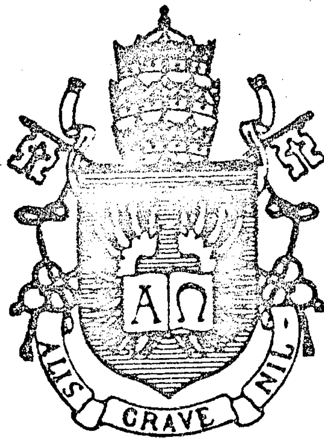

PUC

Série: Monografias em Ciência da Computação,
No. 16/89

A DENOTATIONAL MODEL OF TYPES

Clovis T. Fernandes

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 – CEP-22453
RIO DE JANEIRO – BRASIL

PUC/RIO - DEPARTAMENTO DE INFORMÁTICA

Série: Monografias em Ciência da Computação, 16/89

Editor: Paulo Augusto Silva Veloso julho, 1989

A DENOTATIONAL MODEL OF TYPES[1]

CLOVIS TORRES FERNANDES[2]

[1]Presented by Prof. Daniel Schwabe and Prof. Raul César Baptista Martins

[2]On leave from Computer Sciences Division, ITA – Technological Institute of Aeronautics; work partially financed by CAPES/PICD.

In charge of publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC RIO, Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453 - Rio de Janeiro, RJ
BRASIL

Tel.:(021)529-9386 TELEX:31078 FAX:(021)274-4546
BITNET: userrtlc@lncc.bitnet

ABSTRACT

A model of types for use in both programming and specification languages is presented. The denotational approach is used to build up minimal, closed universes of denotations for all required values. Standard types consisting of finitary values are given set-theoretic denotations, whereas just a simplified use of the Scott domain theory is enough to provide denotations for the remaining constructs, such as expressions and procedures. This model is applied to the VDM metalanguage as an illustration, since it has a rich type set.

Keywords:

type model, type universes, denotational semantics, Scott universes, VDM metalanguage.

RESUMO

Um modelo de tipos para uso tanto em linguagens de programação quanto de especificação é apresentado. A abordagem denotacional é usada para construir gradualmente universos de denotacões mínimos e fechados para todos valores requeridos. Dão-se denotacões de conjuntos para tipos padrões consistindo de valores finitários, ao passo que um uso simplificado da teoria de domínio de Scott é suficiente para fornecer denotacões para as construções restantes, tais como expressões e procedimentos. Este modelo é aplicado à metalanguage VDM como ilustração, devido ao seu rico conjunto de tipos.

Palavras-chaves:

modelo de tipos, universos de tipos, semântica denotacional, teoria de domínio de Scott, metalinguagem VDM.

Contents

1	Introduction		1
2	A type definition sublanguage		1
3	Recursive domain equations		5
4	Type universes		7
	4.1	Basic Types	7
	4.2	Operators	8
	4.3	Closure conditions	8
5	Extended type universes		10
	5.1	The subset closure operator	10
	5.2	Flat domain embedding operator	11
6	Scott Universes		12
7	Applying the model to VDM		14
	7.1	The VDM type definition sublanguage	14
	7.2	The VDM type universes	15
8	Conclusions		17

1 Introduction

A model of types for use in both programming and specification languages is presented. This work attempt to give a simple account of recursively defined data types besides modelling recursive function spaces not involving the full apparatus surrounding the use of Scott domain theory (Scott, 1982).

The work consists of analysing the paper presented by Monahan (1987) and clarifying some obscure points. To start with, his work was directed only to the VDM metalanguage. Nonetheless, I believe his approach can be extended to all kind of languages. So, the main effort was to extract the essential features of his type model and to show that this can be applied to any language, such as Pascal, Z, Ada, etc. Figure 1 shows that type models for any language can be derived from the essential type model presented here.

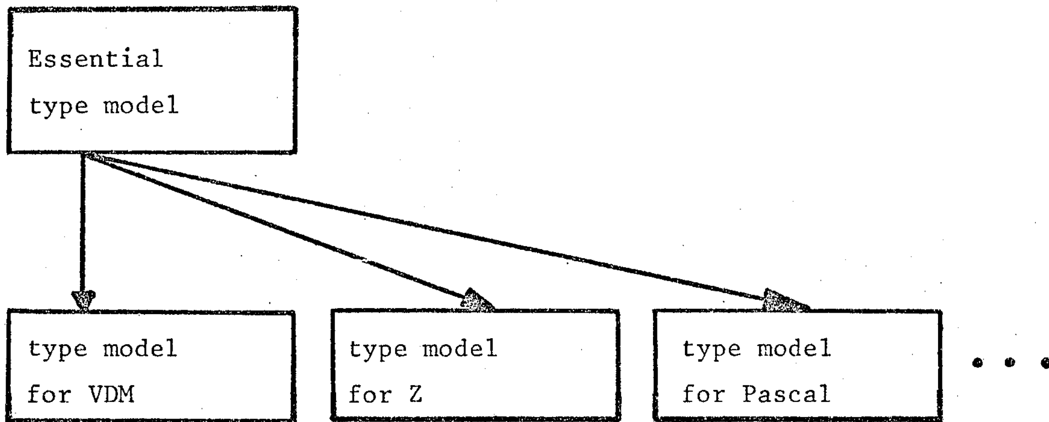

Figure .1: The intended scope of the model

The denotational approach is used to constructively get a minimal, closed universe of denotations for all required values. Initially, a type definition sublanguage is presented and it is shown how to get the denotations for domain equations, including recursive ones. Next, it is shown how to build up a type universe containing all solutions for recursive type definitions. After that, it is shown how to build up a Scott universe, which takes into consideration the concept of invariant and other means of taking subsets besides containing enough continuous functions, domain discriminated unions and domain products to give a type model rich enough to interpret the usual languages. Finally, this model is applied to the VDM metalanguage as an illustration, since it has a rich type set.

