado pelo cliente, torna-se necessária a validação da especificação. Tradicionalmente, uma das técnicas mais eficientes de validação é a prototipagem - protótipos do sistema são gerados e apresentados ao cliente que verifica se suas expectativas (com relação aos aspectos funcionais) são satisfeitas. Sem um protótipo, a validação é praticamente inviável, pois o cliente geralmente não possui conhecimentos necessários para entender a especificação formal.

Além de auxiliar a validação da especificação, o protótipo serve como meio de comunicação com o cliente, permitindo que este observe o progresso do projeto. O protótipo também aumenta a confiança que o programador tem do sistema, inclusive proporcionando a observação de como alternativas de projeto funcionam operacionalmente, descobrindo erros de projeto e especificação.

Outra vantagem da utilização de especificações formais é tornar possível a derivação formal, semi-automática e a baixo custo de protótipos a partir das mesmas [BM89b] [JS89]. Neste trabalho, inicialmente introduzimos um método de transformação de especificações em protótipos. Em seguida, descrevemos a tradução de tipos da linguagem de especificação de VDM [BSI89] para a linguagem de programação SML [HMM86], terminando com a especificação (em VDM) e protótipo (em SML) da parte de tradução de tipos do sistema de transformação de especificações em VDM para protótipos em SML, sendo desenvolvido no DI da UFPE. O protótipo foi obtido utilizando-se o método de transformação e será utilizado para a tradução de tipos do resto do sistema.

2 VDM e Protótipos

VDM é um método formal para desenvolvimento de software. Em VDM, um sistema é descrito (especificado) em uma linguagem matemática formal baseada em Teoria dos Conjuntos e Lógica de Predicados, gerando assim um documento preciso que descreve a funcionalidade do mesmo. A descrição é um modelo explícito do sistema. Este modelo é formado pelo estado (objetos) e por um conjunto de operações que agem sobre esse estado. As operações são especificadas através de pré e pós-condições, que estabelecem, respectivamente, as condições necessárias para que a operação seja realizada e as propriedades do estado e do resultado da operação após sua execução.

O método de desenvolvimento de VDM é baseado em um processo de refinamento passo-a-passo (*stepwise refinement* [Jon86]). Cada passo do processo consiste em:

1. Refinar a especificação. Isto é, a partir de uma especificação abstrata do sistema, obter outra mais concreta, através da inclusão de detalhes e decisões de projeto dirigidos à implementação.

2. Garantir a corretude do refinamento, verificando se a especificação obtida é correta em relação à anterior. Esta verificação é feita pelo descarte de um conjunto de obrigações de prova (*proof obligations* [Jon86]) estabelecidas pelo método. O grau de formalismo utilizado nas provas depende da natureza e realidade (recursos) do projeto sendo desenvolvido.

Este passo é repetido até que a especificação obtida esteja detalhada o suficiente para ser diretamente implementada em uma linguagem de programação. O número de passos do processo depende de uma série de fatores: a natureza do sistema, a eficiência desejada da implementação, o nível de abstração da especificação inicial e da quantidade de detalhes introduzida a cada passo.

Antes do início do processo de refinamento, é necessário validar a especificação para que o programa final obtido seja o desejado pelo cliente. Para derivar um protótipo da especificação utiliza-se três abordagens: Transliteração, Transformação e Refinamento.

Transliteração consiste na tradução dirigida por sintaxe de uma especificação em VDM para um programa em SML. É utilizada para especificações explícitas – descreve-se não só *o que* o sistema faz, mas também *como* faz. O subconjunto de VDM que é diretamente executável em SML (passível de transliteração) é apresentado em [BM89a], onde as operações em VDM são representadas por funções em SML, e o estado é passado como argumento e retornado como resultado das operações que o alteram. A tradução de tipos é apresentada completamente na seção 4 deste artigo. A transliteração é realizada automaticamente pelo sistema de transformação sendo desenvolvido.

