Data Structuring via Functions

B H Venter
Department of Computer Science, University of Port Elizabeth
P O Box 1600, Port Elizabeth, 6000 South Africa
e-mail: csabhv@upe.ac.za

Abstract

A programming language data model is introduced, based on the notions that variables are functions and types are sets of values. It is shown that, despite the simplicity of the underlying mathematical formalism, the data model is very expressive, and does not violate basic intuitions about variables and types.
Keywords: *Data Structures, Functional Data Model, Sets, Lists, Records*
Computing Reviews Categories: *D.3.3, E.2.1*

Presented at Vth S A Computer Symposium

1 Introduction

Consider the following statements:

- A programming language type is only a set of values.
- A type constructor is an expression resulting in a set of values.
- Operators such as $+$ and $-$ are functions and do not form part of types.
- Expressions are function applications, and the type of an expression is the set of values representing the co-domain of the last function to be applied.
- A programming language variable is a functional mapping from the empty set to the set of values making up the type of the variable.
- An assignment statement partially redefines the rule according to which a variable maps the empty set to its type set.

From these statements, a formal model of the semantics of types, variables, assignments and expressions can be constructed. In the rest of this paper, this model will be referred to as the functional data model.

There are many other mathematical formalisms that can be used to model variables and types, but none are as simple and intuitive as the functional data model. Of course, since the functional model is so simple, it does not model all aspects of the type systems of typical programming languages. For example, the functional data model does not associate types with representations or with sets of operators. As a consequence, language theorists tend to explore more complicated formalisms.

Some of the more complicated formalisms used to model old programming languages are in turn used as the basis for designing new programming languages, which tend to be more consistent and complete than designs not based on some formalism. The lambda calculus has been a popular choice.

Given the simplicity and intuitive appeal of the functional data model, it is worth asking whether the functional model can profitably be used as the basis for designing a programming language which is consistent, complete and expressive. Earlier work by Shipman[6] and Buneman[1] already goes a long way towards proving that this can be done. However, the functional data model has not yet been used as a basis for designing an imperative language in the Algol tradition. The author is currently designing such a language, and the type system of the preliminary version of this language is presented in this paper.

2 Types as sets

According to the functional data model a variable such as

$$\mathrm{F} : \mathrm{Integer} := 1;$$

should be viewed as syntactic sugar for the function

$$f : \{\} \rightarrow \mathrm{Integer};\ f() = 1$$

The type, Integer, is thus simply a set of values representing the co-domain of f. Therefore, in the case of

$$\mathrm{G} : \textbf{set of}\ \mathrm{Integer} := \{1,\ 2,\ 3\};$$

one has to view **set of** as an operator which results in the powerset of its argument. Hence, the above is syntactic sugar for

$$g : \{\} \rightarrow \{s | s \subseteq \mathrm{Integer}\};\ g() = \{1,\ 2,\ 3\}$$

A consequence of this view of types is that types A and B where

$$\mathrm{A} = \textbf{set of}\ \mathrm{Integer};\ \mathrm{B} = \textbf{set of}\ \mathrm{Integer}$$

represent the same set of values and are thus identical.

Hence, according to the functional data model, types are simply sets of values which need not be disjoint. This view of types is in marked contrast with the long standing "axioms" of language designers, namely:

- A value belongs to one and only one type[2].
- Types are not sets[4].

The full consequences of abandoning these "axioms" are examined by the author in another paper[7]. The main conclusions drawn there are that a *consistent* language design based on the view that types are sets that need not be disjoint:

- must use structural type equivalence rules
- cannot be completely type checked at compile time
- must separate operators from types
- must treat types as first class values
- must be fully polymorphic
- must allow types to vary at run-time
- must support abstract data types in a non-traditional way

This paper argues that the following syntax for declarations would suffice to declare types, constants and variables in a programming language design based on the functional data model.

constant-definition =
  identifier '=' expression
variable-definition =
  identifier ':' expression [':=' expression]
expression =
  expression {('∪' | '∩' | '−') expression} |
  (**set** | **sets**) **of** expression |
  (**list** | **lists**) **of** expression |
  (**map** | **maps**) **from** expression
    {',' expression} **to** expression |
  set-value | non-set-expression

Note that no special syntax is needed to declare types. Where a type is normally expected, any expression resulting in a set of values will do. Type names can be introduced by declaring named constants with sets as values.

Naturally, some types will have to be predefined. The following seems the most widely expected:

{True, False} = Boolean ⊂ Entity
Integer ⊂ Number ⊂ Entity
Character ⊂ String ⊂ Entity

where Entity denotes the universal set (the set of all possible values) and

Boolean ∩ Number = Boolean ∩ String =
Number ∩ String = {}

3 Set values

Most languages supporting sets allow set values to be denoted by explicit enumerations such as {2, 3, 5, 7}.