2 A type definition sublanguage

Types are generally taken to be sets of values with specific operations. Here a data type is a collection of data values that have been grouped together for reasons of similarity of structure or perhaps mere convenience. One can have languages with only one type: BCPL,

with the type word (implicit), is such an example. One can also have languages with only a few basic types. For instance, FORTRAN has only the types integer, real, complex,boolean, and dimension. As the set of types of a language becomes richer, and its set of definable types becomes infinite, it is useful to define the set of types by a type definition sublanguage, whose syntax can generally be specified by a context-free grammar.

The type of a language construct will be specified by a *type expression or definition* which is declared in advance following the specified syntax. Informally, a type expression is either a basic type or is formed by applying an operator called a *type constructor* to other expressions. Usually, the basic (primitive) types are integer, real, boolean, char, rational, etc. Each basic type is denoted by a semantic set (domain). So, the basic type of integers is denoted by the set of integers, and so on. All values obtained from variables or expression of integer type belong to the set of integers, for instance. Each type constructor is denoted by a set (domain) constructor. For instance, the discriminated union type constructor of basic types is denoted by the set discriminated union constructor of basic sets. And finally, any type definition involving type constructors is denoted by a set (domain) definition involving set (domain) constructors. The use of the terms sets and domains is interchangeable up to the formal definition of domains.

A type declaration should use clear patterns to define structural properties of all possible instances of that type, independent of their size or the component values involved. A type declaration should also allow to associate a complex structure description with a *type name*. A *type system* is a collection of rules for assigning type definitions to the various parts of a program and a *type checker* implements a type system. However, these subjects are outside the scope of this work. The following is the syntactic description of the sublanguage adopted here:

simple-type-def	::=	type-name = type
type	::=	basic-type | type-name | constructed-type
basic-type	::=	bool | nat | rac | real | char | ...
constructed-type	::=	func(type,type) |
		prod(type,type) |
		coprod(type,type)

However, we are interested not only in the syntax of the type definition, but mainly in its semantics. We are interested in what types denote, that is, what their permissible values are.

We have adopted the denotational approach to semantics. What does this mean? Firstly, one makes use of mappings, called semantic interpretation or evaluation functions, that map values of syntactic domains into semantic domains (or domains of meanings). The following are some of the domains considered:

type expressions	⟶	type sets
integer numerals	⟶	set of integers
structures and records	⟶	cartesian products of semantic domains
expression	⟶	mathematical function domains
procedures	⟶	mathematical function domains

Secondly, semantic mappings are defined so that the meaning of any composite syntactic structure is expressed in terms of the meanings of its immediate constituents.

The use of this approach implies that:

1. every syntactic structure is mapped into a unique meaning, so that issues like incompleteness, inconsistency, and semantic ambiguity simply don't arise.

2. semantic functions and meanings are mathematical objects, so that standard mathematical techniques may be used to prove results about their properties.

A denotational approach to language definition must therefore make clear several sets of quantities:

1. the alphabet Σ of the syntactic items.

2. the syntactic domains.

3. the semantic domains.

4. the (abstract) syntax rules.

5. the definition of the various semantic functions, making explicit their domains and respective codomains.

Ex.: Let $\Sigma = \{\ldots,\text{bool},\text{nat},\text{int},\text{rac},\text{func},\text{prod},\text{coprod}\}$ and let W = the language generated by the syntax above. The basic syntactic subdomain of W is:

$$D_{\text{basic-type}} = \{\text{int},\text{nat},\text{bool},\text{rac},\text{char},\ldots\}$$

The corresponding basic semantic subdomain is:

$$S_{\text{basic-type}} = \{\underline{\text{int}}, \underline{\text{nat}}, \underline{\text{bool}}, \underline{\text{rac}}, \underline{\text{char}}, \ldots\}$$

where its elements can be interpreted in the following usual way:

$$\begin{array}{lcl} \underline{\text{int}} & = & \{\ldots,-1,0,1,2,\ldots\} \\ \underline{\text{nat}} & = & \{0,1,2,\ldots\} \\ \underline{\text{bool}} & = & \{\underline{\text{TRUE}},\underline{\text{FALSE}}\} \\ \underline{\text{rac}} & = & \text{the set of the rational numbers} \\ \underline{\text{char}} & = & \{a,b,\ldots,z,A,B,\ldots,Z,0,1,\ldots,9\} \\ \vdots & & \end{array}$$

Let W be the set defined above and U a family of arbitrary sets. The evaluation function eval: $W \rightarrow U$ is recursively defined by:

(i) $\text{eval}(N) = N_U$ for all constant symbols $N \in W$.

(ii) $\text{eval}(N(t_1,\ldots,t_n)) = N_U(\text{eval}(t_1),\ldots,\text{eval}(t_n))$ for all $N(t_1,\ldots,t_n) \in W$.

Example:

$$
\begin{aligned}
\text{eval(int)} &= \underline{\text{int}} \\
\text{eval(nat)} &= \underline{\text{nat}} \\
\text{eval(bool)} &= \underline{\text{bool}} \\
\text{eval(rac)} &= \underline{\text{rac}} \\
\text{eval(char)} &= \underline{\text{char}} \\
\text{eval(func(A, B))} &= \text{eval(A)} \longrightarrow \text{eval(B), ie., the set of all partial mappings} \\
&\quad \text{from eval(A) to eval(B), where } A, B \in W, \text{ and eval(A)} \\
&\quad \text{denotes a finite set.} \\
\text{eval(prod(A, B))} &= \text{eval(A)} \times \text{eval(B), where } \times \text{ is the cartesian product operator} \\
&\quad \text{of sets.} \\
\text{eval(coprod(A, B))} &= \text{eval(A)} + \text{eval(B), where } + \text{ is the discriminated union operator} \\
&\quad \text{of sets.}
\end{aligned}
$$

We treat the evaluation of the item type-name the following way:

$$\text{eval(type-name)} = \underline{\text{type} - \text{name}},$$

where $\underline{\text{type} - \text{name}}$ corresponds to a name of a set of permissible values.

The syntactic rule part type-name = type, whose abstract syntax is given without the = signal, gives rise to semantic equations in which the set denoted by type-name corresponds to the set denoted by type. Formally, we have the following:

$$\text{eval(type-name type)} \triangleq (\text{eval(type-name)} = \text{eval(type)})$$

These kind of equations are called domain equations. The intuitive meaning is that we can assign a name to a set and we can use this name in other equations instead of specifying the set again.