Transformação resume-se na utilização sistemática de regras de transformação (que preservam significado). Estas regras são aplicadas a uma especificação para obter outra mais concreta. Para derivação de protótipos esta técnica é utilizada para especificações implícitas, e só aplicamos regras de transformação de operações, os dados são implementados diretamente. Um exemplo de como obter um protótipo utilizando-se regras de transformação é apresentado em [BM89b]. O sistema de transformação oferece um conjunto base de regras que pode ser estendido. O usuário escolhe as regras, e o sistema realiza as transformações. Quando a especificação estiver na forma explícita, é efetuada a transliteração.

Refinamento é o método de desenvolvimento utilizado por VDM e foi descrito no início da seção. Quando não for possível utilizar as técnicas anteriores, devido ao alto nível de abstração da especificação, faremos um refinamento superficial apenas das operações, pois os dados são diretamente implementados em SML, e não desejamos que as operações sejam eficientes, apenas queremos que sejam executáveis. O processo de refinamento é manual e pode ser auxiliado por geradores de obrigações e assistentes de provas.

3 Standard ML

Standard ML é uma linguagem fortemente tipada. Cada expressão na linguagem tem um tipo associado, que é inferido em tempo de compilação, evitando-se, assim, erros de tipo durante a execução do programa. Os tipos básicos da linguagem são: inteiros (`int`), reais (`real`), cadeias de caracter (`string`) e booleanos (`bool`). Existe um conjunto de operadores primitivos para os elementos destes tipos. Como exemplo: `+`, `=`, `not`, etc.

Tipos compostos podem ser definidos através do construtor de tuplas. A expressão de tipo:

```
(bit * bitlist)
```

representa o tipo cujos elementos são compostos por um elemento de `bit` e outro de `bitlist`.

Novos tipos podem ser definidos estabelecendo como os seus elementos são formados a partir de elementos de tipos existentes. O tipo lista de dígitos binários pode ser definido por:

```
datatype bitlist  = Nil |
                    Cons of (bit * bitlist)
     and bit = Zero | Um;
```

As declarações definem o tipo `bit`, cujos elementos são os construtores (constroem elementos do tipo): `Zero` e `Um`; e o tipo `bitlist`, que tem como elementos `Nil` (a lista vazia) e os pares formados por elementos de `bit` (cabeça) e de `bitlist` (cauda), prefixados pelo construtor `Cons`. Numa declaração `datatype`, as definições são separadas pela palavra-chave `and` e não precisam estar ordenadas (pode-se fazer referência a um tipo antes deste ser definido), além de poderem ser recursivas e mutuamente recursivas. Um exemplo de um elemento de `bitlist` é:

```
Cons (Zero,Cons (Um,Nil))
```

que representa a lista com cabeça `Zero` e cauda `Cons (Um,Nil)`.

Novos nomes podem ser dados a tipos já definidos:

```
type bit1 = bit
 and bit2 = bit1;
```

Esta definição dá nomes `bit1` e `bit2` aos tipos `bit` e `bit1`. Como se trata apenas da nomeação de tipos (declaração `type`), não faz sentido recursão. As definições precisam estar em ordem (só pode-se fazer referência a `bit1` após sua definição).

Constantes podem ser introduzidas da seguinte maneira:

```
val x = Zero;
```

definindo a constante `x` como `Zero`. Funções são definidas através de um grupo de equações. Cada equação determina o valor da função para um determinado subconjunto de valores do seu domínio. Este subconjunto é estabelecido por um padrão – uma expressão formada apenas por construtores e variáveis. Um padrão serve ao mesmo tempo para seleção da equação da definição a ser utilizada e ligação de variáveis a valores. A função que dá o número de elementos de uma lista de `bits` pode ser definida como:

```
fun length Nil = 0
  | length (Cons (h,t)) = 1 + length t;
```

O número de elementos da lista vazia (`Nil`) é zero. O número de elementos de uma lista com cabeça `h` e cauda `t` é igual a 1 mais o número de elementos da cauda.