Very few languages, however, allow a version of the most general notation for sets: $\{x \mid \phi(x)\}$ where $\phi(x)$ is some statement about x. Since the general notation is very expressive, the expressive power of a language will be greatly enhanced if it provides a version of this notation.

When devising a version of the general notation that is suitable for a programming language, the first thing to note is that it is not practical to always interpret x as potentially being *any* member of the universal domain of discourse. Thus, the programmer should be required to explicitly indicate the set of values that x could possibly be. For example:

$\{x \in \{2 \ldots 9\} \mid \text{Is-a-prime}(x)\}$

It is also useful to allow x to be an expression, possibly containing more than one local variable. For example

{a+b, a ∈ {'a','b'}, b ∈ {'1','2'}}

which is equivalent to

{'a1', 'a2', 'b1', 'b2'}

It is not difficult to generate the potential members of sets specified in the general format. Nor is it difficult to decide if a potential member qualifies for membership. However, a simplistic compiler faced with {x ∈ Integer | Is-a-prime(x) **and** x < 10}

is likely to generate code that will consider *all* integers from min(Integer) to max(Integer). To do otherwise, the compiler will have to analyze the selection condition. Such an analysis is not trivial for *arbitrary* conditions. Moreover, in the example, Is-a-prime may well have been defined in such a way that that even a sophisticated analysis algorithm will fail to determine that it will return False for every argument value less than 2.

The potential generation of atrocious code for examples such as the one above is probably the reason why so few languages allow set values to be specified using the $\{x \mid \phi(x)\}$ notation. However, *potential* inefficiency is not a good enough reason to eliminate an otherwise useful language construct. After all, even a simplistic compiler is likely to produce acceptable code for

{x ∈ {2...9} | Is-a-prime(x)}

Moreover, in a case where the set cannot be as easily specified by an explicit enumeration of values, for example

{x ∈ {1...1000} | A-complicated-rule(x)}

an optimizing compiler may well compute the members of this set during compilation, whereas a programmer denied the use of the general notation would probably have used a run-time routine to compute the set, which would be much more difficult to optimize away.

The following syntax is proposed for set-values:

set-value =
  '{' {expression ['...' expression] [',']} '}' |
  '{' **all** [expression] [**where**]
    {identifier '∈' expression}
    ['|' expression] '}'

Note that the noise words **all** and **where** are needed to make it clear that

{**all** a+b **where** a ∈ {'a','b'}, b ∈ {'1','2'}}

is not a set of three elements, of which the last two are boolean values. Such an interpretation is possible when given only

{a+b, a ∈ {'a','b'}, b ∈ {'1','2'}}

4 Lists

The expression

list of expression

is simply syntactic sugar for

maps from {1...max(Integer)} **to** expression

with the understanding that the list

['a', 'a']

represents the mapping

(1, 'a')
(2, 'a')
(3, Undefined)
$\vdots$
(max(Integer), Undefined)

where Undefined is a predefined value not present in any predefined type but Entity.

The above definition allows all the normal list operations to act in the expected way, provided that the length of a list L is defined to be

$$\max\{\textbf{all}\ \mathrm{n} \in \mathrm{Integer} \mid \mathrm{L(n)} \neq \mathrm{Undefined}\}$$

The same syntax and conventions used to define set values can be used to define list values (but using [] instead of {}).

5 Strings

The predefined set String can be regarded as equivalent to

lists of Character

and the predefined set Character can be regarded as equivalent to

$$\{\textbf{all}\ \mathrm{S} \in \mathrm{String} \mid \mathrm{length(S)} = 1\}$$

This is, of course, a circular definition. However, it doesn't matter, so long as it is not used to formalize the semantics of a language. To do the latter, one would have to specify Character as an enumeration, typically the ASCII character set.

The circular definition does have the advantage of making it clear that character 'a' and string 'a' denote the *same* value. This seems rather more convenient than continually having to convert characters into strings and strings into characters, even if this is only done conceptually. It also has the advantage that 'a' = ['a'], 'ab' = ['a', 'b'], and so on.

6 Arrays

The array construct of classical languages is simply a special case of the **map** type constructor provided by the proposed syntax. Thus, the classical declaration

A : **array** 1...10 **of** Integer;

would be written as

A : **map from** {1...10} **to** Integer

and should be viewed as syntactic sugar for the function

$$a : \{\} \rightarrow \{f | f : \{1 \ldots 10\} \rightarrow \mathrm{Integer}\}$$

Furthermore, A should be understood as $a()$ and A[1] as $(a())(1)$. Finally, assignments to elements of A should be viewed as *incremental* redefinitions of a.