For instance, the type definitions below

n = nat
b = bool
p = prod(n,b)
c = coprod(b,p)

correspond to the following domain equations

$$
\begin{aligned}
&\underline{n} = \underline{\text{nat}} \\
&\underline{b} = \underline{\text{bool}} \\
&\underline{p} = \underline{n} \times \underline{b} \\
&\underline{c} = \underline{b} + \underline{p}
\end{aligned}
$$

where $\underline{c}$ corresponds to the set $\underline{\text{bool}} + (\underline{\text{nat}} \times \underline{\text{bool}})$.

Next, the problem of recursive domain equations will be considered. Besides that, it will be presented a solution that will serve as a paradigm for the type model shown in this report.

3 Recursive domain equations

The type definition sublanguage presented above allows to construct infinite domains through recursive domain equations. So long as the domains are constructed from basic ones using only the operations described above, Scott's theory guarantee that appropriate solutions of these equations exist (Stoy, 1977). But how will they be constructed?

Before showing that, let us consider the following Pascal-like definition, which is a more familiar notation (Tennent, 1981):

```
type string = record case isnull:boolean of
                     true: ();
                     false: (first:char; rest:string)
              end;
```

Each of the elements of the domain it describes is either null, or it consists of a 'first' component, which is a character, and a 'rest' component which is a string itself. A variant record can be modeled by discriminated union and a sequence by a cartesian product, so we obtain the following using the syntax above:

string = coprod({null}, prod(char,string))

where $\{null\}$ stands for the n-tuples of elements of char, $n = 0$, that is, $char^0 = \{null\}$.

The mathematical counterpart of this type definition is the following:

$$\text{eval(string)} = \text{eval(coprod(\{null\}, prod(char,string)))}$$
$$\underline{\text{string}} = \langle\rangle + (\underline{\text{char}} \times \underline{\text{string}}) \qquad (\star)$$

where $\underline{\text{string}}$ denotes the domain of strings and $\langle\rangle$ represents the empty string. Let us look into the following infinite set:

$\langle\rangle$	- the empty string
$\langle c \rangle \ \forall c \in \underline{\text{char}}$	- all strings of length one
$\langle c_1, c_2 \rangle \ \forall c_1, c_2 \in \underline{\text{char}}$	- all strings of length two
$\vdots$	
$\langle c_1, c_2, \ldots, c_n \rangle \ \forall c_i \in \underline{\text{char}},\ i = 1, \ldots, n$	- all strings of length n
$\vdots$	

Every element of this set is an element of $\langle\rangle + (\underline{\text{char}} \times \underline{\text{string}})$ and every element of $\langle\rangle + (\underline{\text{char}} \times \underline{\text{string}})$ is an element of this set. In fact, this set is the smallest solution of equation $(\star)$. In general, one can consider a recursive domain equation of the form

$$D = \cdots \ D \ \cdots$$

where the dots can stand for an arbitrary, correct combination of both basic domains, domains and domain operations. D is defined to be the smallest solution (up to isomorphism) of the equation. So, the equation $(\star)$ denotes the domain of all finite sequences of elements of $\underline{\text{char}}$. It is usual to designate this domain by $\underline{\text{char}}^*$.

Among the strictly constructive processes for solving this kind of equations, the following is undertaken. It involves defining a hierarchy of subspaces $D_0, D_1, D_2, \ldots$, where

$$D_{n+1} = \cdots \; D_n \cdots$$

then suitably embed each D_n in D_{n+1} and take limits on both sides as $n \rightarrow \infty$. That is, the infinite domain that it is the smallest solution to (up to isomorphism) a recursive domain definition is the limit of a sequence of domains that approximate it. As an illustration of this process, consider the following equation:

$$N = \{zero\} + N, \quad \text{where } N^0 = \{zero\}$$

Following the process previously sketched, define a sequence of domains N_i, for $i \geq 0$, as follows:

$$\begin{aligned} N_0 &= \{\,\} \\ \forall i \in \omega . N_{i+1} &= \{zero\} + N_i \end{aligned}$$

where ω corresponds to the natural number set.

Note that each domain in the hierarchy (except the initial one) is defined by using the equation, but with the recursive occurrence of the domain name replaced by the preceding domain in the hierarchy, so that there is no longer any circularity. Let us look into the sequence of domains that approximate the smallest solution to (up to isomorphism) this equation:

$$\begin{aligned} &N_0 = \{\,\} \text{ denoting an undefined value}(\perp) \\ &N_1 = \{zero\} + \{\,\} \text{ denoting } \{0\} \\ &N_2 = \{zero\} + \{zero\} + \{\,\} \text{ denoting } \{0, 1\} \\ &\vdots \\ &N_k = \{zero\} + \{zero\} + \cdots + \{zero\} + \{\,\} \text{ denoting } \{0, 1, \cdots, (k-1)\} \\ &\vdots \end{aligned}$$

The elements of each N_i are 0 and its sucessors up to, but not including, the ith. Domains N_i are finite approximations to (i.e. subsets of) the desired infinite solution of the equation, so that if we take the limit of this sequence to be the union of all these approximating domains, we obtain the set of all the natural numbers, the desired solution of the equation above:

$$\text{Natural numbers} = \left(\bigcup_{i \in \omega} N_i \right)$$

It can be shown that in this framework, the smallest solution (up to isomorphism) of recursive domain equations using operations $\times, +$ and $\rightarrow$ always exist (Stoy, 1977). This is not true if arbitrary sets and functions are allowed; for instance, if $V \rightarrow V$ must contain all the functions from V to V, then the only solution of the equation $V = D + (V \rightarrow V)$ is a trivial (one-point) one when D is the empty set. But if $V \rightarrow V$ is taken to be the domain of all continuous functions, then the smallest solution (up to isomorphism) does exist, for any domain D, according to the definition of domain given subsequently. Domains containing their own (continuous) function space are called reflexive.