Além destas características, SML é uma linguagem polimórfica (definições podem ser genéricas), de alta ordem (funções são valores) e tem primitivas imperativas (como atribuição). Estas características não são utilizadas aqui. SML também tem um poderoso sistema de módulos, similar ao existente para VDM [Mid89], que será utilizado para tradução de especificações modulares.

4 Tradução de Tipos de VDM para SML

Nesta seção apresentamos a linguagem de definição de tipos de VDM. Ao mesmo tempo, descrevemos informalmente como os tipos de VDM são traduzidos para os equivalentes em SML, utilizando como exemplo os tipos usados na especificação apresentada na seção seguinte.

Os objetos de um sistema especificado em VDM são modelados por elementos dos tipos disponíveis na linguagem. VDM dispõe de uma série de tipos primitivos, entre eles: naturais (N), reais (R) e booleanos (B). A partir destes pode-se definir novos tipos através dos construtores de tipo: set of, seq of, map to, que definem tipos cujos elementos são, respectivamente, conjuntos, seqüências e mapeamentos. Com as definições (declarações) de tipo:

TypeDecs = set of *TypeDec*

SMLTypeDecs = seq of *TypeDec*

os elementos do tipo *TypeDecs* (declarações de tipo em VDM) são conjuntos de elementos de *TypeDec* (declaração de tipo), e os do tipo *SMLTypeDecs* (declarações type em SML) são seqüências de *TypeDec*. Em VDM pode-se fazer referência a um tipo mesmo antes deste ser definido (na seqüência de definições da especificação), o que não ocorre em SML (nas definições type). Daí a diferença entre a representação para declarações de tipo em VDM (conjunto, não importa a ordem dos elementos) e SML (seqüência, a ordem dos elementos é considerada).

Cada construtor tem associado uma coleção de operadores sobre os elementos do tipo. Como exemplo, para conjuntos temos: união (∪), cardinalidade (card); para seqüências: primeiro-elemento (hd), tamanho (len); para mapeamentos: domínio (dom), imagem (rng).

Os tipos primitivos de VDM são traduzidos para os correspondentes em SML. Os construtores de tipo e operações associadas são traduzidos para os correspondentes em SML implementados em [BM89a]. Assim, a declaração acima é representada em SML por:

```
type TypeDecs = TypeDec set
 and SMLTypeDecs = TypeDec seq;
```

onde set e seq são os equivalentes de SML a set of e seq of.

Tipos compostos, similares aos *records* de Pascal, também podem ser definidos, listando os componentes de cada elemento do tipo:

TypeDec :: *tpname* : *TypeName*
type : *Type*

Com esta definição, cada elemento de *TypeDec* (declaração de tipo) é formado por um elemento de *TypeName* (nome de tipo) e um elemento de *Type* (expressão que define um tipo). A cada objeto composto está associado um conjunto de seletores dos seus componentes. Os seletores são identificados pelos nomes dos campos do objeto; neste

exemplo os seletores são: *tpname* e *type*. Considerando A um elemento de *TypeName* e B pertencente a *Type*, o elemento de *TypeDec* composto por A e B é representado por:

mk-TypeDec(A, B)

e o nome do tipo associado a esta declaração pode ser obtido utilizando-se o seletor *tpname*, assim temos:

tpname(*mk-TypeDec*$(A, B)) = A$

o mesmo pode ser feito para o outro componente do objeto.

Como mostrado, cada elemento de um tipo composto é prefixado pelo nome do tipo – o nome do tipo faz parte da expressão que define o tipo. Desta forma, definindo:

DecType :: *TypeName*

os elementos de *DecType* são os elementos de *TypeName* prefixados pelo indicador (*tag*) *DecType*.