Note, however, that the construction of generalized array operators, makes it necessary to specify that type expressions may contain free variables in certain circumstances. The reason for this is that types Vector10 and Vector20, where

Vector10 = **maps from** {0...9} **to** Integer
Vector20 = **maps from** {0...19} **to** Integer

represent *disjoint* sets of values, and hence operator Sum20, where

$$\text{Sum20} : \text{Vector20} \rightarrow \text{Integer}$$

may not be applied to any array of type Vector10.

Using free variables instead, one may have

$$\text{Vector} = \textbf{maps from } \{\text{L} \ldots \text{U}\} \textbf{ to } \text{Integer}$$
$$\text{Sum} : \text{Vector} \rightarrow \text{Integer}$$

where type Vector is understood to be the set

$$\bigcup_{\substack{\text{L, U} \in \text{Integer} \\ \text{L} \leq \text{U}}} \{\text{f} \mid \text{f} : \{\text{L}, \ldots, \text{U}\} \rightarrow \text{Integer}\}$$

The above definition would allow Sum to be applied to any map from a single domain, consisting of a contiguous range of integers, to Integer. Furthermore, by specifying that L and U are *parameters* of Sum, and that L and U must automatically be set to integer values representing the lower bound and upper bound of the first parameter, it becomes possible for Sum to determine the index range of an actual parameter.

7 Records

Most modern programming languages provide some form of record constructor, and most implementations of abstract data types make very heavy use of records. More recently, record-based type extensions have been used to integrate object-oriented programming ideas into a language of the Algol family[8].

The reader may therefore have been surprised by the omission from this proposal of a type constructor for records. The reason for this is that such constructors are very difficult to model as operators resulting in sets of values.

Consider the following (Ada-like) type definition:

```
type Stack is
  record
    Top : range 0..100;
    Elems : array 1..100 of Integer;
  end record;
```

and the following interpretation of Stack as a set of values:

$$\text{Stack} = \{f \mid f : \{\text{Top, Elems}\} \rightarrow \text{R} \cup \text{E}\}$$
$$\text{R} = \{0, \ldots, 100\}$$
$$\text{E} = \{g \mid g : \{1, \ldots, 100\} \rightarrow \text{Integer}\}$$

Provided that one views a reference to S.Top, where S is a variable of type Stack, as syntactic sugar for $(s())(\text{Top})$, and an assignment to S.Top as a partial redefinition of the function returned by s, this interpretation of Stack makes it possible to model S as a function and Stack as a set of values.

Note, however, that the constraint that the function returned by s may only return a value between 0 and 100 when applied to Top, is not captured by the model. Another problem is that there is no explicit connection between type Stack and the names Top and Elems.

The latter becomes a significant problem when deciding whether or not type Stack1, below, represents the same set of values as type Stack.

```
type Stack1 is
  record
    Top : range 0..100;
    Elems : array 1..100 of Integer;
  end record;
```

If the intention is to use Stack as an *abstract* data type, then the irrelevant detail that Stack values are records must be hidden from the client code. Specifically, the client code must not be able to do something like this:

```
S : Stack;
...
put(S.Elems[S.Top−1]);
put(S.Elems[S.Top]);
S.Top := S.Top − 2;
```

In conventional languages like Ada this is achieved by hiding the component selector operators, .Top and .Elems, from the client code. Correspondingly, a language based on the functional data model could provide a means of exporting the name, Stack, without also exporting the names Top and Elems.

However, what happens if a client module includes a definition of type Stack1? If type Stack and Stack1 represent the same set of values, the names Top and Elems are not hidden from the client, since they have to be available for use with variables of type Stack1.

Thus to support both records and information hiding in a language based on the functional data model, it is necessary to specify that different applications of the same record constructor yield *different* sets of values. This is much like saying that 5+6 should yield a different value every time it is evaluated.

8 Living without records

Records are not only difficult to marry with the functional data model. They cause problems even in conventional languages. For example, it is difficult to use records to define an abstract data type Person, as well as subtypes Student and Staff, so that some values can belong to all three sets while certain operations are restricted to values of type Student and others to values of type Staff. Wirth's type extensions[8] come close to allowing this, but fall short because he requires an extended type to be disjoint from the type it is extending. Thus, a record value of type Student, defined as an extension of Person, may be assigned to a variable of type Person, but only with loss of information. Furthermore, a value of type Student cannot *also* be a value of type Staff.

Problems such as the above also occur when record types are used to model persistent information (databases), and have been discussed at length in [3]. Various solutions have been proposed, including some which are still record based (for example, semantic data models[5]). The "record free" functional data model introduced by Shipman[6], however, offers a far simpler solution.