In the context of data type definition, the operation $\rightarrow$ is applied only on a finite set, being consequently continuous. In other contexts will be necessary to specify explicitly this

requirement of continuity. For instance, modeling recursive function space implies to restrict the whole universe of functions just to the continuous ones.

These are the kind of sets we are interested in order to begin to construct a type universe (or universal domain). In fact, all of the type universes shown here will be constructed using this model (Tennent, 1981; Allison, 1985). The basic idea is to form a "tower" of (families of) sets by iteration of a function mapping families to families, starting from a given family, namely the basic types. The type universe is then obtaind by taking the union over this tower of families to give a single family of sets, and then closing up under union of countable chains of sets to ensure that recursive type definitions will always have a solution within the type universe (a chain is an example of a directed set and a set X is a directed set if every finite subset of X has an upper bound in X; a chain models the behaviour of better and better aproximations to a solution). That way, the type universe constructed will be a minimal, closed one.

4 Type universes

The task is to specify what all language values look like, and to say which types they belong to. Therefore, it must be found out a **model**, that is, a system which satisfies all the required constraints. There are two approaches to solve this problem:

(i) we could construct one specifically for the purpose;

(ii) we could choose a suitable candidate from systems we already know and check that it satisfies our requirements.

Following Monahan, the first approach above was chosen using the constructive denotational model seen before. Initially, it will be defined a type universe U, which contain all solutions for well-formed recursive type definitions. By this stage, only sets of values are needed, for the permissible values can only be propers (that is, different of $\perp$, the undefined value), according to the construction previously shown. Later it will needed domains for modelling procedures, which can return a undefined value ($\perp$) and be defined by recursion, in order to specify a special universe SU, called Scott Universe. A domain is formally defined as having a set of values, an ordering between them and a least element.

The type universe U is a collection of sets in which any required set may be found as one of its subsets. It is also required that this universe is minimal, that is, it is sufficiently rich, not having more than the necessary number of values sets. Moreover, all values are denoted by a ground term, that is, they are finitely generated.

The type universe will be constructed from the knowledge we already have of the look of the permissible values sets, through the sets they denote. This can be done conceptually, by examining the elements of each type belonging to the type universe, from the basic types to the structured ones.

4.1 Basic Types

There is a collection, named Bty, of specific sets that are to be regarded as basic types:

$$\text{Bty} = \{\underline{\text{int}}, \underline{\text{nat}}, \underline{\text{rac}}, \underline{\text{bool}}, \underline{\text{char}}, \ldots\}$$

where

- $\underline{\text{bool}} = \{\underline{\text{TRUE}}, \underline{\text{FALSE}}\} \wedge |\, \underline{\text{bool}} \,| = 2$ and

  $\forall \text{ty} \in \text{Bty}.(\underline{\text{bool}} \cap \text{ty}) \neq \emptyset \Rightarrow \underline{\text{bool}} = \text{ty}$

- The sets $\underline{\text{int}}$, $\underline{\text{nat}}$, $\underline{\text{char}}$ and $\underline{\text{rac}}$ correspond to the integer set, the natural numbers, the set of lowercase and uppercase letters plus decimal digits, and the rational numbers respectively, as usual.

- Each set is finite or countably infinite one, that is:

  $$\text{ty} \in \text{Bty} \Rightarrow |\, \text{ty} \,| \in \omega \;\text{ or }\; |\, \text{ty} \,| = \omega$$

We consider two disjoint countably infinite sets named Text and Atom, whose elements can be literally expressed. The sets Text and Atom are also included as basic types.

4.2 Operators

At this stage, the following notation is introduced:

$\mathcal{M}(S_1, S_2)$	–	the set of all partial mappings from S_1 to S_2 with finite domain of definition.
$\mathcal{F}(S)$	–	the set of all finite subsets of the set S.
$\mathcal{R}(F)(rt)$	–	the set of record values that conforms to the finite mapping rt, from names to sets, that is, $rt \in \mathcal{M}(\text{ATOM}, F)$, where F is a family of sets.

There is also a *tagging* operation that takes a token $tk \in \text{Text}$ and an arbitrary set S and produces an isomorphic copy of S, denoted by (tk::S). This operation is assumed to have the following properties, where S, S_1, and S_2 stand for any sets:

$$\forall x.\forall tk \in \text{Text}.\ |\, tk :: \{x\} \,| = 1$$
$$\forall tk \in \text{Text}.(tk :: S) = \cup\{(tk :: \{a\}) \mid a \in S\}$$
$$\forall tk_1, tk_2 \in \text{Text}.\ (tk_1::S_1) \subseteq (tk_2::S_2) \Leftrightarrow (tk_1 = tk_2) \wedge (S_1 \subseteq S_2)$$
$$\forall tk_1, tk_2 \in \text{Text}.(tk_1 \neq tk_2) \Rightarrow (tk_1 :: S_1) \cap (tk_2 :: S_2) = \emptyset$$
$$\forall tk \in \text{Text}.(tk :: S) \cap S = \emptyset$$

4.3 Closure conditions

A type universe is closed with regard to specific operations on sets when these operations are applied to any appropriate choice of sets from U and the resulting set also belongs to U.

Let

$$\text{eval}(\text{func}(A_W,B_W)) = \underline{\text{func}}(A,B), \text{ where } A_W,B_W \in W \text{ and } A,B \in U.$$
$$\text{eval}(\text{prod}(A_W,B_W)) = \underline{\text{prod}}(A,B).$$
$$\text{eval}(\text{coprod}(A_W,B_W)) = \underline{\text{coprod}}(A,B).$$

Suppose that $A,B \in U$. So, these operators, specified as follows, are to be closed:

$$\begin{aligned}
\underline{\text{func}}(A,B) &= \text{tag}(\text{"func"}, \mathcal{M}(A,B)) \in U \\
\underline{\text{prod}}(A,B) &= (A \times B) \in U \\
\underline{\text{dunion}}(A,B) &= (A+B) \in U, \text{where}(A+B) = \text{tag}(\text{"tag1"},A) \cup \text{tag}(\text{"tag2"},B), \\
& \quad \text{"tag1"}, \text{"tag2"} \in \text{Text} \\
\text{tag}(\text{tk},A) &= (\text{Gentag}(\text{tk}) :: A) \in U, \ \forall\, \text{tk} \in \text{Text}
\end{aligned}$$

The function Gentag:Text $\rightarrow$ Tag is used in the definition of the tag operator to guarantee that the tagged set so formed is unique. Tag is a countable set isomorphic to Text, such that Tag does not intersect any set belonging to the type universe.