Como o nome faz parte do tipo, os objetos compostos são traduzidos para tuplas em SML precedidas pelo nome do tipo:

```
datatype TypeDec = TypeDec of (TypeName * Type)
     and DecType = DecType of TypeName;
```

e pode-se definir os seletores como funções:

```
fun tpname (TypeDec (tpn,tp)) = tpn;
```

Novos tipos podem ser definidos pela união de outros tipos da seguinte maneira:

Type = *BasicType* | *DecType*

definindo o tipo *Type* (expressão de tipo) formado pelos elementos de *BasicType* (tipo básico) e de *DecType* (tipo declarado). Uma forma alternativa de união de tipos é:

BasicType = NAT | REAL | BOOL

onde os elementos do tipo *BasicType* (tipo básico) são as constantes: NAT, REAL, BOOL.

A tradução da união de tipos é dada pelo similar `datatype` em SML. Assim as definições acima correspondem a:

```
datatype BasicType = Nat | Real | Bool
     and Type = BasTp of BasicType | DecTp of DecType;
```

Além destes tipos, é permitido definir um tipo sem dar uma representação construtiva para o mesmo (*Given Set*). Tal representação pode não ser necessária num determinado estágio da especificação e será dada numa versão mais concreta da mesma. Assim,

TypeName, is not yet defined

define o tipo *TypeName* (nome de tipo) mas não dá uma representação para o mesmo. Especifica-se que A é um elemento de *TypeName* da seguinte forma:

$A \in$ *TypeName*

A implementação de um *Given Set* é dada através de um tipo básico de SML. Utilizando `string` para implementar *TypeName*, temos:

```
type TypeName = string;
```

e o elemento pertencente a *TypeName* é dado por qualquer `string`:

```
val A = "A";
```

Os tipos disponíveis em VDM são também utilizados pela maioria das linguagens de especificação baseadas em modelos. Assim, o problema que resolvemos - tradução de tipos - pode ser utilizado também para outras linguagens de especificação como Z [Hay87] e RSL, a linguagem de especificação do RAISE [NHWG89].

5 Especificação do Sistema de Transformação de Tipos

Apresentaremos agora a especificação em VDM de uma parte do sistema de transformação. O subsistema considerado realiza a tradução de tipos. Como mostrado, pela proximidade dos tipos de SML e os de VDM, a tradução é direta, parcialmente sintática e resume-se basicamente a impor uma ordem em um subconjunto de definições de tipo de VDM, as definições que são traduzidas para declarações `type` em SML, isto é, as declarações que dão novos nomes a tipos já definidos. As outras definições são traduzidas como `datatype` em SML, não precisando estar ordenadas.

Assim consideraremos que uma declaração de tipo define um tipo como sendo igual a um tipo básico da linguagem, ou como igual a um tipo definido por outra declaração. Logo, dado um conjunto de declarações de tipo em VDM, por exemplo:

$C = \text{BOOL}$

$B = D$

$E = C$

$D = C$

queremos obter uma equivalente em SML. Pode existir mais de uma definição equivalente. Uma solução é:

```
type C = bool
 and D = C
 and B = D
 and E = C;
```

onde um tipo só é usado depois de definido. Outra opção é:

```
type C = bool
  and E = C
  and D = C
  and B = D;
```

Relembrando, uma declaração de tipo é modelada pelo objeto composto

TypeDec :: *tpname* : *TypeName*
type : *Type*

onde

TypeName, is not yet defined

e só consideramos tipos básicos e nomes de tipo, nas expressões de tipo:

$Type = BasicType \mid DecType$

$BasicType = \text{NAT} \mid \text{REAL} \mid \text{BOOL}$

$DecType = TypeName$

As restrições de contexto, nas declarações de tipo de VDM e SML, são estabelecidas por funções, que determinam se um objeto é bem-formado (*Well-Formed* [BSI89]). Por exemplo, em uma declaração de tipo não é permitido definir um tipo em função dele mesmo. Portanto, a definição:

$A = A$

não é permitida. Esta propriedade é verificada pela seguinte função:

$WF\_TypeDec : TypeDec \rightarrow \text{B}$

$WF\_TypeDec(mk\text{-}TypeDec(tpid, tp)) \triangleq$
$\quad \text{is-}DecType(tp) \Rightarrow tp \neq tpid$