In short, when using the functional data model, one declares *groups* of functions in the place of records. For example, instead of declaring:

```
type Person is
  record
    Name : String;
    Date-of-birth : Integer;
  end;
P : Person
```

one declares a group of functions:

Name : Person $\rightarrow$ String
Date-of-birth : Person $\rightarrow$ Integer
P : {} $\rightarrow$ Person

and declares Person to be a set of abstract values, for which no operators exist, apart from explicitly defined functions. Information hiding is achieved by exporting only those functions that are needed by clients.

Note furthermore that it is possible for a client module to define types Staff and Student to be subsets of type Person, and to define additional functions:

Student-number : Student $\rightarrow$ Integer
Staff-number : Staff $\rightarrow$ Integer

Furthermore, if PS denotes a value which is a member of type Staff *and* type Student, then *all* of the following function calls are valid:

Name(PS)
Date-of-birth(PS)
Student-number(PS)
Staff-number(PS)

It is very cumbersome to achieve the same functionality using a record-based data model.

9 Pointers

Another type constructor that is very heavily used in implementing abstract data types, but absent from this proposal, is **pointer to** ...

It is not difficult to view a pointer variable as a function. For example:

P : **pointer to** Person;

can be viewed as a function of the following type:

P : {} $\rightarrow$ {f | f : {} $\rightarrow$ Person}

However, pointers are error-prone and troublesome. Moreover, a language supporting sets, lists, generalized functional mappings and abstract values, has no need of pointers.

10 Dynamic Types

The author has concluded elsewhere[7] that one of the consequences of adopting the functional data model is that a consistent language design must allow types to vary at run-time. In other words, the language must allow the set of values making up a type to be computed (and *re*computed, if necessary) at run-time. Such types are called dynamic types, since they represent dynamically varying sets of values.

Dynamic types make good sense in database-oriented languages. For example, if one declares

Parts = {1, ..., 1000}
Suppliers = {1, ..., 200}
P : **set of** Parts
S : **set of** Suppliers

and one *may* also declare

Supplies : **map from** S **to set of** P

instead of

Supplies : **map from** Suppliers **to set of** Parts

then the constraints that Supplies may only return sets of values that are in P, and may only be applied to values that are in S, are expressed in their most natural form.

Note that no special notation is needed to allow dynamic types. Any expression which can only be evaluated at run-time, and is used in the role of type, represents a dynamic type.

11 Abstract values

The astute reader will have noticed by now that, although the existence of sets of "abstract" values have been referred to, no syntax for generating such sets have been presented. The reason is that dynamic types make it unnecessary to have such a syntax.

For example, instead of declaring Parts to be a large set of values, and P to be a variable of type **set of** Parts, as is done above, one would declare P to be a variable of type **set of** Entity, and Parts to be a constant function that returns the current value of P. Initially, P would have no elements: As new parts are added to the database, new abstract values (elements of Entity that are not yet included in any set but Entity) are added to P.

To support this, the language only has to supply a function that delivers a different, unused element of Entity every time it is called.

12 Conclusion

Not much needs to be said to define the functional data model: Variables are functions whose definitions may be changed by assignment statements. Types are sets of values that act as membership constraints on the domains and ranges of functions. *That's it.*

Despite the almost ridiculous simplicity of its definition, the functional data model provides as much security and power of expression as any other data model. The same can be expected of a programming language based on this data model.

At this stage it cannot be stated with absolute certainty that such a language would be efficient or even implementable. However, there are many good techniques for implementing sets and functions mapping sets to sets. It therefore seems likely that an efficient programming language, based on the functional data model, can be designed and implemented.

References

[1] O. P. Buneman, R. E. Frankel, and R. Nikhil. An implementation technique for database query languages. *ACM Trans. Database Syst.*, 7(2):164–186, June 1982.

[2] O. J. Dahl, C. A. R. Hoare, and E. W. Dijkstra. *Structured Programming.* Academic Press, 1972.

[3] W. Kent. Limitations of record-based information models. *ACM Trans. Database Syst.*, 4(1):107–131, March 1979.

[4] J. Morris. Types are not sets. In *ACM Symposium on Principles of Programming Languages*, pages 120–124, New York, 1973. ACM.

[5] J. Peckham and F. Maryanski. Semantic data models. *ACM Comput. Surv.*, 20(3):153–189, September 1988.

[6] D. Shipman. The functional data model and the data language DAPLEX. *ACM Trans. Database Syst.*, 6(1):149–173, March 1981.

[7] B. H. Venter. A value can belong to many types. In *Proceedings of the 6th Southern African Computer Symposium*, pages 10–15, 1991.

[8] N. Wirth. Type extensions. *ACM Trans. Prog. Lang. Syst.*, 10(2):204–214, April 1988.