Each of the basic type constructors are appropriately tagged in order to prevent any undesired identifications within the model. In addition, all this type operators are monotonic and continuous, with respect to set inclusion.

The construction of the type universe U (basically a particular family of sets) is given such that it satisfies the basic closure requirements. This universe is intended to be the space of denotations for the regular type denotations, disregarding any invariants and other means of taking subsets.

As it was previously said, the basic idea is to form a "tower" of (families of) sets by iteration of a function mapping families to families, starting from a given family, namely the basic types Bty. The type universe U is then obtained by taking the union over this tower of families to give a single family of sets, and then closing up under unions of countable chains of sets. The latter statement can be put in the following formal way:

$$\forall S \in U^{\omega}.(\forall i \in \omega. S_i \subseteq S_{i+1}) \Rightarrow \left(\bigcup_{i \in \omega} S_i \right) \in U$$

This says that for every sequence $< S_i >_{i \in \omega}$ of sets from U, if the sequence is an ascending chain, then the union of the chain also belongs to U. With this, one can define the closure operator OMEGA as follows:

$$\text{OMEGA}(F) = \left\{ \bigcup_{j \in \omega} S_j \mid S \in F^{\omega} \wedge (\forall i \in \omega. S_i \subseteq S_{i+1}) \right\}$$

The following operators on families of arbitrary sets are now defined. Let F stand for an arbitrary family of sets:

$$\begin{aligned}
\text{FUNC}(F) &= \{\underline{\text{func}}(a,b) \mid a,b \in F\} \\
\text{PROD}(F) &= \{\underline{\text{prod}}(a,b) \mid a,b \in F\} \\
\text{COPROD}(F) &= \{\underline{\text{coprod}}(a,b) \mid a,b \in F\} \\
\text{TAG}(F) &= \{\underline{\text{tag}}(\text{tk},a) \mid \text{tk} \in \text{Text} \wedge a \in F\}
\end{aligned}$$

Define the following sequences of sets $< T_i >_{i\in\omega}$ by primitive recursion on the index i as follows:

$$\begin{aligned} T_0 &= \text{Bty} \\ \forall i \in \omega . T_{i+1} &= T_i \cup \text{FUNC}(T_i) \cup \text{PROD}(T_i) \cup \\ & \quad\ \text{COPROD}(T_i) \cup \text{TAG}(T_i) \end{aligned}$$

Each T_i is clearly a family of sets for each $i \in \omega$ such that $T_i \subseteq T_{i+1}$, for each $i \in \omega$. Finally, U can be defined by the following equations:

$$\text{U} = \text{OMEGA}\left(\bigcup_{i\in\omega} T_i \right)$$

That way one have denotations for all data types, including recursive ones. In the following, the type universes are extended to include denotations for all constructs in the language such as invariants, arithmetic expressions, procedures and functions (including recursive ones).

5 Extended type universes

The notion of type universe covered so far only provide (set theoretic) types for simple data structures used in the language. For example, it does not possess type denotations for general (user defined) functions or operations.

5.1 The subset closure operator

If the VDM notion of **invariant** is introduced in our language, we will obtain the following syntax:

type-def ::= simple-type-def |
simple-type-def **where inv**-ty-name(args) $\triangleq e$

where e should contain no free variables other than those given within args. An invariant stands for a condition which restricts the range of the permissible values of the object being defined, i.e., it stands for a condition of existence of this object.

The semantics intended is to allow to work with subsets of types since the invariant implies in a tighter description of the permissible values of the type under this condition. So, it is needed to introduce a new operator, Sub, in order to treat this fact.

The operator Sub has the following definition:

$$\text{Sub(F)} = \{s \mid \exists t \in \text{F}. s \subseteq t\} = \bigcup\{\mathcal{P}(t) \mid t \in \text{F}\}$$

Sub(F) is well defined for an arbitrary family of sets F, and it is easy to show that F $\subseteq$ Sub(F) and Sub(Sub(F)) = Sub(F). Therefore, if U is a type universe then Sub(U) is also a type universe but extended by all subsets of types.

To show that Sub(U) is a type universe it is necessary to show that it satisfies the condition given for type universes. It will shown that it satisfies such a closure condition for the dunion operation (+) as an example. The reasoning it will be similar for the other operations.

Suppose A,B $\in$ Sub(U), it will showed that dunion(A,B) $\in$ Sub(U):

- if A,B $\in$ U $\subseteq$ Sub(U), then dunion(A,B) $\in$ U $\subseteq$ Sub(U). Therefore, dunion(A,B) $\in$ Sub(U)
- if A,B $\in$ Sub(U) and A,B $\notin$ U, then , for the definiton of Sub, there exists C, D $\in$ U such that A $\in$ $\mathcal{P}$(C), B $\in$ $\mathcal{P}$(D) and consequently A $\subset$ C and B $\subset$ D. So, dunion(A,B) = tag("tag1",A) $\cup$ tag("tag2",B) $\subset$ dunion(C,D) = tag("tag1",C) $\cup$ tag("tag2",D), by the tagging properties seen before, and dunion(C,D) $\in$ Sub(U), by the previous item. Therefore dunion(A,B) $\in$ Sub(U).

5.2 Flat domain embedding operator

A domain $< D, \sqsubseteq_D, \bot_D >$ is a structure consisting of a carrier D, together with a chain complete partial ordering $\sqsubseteq$ and a least element, denoted by $\bot_D$. A complete partial order (cpo) $< S, \sqsubseteq >$ is a countable set S with a partial order $\sqsubseteq$, such that there is an element $\bot$ for which $\bot \sqsubseteq$ x for all x in S, and in which every ascending chain $x_1 \sqsubseteq x_2 \sqsubseteq x_3 \cdots$ has a least upper bound. That is, a domain is a cpo.