As declarações em VDM são modeladas por:

$TypeDecs = \text{set of } TypeDec$

e obedecem as seguintes restrições: dado um conjunto de declarações de tipo em VDM, não pode existir duas declarações definindo o mesmo tipo. Assim, o grupo de declarações a seguir não é permitido:

$A = B$

$A = \text{NAT}$

As declarações não podem ser cíclicas, isto é, um tipo não pode ser definido em função de um outro declarado em função do primeiro. Assim,

$B = A$

$A = B$

não é permitido. Toda declaração de tipo definida em função de outro tipo (não primitivo) faz referência a um tipo pertencente ao conjunto de declarações. Assim não podemos ter:

$A = B$

$C = \text{REAL}$

pois o tipo B não é declarado. As restrições anteriores são especificadas pela seguinte função:

$$WF\_TypeDecs : TypeDecs \rightarrow \text{B}$$

$$\begin{aligned}
&WF_TypeDecs(s) \triangleq \\
&\quad (\forall d, e \in s \cdot d \neq e \Rightarrow tpname(d) \neq tpname(e)) \\
&\quad \wedge\, noCycles(s) \\
&\quad \wedge\, (\forall e \in s \cdot \\
&\qquad \text{is-}DecType(type(e)) \Rightarrow \\
&\qquad\quad (\exists d \in s \cdot type(e) = tpname(d)))
\end{aligned}$$

onde $noCycles(s)$, definido em [BM90], é verdadeiro quando o conjunto de definições s não tem ciclos.

As declarações de tipo em SML são representadas por:

$$SMLTypeDecs = \text{seq of } TypeDec$$

obedecendo as restrições a seguir: o conjunto formado pelos elementos da seqüência não tem ciclos, e todo tipo referenciado é definido antes, na seqüência de declarações, da definição do tipo que faz referência. Estas restrições são formalizadas pela função:

$$WF\_SMLTypeDecs : SMLTypeDecs \rightarrow \text{B}$$

$$\begin{aligned}
&WF_SMLTypeDecs(s) \triangleq \\
&\quad noCycles(\text{elems } s) \\
&\quad \wedge\, (\forall i \in \{1, \ldots, \text{len } s\} \cdot \\
&\qquad \text{is-}DecType(type(s(i))) \Rightarrow \\
&\qquad\quad (\exists j \in \{1, \ldots, i-1\} \cdot type(s(i)) = tpname(s(j))))
\end{aligned}$$

Finalmente a especificação da função que transforma declarações de tipo em VDM para declarações de tipos em SML é:

$$\begin{aligned}
&Vdm_Sml\ (td: TypeDecs)\ std: SMLTypeDecs \\
&\text{pre } WF_TypeDecs(td) \\
&\text{post elems } std = td \\
&\quad \wedge\, WF_SMLTypeDecs(std) \\
&\quad \wedge\, (\forall i, j \in \{1, \ldots, \text{len } std\} \cdot \\
&\qquad i \neq j \Rightarrow tpname(std(i)) \neq tpname(std(j)))
\end{aligned}$$

que recebe declarações de tipo válidas (obedece as restrições de contexto) em VDM e dá como resultado declarações equivalentes válidas em SML. A função é especificada num alto nível de abstração, apenas estabelece as propriedades do resultado em função do parâmetro, e não explicita como este resultado pode ser calculado. Diz apenas que o resultado é uma declaração válida em SML, que os componentes da seqüência resultado são os mesmos do conjunto parâmetro, e que a declaração em SML não tem tipos definidos duas vezes.

Como a especificação da função não é explícita, não podendo ser traduzida diretamente para SML, temos que obter uma definição executável que seja uma implementação desta. Isto é feito pela escolha de um algoritmo simples que satisfaça a especificação.