Flat domains are domains in which all the elements apart from $\bot$ are incomparable with each other. One can introduce a flat topology into a given domain, by adding the symbol $\bot$ to it and which stands for the result of executing a program not returning sensible output. In short, the improper element $\bot$ denotes *improper termination, undefined value* or *nonsense value*. The aim is to allow all functions with such a domain to be total.

Let $\bot$ be an arbitrary value and let F be an arbitrary family of sets. Consider the following set function defined for any s $\in$ F:

$$\forall s \in F. s_\bot = s \cup \{\bot\}$$

and consider the ordering relation $\sqsubseteq_S \subseteq (s_\bot \times s_\bot)$ defined by:

$$\forall x, y \in s_\bot.(x \sqsubseteq y) \Leftrightarrow (x = \bot) or (x = y) \qquad (**)$$

The definition (**) says that all $x, y \in s_\bot$ are incomparable each other, apart from $\bot$ and that $\bot$ is the least element. All ascending chain have length 2 with the following aspect:

$$\bot \sqsubseteq x, \forall x \in s_\bot$$

and x is the upper bound of each chain $< \bot, x >$. In fact, x is the least upper bound since there is only one upper bound: the x itself. So it was shown that $< s_\bot, \sqsubseteq_S, \bot >$ is a cpo.

It can now be defined the flat domain operator **FD**, that takes any set from **F** to a corresponding flat domain:

$$\mathbf{FD}(F) = \{< s_\bot, \sqsubseteq_S, \bot > \mid s \in F\}$$

Each set combining operation can be uniquely extended to operate upon flat domains in a standard way. Let $m,n \in \omega$ and consider any (set) operator $f : F^m \to F^n$. This extend uniquely to a (flat domain) operator $f_\perp : \mathbf{FD}(F)^m \to \mathbf{FD}(F)^n$ in the following way:

$$f_\perp(a_{1_\perp}, a_{2_\perp}, \cdots, a_{m_\perp}) = (b_{1_\perp}, b_{2_\perp}, \cdots, b_{n_\perp})$$

where $f(a_1, a_2, \cdots, a_m) = (b_1, b_2, \cdots, b_n)$

The following is obtained in the model:

- The flat basic domain are defined as:

$$\begin{aligned}\mathbf{FD}(\text{Bty}) \;=\; & \{< \underline{\text{int}}_\perp, \sqsubseteq_{int}, \perp_{int}>, < \underline{\text{nat}}_\perp, \sqsubseteq_{nat}, \perp_{nat}>, \\ & < \underline{\text{bool}}_\perp, \sqsubseteq_{bool}, \perp_{bool}>, < \underline{\text{rac}}_\perp; \sqsubseteq_{rac}, \perp_{rac}>, < \underline{\text{char}}_\perp; \sqsubseteq_{char}, \perp_{char}>, \cdots\}\end{aligned}$$

- The operation $\underline{\text{prod}}_\perp(A_\perp, B_\perp) = (\ \underline{\text{prod}}(A, B))_\perp$ where $\underline{\text{prod}}_\perp$ is known as smash (or strict) product and preserves flatness of domains:

$$\underline{\text{prod}}_\perp(\perp_A, B_\perp) = \underline{\text{prod}}_\perp(A_\perp, \perp_B) = (\perp_A, \perp_B) = \perp_{A\times B}.$$

- The operation $\underline{\text{coprod}}_\perp(A_\perp, B_\perp) = (\ \underline{\text{coprod}}(A, B))_\perp$ where $\underline{\text{coprod}}_\perp$ is known as coalesced sum and preserves flatness of domains:

$$\underline{\text{coprod}}_\perp(\perp_A, B_\perp) = \underline{\text{coprod}}_\perp(A_\perp, \perp_B) = \perp_{A+B}.$$

- The operation $\underline{\text{func}}_\perp(A_\perp, B_\perp) = (\underline{\text{func}}(A, B) \cup \underline{\text{func}}(\perp_A, \perp_B))_\perp$.
- The operation $\text{tag}_\perp(tk, A_\perp) = (\text{tag}(tk, A))_\perp$.

6 Scott Universes

In order to interpret the whole language it is necessary to provide this family of flat domains with enough (continous) functions, discriminated unions and domain products. A Scott universe is a family of domains closed under the formation of product domains, discriminated union domains, Scott continous function domains and the Bekic mapping domains. Suppose that f is an arbitrary flat domain and that d, d_1, and d_2 are arbitrary domains:

- **Scott Continuous Function Space** $[d_1 \to d_2]$ — Following the definition of domains, one have to present three components for this domain:

  $$< [D_1 \to D_2]; f \sqsubseteq_{D_1 \to D_2} g \Leftrightarrow (\forall x \in D_1 . f(x) \sqsubseteq g(x)); \lambda x \in D_1 . \perp_{D_2}>$$

  where D_1 and D_2 are the carriers of the domains d_1 and d_2 respectively; the first component, the carrier, is defined like that:

  $$[D_1 \to D_2] \triangleq \{f \in (D_1 \to D_2) \mid \text{f is continuous}\};$$

the second one gives the ordering relation and the third one the least element, in this case the completely undefined function. The choice of this is for correctly dealing with functions defined by recursion, at arbitrary functional type.

- **Full Cartesian Product** $(d_1 \times d_2)$ — It denotes the domain described by the following:

  $< (D_1 \times D_2); \forall a_1, a_2 \in D_1, b_1, b_2 \in D_2.(a_1, b_1) \sqsubseteq (a_2, b_2) \Leftrightarrow (a_1 \sqsubseteq a_2) \wedge (b_1 \sqsubseteq b_2); (\bot_{D_1}, \bot_{D_2}) >$.

  Differently of the smash product between flat domains, this (non- strict) cartesian product do not maintain the flatness of domains.

- **Discriminated Union** $(d_1 + d_2)$ — It denotes the domain described by the following:

  $< (D_1 + D_2); \forall a_1, a_2 \in (D_1 + D_2).(a_1 \sqsubseteq a_2) \Leftrightarrow (a_1 \sqsubseteq_{D_1} a_2) \vee (a_1 \sqsubseteq_{D_2} a_2); (\bot_{D_1} = \bot_{D_2} = \bot_{D_1 + D_2}) >$.