Através de um refinamento superficial da função *Vdm_Sml*, obtemos o desejado, com a seguinte especificação explícita:

$$Vdm\_Sml1 : TypeDecs \to SMLTypeDecs$$

$$\begin{array}{l} Vdm\_Sml1(td) \triangleq \\ \quad \textsf{if } WF\_TypeDecs(td) \\ \quad \textsf{then if } td \neq \{\,\} \\ \quad\quad \textsf{then let } s = \{e \mid e \in td \bullet \forall f \in td \cdot \text{is-}DecType(type(f)) \Rightarrow \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad type(f) \neq tpname(e)\} \textsf{ in} \\ \quad\quad\quad \textsf{let } e \in s \textsf{ in} \\ \quad\quad\quad Vdm\_Sml1(td - \{e\}) \frown [e] \\ \quad\quad \textsf{else } [\,] \\ \quad \textsf{else } [\,] \end{array}$$

O algoritmo escolhe os elementos, de um conjunto de declarações de tipo, que não são referenciados por nenhuma declaração de tipo pertencente a este conjunto (sempre existe um elemento com esta propriedade, se não existisse a definição seria cíclica; a declaração sem estes elementos é ainda uma declaração válida). Estes elementos podem ser colocados no final da seqüência de declarações de SML, e o processo é repetido, considerando que uma declaração de tipo equivalente ao conjunto vazio é a seqüência vazia.

Para provarmos a corretude do refinamento temos que descartar a seguinte obrigação de prova:

$$td \in TypeDecs \vdash WF\_TypeDecs(td) \Rightarrow \text{post-}Vdm\_Sml(td, Vdm\_Sml1(td))$$

que estabelece que *Vdm_Sml1* é uma implementação de *Vdm_Sml* e pode ser facilmente provada, o que é feito em [BM90]. A condição

$$\text{post-}Vdm\_Sml(td, Vdm\_Sml1(td))$$

é uma abreviação da pós-condição da função *Vdm_Sml*, onde *std* é substituído por $Vdm\_Sml1(td)$.

VDM	SML
$S - R$	`S difset R`
$S \frown R$	`S concat R`
let $e \in S$ in	`let val e = choice(S) in`
$\{e \mid e \in S \bullet p(e)\}$	`setcomp(S,p)`
$\forall e \in S \cdot p(e)$	`forall S p`
$\{i,\ldots,j\}$	`setint(i,j)`
$\{\}$	`emptyset(eq)`
$\{e\}$	`singleton(e,eq)`
$[\,]$	`emtptyseq(eq)`
$[e,f,g]$	`makeseq([e, f, g],eq)`

Tabela 1: Operadores em VDM e SML.

6 Protótipo do Sistema de Transformação de Tipos

Como chegamos numa especificação explícita do problema podemos obter um protótipo diretamente. Este protótipo será utilizado para a tradução de tipos da especificação da outra parte do sistema - tradução de operações.

Os tipos da especificação são traduzidos como indicado na seção 4. As funções são traduzidas para SML como apresentado em [BM89a]. As estruturas que precisamos ter em SML para implementar as funções da especificação em VDM são:

- if-then-else, let-in: estruturas equivalentes são disponíveis em SML.
- Os operadores de VDM e os correspondentes em SML são apresentados na Tabela 1, de acordo com a implementação de [BM89a]. Na tabela, `eq` é o operador de igualdade dos elementos do objeto (seqüência ou conjunto).
- $A \Rightarrow B$ é implementado por: `if A then B else true`.
- Seletores (ex: *tpname*, *type*) são implementados como na seção 4.
- *is-DecType* é implementado por casamento de padrão. Um elemento é do tipo *DecType* se é precedido pelo construtor `DecType`.

Assim, a função correspondente a *Vdm_Sml*1 em SML é:

```
fun Vdm_Sml1 (td:TypeDecs):SMLTypeDecs =
if WF_TypeDecs(td)
then if not(td TypeDecs_eq emptyset(TypeDecs_eq))
     then let val
              g = (fn e => forall td
                           (fn f => case type(f) of
                                DecType t => t <> tpname(e) |
                                _ => true)
              val s = setcomp(td,g)
              val e = choice(s)
          in Vdm_Sml1(td difset singleton(e,TypeDecs_eq))
           concat makeseq([e],SMLTypeDecs_eq)
          end
     else emptyseq(SMLTypeDecs_eq)
else emptyseq(SMLTypeDecs_eq)
```

onde `fn e => exp` é uma forma de abstração lambda, equivalente a $\lambda e \cdot exp$. A expressão `case e of p1 => e1 | _ => e2` é equivalente à expressão `e1` (avaliada com as ligações geradas pelo casamento de padrão) se a expressão `e` casa com o padrão `p1`, caso contrário é a expressão `e2`. TypeDecs_eq e SMLTypeDecs_eq são os operadores de igualdade de TypeDecs e SMLTypeDecs, obtidos de acordo com [BM89a]. As outras funções são implementadas da mesma forma.

7 Conclusões

Apresentamos um método de transformação de uma especificação em VDM para um protótipo em SML. Além disso descrevemos a tradução de tipos segundo o método. No final, descrevemos a especificação e o protótipo da parte de tradução de tipos do sistema de transformação baseado no método apresentado.

Pela proximidade dos tipos de SML e de VDM, a tradução é, em parte, sintática, e resume-se basicamente a impor uma ordem em um subconjunto de definições de tipo de VDM. A tradução de tipos é feita automaticamente pelo sistema, inclusive com geração de seletores e operadores de igualdade para os tipos definidos.

Como forma de validação da especificação e prototipagem rápida, a ferramenta facilita o desenvolvimento formal de software. O método definido pode ser aplicado (com mínimas alterações) a outros métodos construtivos de especificação, como Z e RAISE, e independe da linguagem funcional sendo utilizada (SML).

Como trabalhos futuros terminaremos o desenvolvimento do transformador (utilizando o protótipo gerado para a tradução de tipos) e aumentaremos o conjunto de regras de transformação disponíveis. Aí então, poderemos avaliar melhor a utilização do método e ferramenta.

Referências

[Bar89] G. Barrett. Formal Methods Applied to a Floating-Point Number System. *IEEE Transactions on Software Engineering*, 15(5), May 1989.

[BM89a] P. H. M. Borba e S. R. L. Meira. Notação VDM Executável em Standard ML. Em *Anais do IX Congresso da Sociedade Brasileira de Computação*, Uberlândia - MG, Julho 1989.

[BM89b] P. H. M. Borba e S. R. L. Meira. Protótipos funcionais a partir de Especificações em VDM. Em *Anais do III Simpósio Brasileiro de Engenharia de Software*, Recife - PE, Outubro 1989.

[BM90] P. H. M. Borba e S. R. L. Meira. *Sistema de Transformação de VDM para SML - Tradução de Tipos*. Relatório Técnico, Departamento de Informática, Universidade Federal de Pernambuco, 1990.

[BSI89] BSI. *VDM Specification Language - Proto-Standard*. Relatório Técnico, Draft, BSI IST/5/50, June 1989.

[Hay87] I. Hayes, editor. *Specification Case Studies. C. A. R. Hoare Series Editor*, Prentice-Hall, 1987.

[HMM86] R. Harper, D. MacQueen, e R. Milner. *Standard ML*. Relatório Técnico ECS-LFCS-86-2, Edinburgh University, 1986.

[Jon86] C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1986.

[JS89] M. Johnson e P. Sanders. From Z Specifications to Functional Implementations. Em *Z Technical and Users Meeting*, December 1989.

[Mid89] C. A. Middelburg. VVSL: A Language for Structured VDM Specifications. *Formal Aspects of Computing*, 1(1), 1989.

[NHWG89] M. Nielsen, K. Havelund, K. Wagner, e C. George. The RAISE Language, Method and Tools. *Formal Aspects of Computing*, 1(1), 1989.