  Differently of the coalesced sum between flat domains, this discriminated union (also known as separated sum) do not maintain the flatness of domains.

- **Bekic Mappings** $(f \xrightarrow{B} d)$ — It denotes the domain described by the following:

  $< (F \xrightarrow{B} D); \forall m_1, m_2 \in (F \xrightarrow{B} D).m_1 \sqsubseteq m_2 \Leftrightarrow (m_1 = \lambda x \in F.\ \bot_D) \vee (\exists S \in \mathcal{F}(F \setminus \{\bot_F\}.\{m_1, m_2\} \subseteq (S \to D) \wedge (\forall x \in S.m_1(x) \sqsubseteq m_2(x))); \lambda x \in F.\ \bot_D >$.

  where F and D correspond to the carriers of f and d respectively and the Bekic Mapping carrier is defined as

  $$(F \xrightarrow{B} D) \triangleq \{\lambda x \in F.\ \bot_D\} \cup \bigcup\{(S \to D) \mid S \in \mathcal{F}(F \setminus \{\bot_F\}\}$$

  and $\setminus$ corresponds to the set minus operator. The least value in this domain is the function whose result is the least value from d everywhere.

Let both $\mathcal{D}, \mathcal{S}$ be arbitrary families of domains. Then, $\mathcal{S}$ is a Scott universe for $\mathcal{D}$ iff

$$\mathcal{D} \subseteq \mathcal{S}$$

$$\forall d_1, d_2 \in \mathcal{S}.[d_1 \to d_2] \in \mathcal{S}$$

$$\forall d_1, d_2 \in \mathcal{S}.(d_1 \times d_2) \in \mathcal{S}$$

$$\forall d_1, d_2 \in \mathcal{S}.(d_1 + d_2) \in \mathcal{S}$$

$$\forall f, d \in \mathcal{S}.(f \text{ is a flat domain}) \Rightarrow (f \xrightarrow{B} d) \in \mathcal{S}$$

Initially, one takes the subset closure of U. The subset-enriched universe Sub(U) is used to give the denotations for the full type language where invariants are taken into account. This has the consequence that an invariant applied to a recursively defined type also holds for any of its nested uses.

The next universe, FD(Sub(U)), maps every set (i.e. type) in Sub(U) into a flat Scott domain. This is a preparatory stage to constructing a particular Scott Universe, named SU, following along similar lines to the construction of U above.

So, let $\mathcal{D}$ stand for the family of flat domains given by FD(Sub(U)). The following domain operators are now defined, where $\mathcal{G}$ is an arbitrary family of domains:

$$\begin{aligned}
\mathrm{FUNCTION}(\mathcal{G}) &= \{[d_1 \rightarrow d_2] \mid d_1, d_2 \in \mathcal{G}\} \\
\mathrm{CARTESIAN}(\mathcal{G}) &= \{[d_1 \times d_2] \mid d_1, d_2 \in \mathcal{G}\} \\
\mathrm{DUNION}(\mathcal{G}) &= \{[d_1 + d_2] \mid d_1, d_2 \in \mathcal{G}\} \\
\mathrm{BEKICMAP}(\mathcal{G}) &= \{(f \xrightarrow{B} d) \mid f \in \mathcal{D} \wedge d \in \mathcal{G}\}
\end{aligned}$$

As before, a "tower" of families (of domains), $< D_i >_{i \in \omega}$, is constructed by primitive recursion on the index i:

$$\begin{aligned}
D_0 &= \mathcal{D} = \mathrm{FD}(Sub(U)) \\
\forall i \in \omega . D_{i+1} &= D_i \cup \mathrm{FUNCTION}(D_i) \cup \\
&\quad \mathrm{CARTESIAN}(D_i) \cup \mathrm{DUNION}(D_i) \cup \mathrm{BEKICMAP}(D_i)
\end{aligned}$$

Finally, define SU by the equation:

$$\mathrm{SU} = \left(\bigcup_{i \in \omega} D_i \right)$$

The closure operation OMEGA is not used here because it was defined for ensuring that recursive type definitions would always have a solution within the type universe U, what it is not the case here.

7 Applying the model to VDM

Initially it will be presented the type definiton syntax used in VDM. Next it will be shown how to construct the space of denotations for the type denotations and then the Scott universe following the previous framework. All the sets, operators, and notation previously developed will be used in this section. The presentation here is similar to the one followed by Martins and Moura (1988).

7.1 The VDM type definition sublanguage

In VDM, types are introduced through type definitions whose full syntactic form is described by the following fragment, using a BNF notation:

type-def	::=	simple-type-def |
		simple-type-def **where** **inv**-ty-name(args) $\triangleq$ e
simple-type-def	::=	type-name = type |
		type-name :: type
type	::=	basic-type | type-name | constructed-type
basic-type	::=	BOOL | NAT1 | NAT0 | INT | RAC | ATOM | TEXT
constructed-type	::=	{OBJECT_1, OBJECT_2,...,OBJECT_N} |
		seq of type |
		set of type |
		map type_1 **to** type_2 |
		opt type |
		(type_1 | ... | type_n |)
		record s-id_1:type_1,...,s- id_n:type_n **end**

The rule type-name :: type is used when one wants to have objects for which one can give a name and decompose their component parts, that is, the objects having explicit tag. This construction gives rise to the following semantic equation:

eval(type-name) :: eval(type)

where eval(type-name) stands for a domain of structured object described by eval(type). The VDM *mk*-notation is used to construct an element of this domain where eval(type-name) is to be considered the explicit tag of the resulting structure.

The rule type-name = type has the semantic meaning similar to the one described in the section about the type sublanguage definition.

7.2 The VDM type universes

To the basic syntactic subdomain correspond the basic semantic subdomain (family of sets), named Basic:

Basic = {<u>BOOL</u>, <u>NAT0</u>, <u>NAT1</u>, <u>INT</u>, <u>RAC</u>, <u>ATOM</u>, <u>TEXT</u>}

where

- <u>BOOL</u> = <u>bool</u> — consists of a set with two distinct values <u>TRUE</u> and <u>FALSE</u> corresponding to the true and false values of the propositional calculus;
- <u>NAT1</u> = {1,2,...,n} — correspond to a finite subset of the natural numbers without zero;
- <u>NAT0</u> = {0} ∪ <u>NAT1</u> — correspond to a finite subset of the natural numbers;
- <u>INT</u> = {...,-2,- 1,0,1,2,...,n} — correspond to a finite subset of the integer numbers;
- <u>RAC</u> correspond to a countable subset of the real numbers which include the rational ones.

- $\underline{ATOM} = \mathcal{F}(Atom)$ — correspond to a not empty, finite subset of values withdrawn of a denumerable set of primitive elements named Atom. Atom values are given as underlined uppercase words, for instance, $\underline{THIS\ IS\ AN\ ATOM}$.

- $\underline{TEXT}$ = Text — correspond to a denumerable set of elements which one can give a name. This set is completly different of the set Atom and also completly different of all other sets previously defined. Its elements are denoted by a sequence of letters between ' '.

One begin constructing the type universe UD intended to be the space for the type denotations, disregarding any invariants and other means of taking subsets.

One can define the following operators on sets and on families of arbitrary sets. Let be the elements $B, B_1, B_2 \in F$, F an arbitrary family of sets:

$$\begin{aligned}
seq(B) &= tag(\text{"seq"}, B^*) \\
set(B) &= tag(\text{"set"}, \mathcal{F}(B)) \\
opt(B) &= (B \cup \{\underline{NIL\ VALUE}\}) \\
record(B)(rt) &= tag(\text{"record"}, \mathcal{R}(B)(rt)) \\
SEQ(F) &= \{seq(B) \mid B \in F\} \\
SET(F) &= \{set(B) \mid B \in F\} \\
RECORD(F) &= \{record(B)(rt) \mid B \in F, rt \in \mathcal{M}(ATOM, F)\} \\
MAP(F) &= FUNC(F) \\
OPT(F) &= \{opt(B) \mid B \in F\} \\
UNION(F) &= COPROD(F) \\
PRODUCT(F) &= PROD(F) \\
VTAG(F) &= TAG(F)
\end{aligned}$$

The hierarchy of sets $< VU >_{i \in \omega}$ is defined as follows:

$$\begin{aligned}
VU_0 &= Basic \\
\forall i \in \omega . VU_{i+1} &= SEQ(VU_i) \cup SET(VU_i) \cup \\
&\quad RECORD(VU_i) \cup MAP(VU_i) \cup \\
&\quad OPT(VU_i) \cup UNION(VU_i) \cup \\
&\quad PRODUCT(VU_i) \cup VTAG(VU_i)
\end{aligned}$$

One can observe that one has a family of sets for each VU_i such that $VU_i \subseteq VU_{i+1}$, since $OPT(VU_i) = (VU_i \cup \{NIL\ VALUE\})$. Thus, the universal domain UD can be defined as follows:

$$UD = OMEGA\left(\bigcup_{i \in \omega} VU_i \right)$$

where OMEGA takes place to guarantee the closure of the universe with respect to the unions of the chains VU_i.

The subset-enriched universe Sub(UD) is taken to give the denotations for the VDM full type language where invariants are taken into account. FD(Sub(UD)), which maps every type in Sub(UD) into a flat domain, is a preparatory stage to constructing the Scott

Universe, named VUD. This universal domain contain all collections of values denoted by all constructions of the VDM metalanguage.

Using the operators FUNCTION, CARTESIAN, DUNION, and BEKICMAP previously defined, the following hierarchy of sets $< \mathrm{VSU}_i >_{i \in \omega}$ can be defined:

$$\begin{aligned} \mathrm{VSU}_0 &= \mathbf{FD}(\mathrm{Sub}(\mathrm{UD})) \\ \forall i \in \omega . \mathrm{VSU}_{i+1} &= \mathrm{VSU}_i \cup \mathrm{FUNCTION}(\mathrm{VSU}_i) \cup \\ & \quad \mathrm{CARTESIAN}(\mathrm{VSU}_i) \cup \mathrm{DUNION}(\mathrm{VSU}_i) \cup \mathrm{BEKICMAP}(\mathrm{VSU}_i) \end{aligned}$$

Finally, the Scott universe for the VDM metalanguage is defined as

$$\mathrm{VUD} = \left(\bigcup_{\in \omega} \mathrm{VSU}_i \right)$$

8 Conclusions

The denotational approach was used to constructively build up minimal, closed universes of denotations for all required values. Starting from a type definition sublanguage , it was shown how to get the denotations for the domain equations, including recursive ones. Next, it was shown how to build up a type universe containing all solutions for recursive type definitions and how to build up a Scott universe, which regards the concept of invariant and other means of taking subsets besides containing enough continuous functions, domain discriminated unions and domain products to give a type model rich enough to interpret the usual languages. Finally, that model was applied to the VDM metalanguage as an illustration.

Acknowledgements: I am indebted to Edward Hermann Haeusler and Valéria Paiva for useful suggestions. I am especially indebted to Prof. Raul Martins, who suggested the theme and provided guidance.

References

MONAHAN, B .Q. A type model for VDM. In: *VDM'87 – A formal method at work. Proc. of the VDM-Europe Symposium* , Bjorner, D. et al. Brussels, Belgium, March 1987. Also as *LNCS 252*. Berlin, Springer-Verlag, 1987.

TENNENT, R.D. **Principles of programming languages**. Englewood Cliffs, NJ, Prentice-hall, 1981.

ALLISON, L. **A practical introduction to denotational semantics**. Cambridge, UK, Cambridge Press, 1985.

MARTINS, R.C.B.; MOURA, A.V. **Desenvolvimento sistemático de programas corretos**. Campinas, VI Escola de Computação, 1988.

SCOTT, D. Domains for denotational semantics, **ICALP'82**, Aarhus, Denmark, July, 1982.

STOY, J.E. **Denotational semantics: the Scott- Strachey approach to programming language theory**. MIT Press, 1977.