A DOMAIN FOR FUNCTIONAL PROGRAMMING SYSTEMS

by

Johannes J. Martin

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia

March 18, 1981

Technical report No. CS81004

KEYWORDS

Programming languages, functional programming languages, FP-systems, domains, nondeterminism.

CR Categories: 4.22, 5.24

ABSTRACT

A domain for functional programming systems is proposed. This domain is the powerset of a set of items where items are either atomic or ordered pairs of items. The structure of the domain is determined by the relation 'is weaker than' and by three basic operations, under which the domain is closed: Union, the cartesian product, and the operation of application.

This domain has several favorable properties.

1. The domain not only contains data objects and programs (that is descriptions of computations) but also functions (that is sets of ordered pairs of items) as members. In fact, it turns out that all objects and mappings needed for a programming language (including the program forming operations) are members of the domain.

2. The bottom symbol, which represents the 'undefined' condition, is the empty set PHI. This leads to a rather natural intuitive model of computation.

3. Functions are almost strict in the following sense. Since the members of the domain are sets, functions map sets to sets. All functions and functionals are of the form

   $$f \cup \{c(1),\ldots,c(n)\}$$

   where f is strict and the c(i) are constant.

4. Nondeterministic processes are naturally accommodated by the domain.

5. Without introducing special cases, one can define application in such a way that the 'undefined' condition is always handled correctly. (i) It is impossible to construct a test for the bottom symbol PHI. (ii) Expressions such as 'PHI v true' or 'PHI * Ø' are evaluated to their proper values, namely 'true' and 'Ø', respectively.

6. Though not treated in this paper, data types can be added in a straightforward way.

1. INTRODUCTION

Strachy observed [Strachy73] that a considerable amount is known about a programming language once its value domain is defined. In [Backus79] Backus goes even further and asserts that the very concept of what a program is, "is largely determined by the choice of the 'program domain' that 'programs' map into itself." If the domain were only an amorphous set of objects that programs manipulate, then it would hardly have this significance. But the domain must be viewed as an algebraic structure. This structure consists of the set of computational objects, relations on this set, and basic operations under which the set is closed.

Besides data objects such as numbers and characters, the domain should contain all programs so that compilers, verifiers, and optimizers can be written. Moreover, for reasons that will become clearer as we proceed, all desirable functions should be members of the domain.

> NOTE: A function f: A -> B is a special type of subset of A x B, whereas a program is the description of the computation of a function. Since there may be many computations that yield the same function, a function is a more abstract object than a program.

Among the desirable functions, there are those that take functions as parameters and compute functions as results. These are the operations needed for the composition of programs (we refer to them as functionals or functional forms [Backus78,79]). The question arises whether one can indeed combine all these seemingly rather different objects into one domain without loosing simplicity and elegance.

That this is possible has been demonstrated in [Scott76] and [Plotkin76]. Here we apply some of these results to FP-type languages [Backus78]. The fact that FP-functions have exactly one parameter makes it necessary to consider the empty set PHI an atomic object, for otherwise the set of permissable functions seems to be needlessly restricted.

The proposed domain is the powerset of the set of all items, where items are atomic (including the empty set PHI) or ordered pairs of items.

This domain is partially ordered and it is closed under union and the cartesian product. The third basic operation needed is the application of a function to an object.

In the following we deal first with the domains of LISP and FP-systems, then we describe some properties of the domain suggested. Thereafter we show that all interesting functions and functionals are members of our domain and that nondeterministic programs can be specified in a natural way. Finally we show how values of expressions such as 'true v undefined' can be given their natural interpretation within the formalism of the system.

Most of the set terminology used is taken from [Hayden68]. The symbol '==' denotes equality by definition; sometimes informal comments are offset from formulas by '//'.

2. THE DOMAINS D1, D2, AND D3

We start with the definitions of three domains. By 'A' we denote a countable set of atoms. These atoms may, for example, be objects such as numbers, character strings, T and F for 'true' and 'false', the null element 'nil' and other items that we do not care to analyze further. The domains are defined as follows.

DEFINITION 2.1: D1: 1. All elements of A including the null element 'nil' are in D1.

2. If x and y are in D1, then so is the ordered pair (x,y).

DEFINITION 2.2: D2: 1. All elements of A including the null element 'nil' are in D2.

2. If x1,x2,...,xn are in D2,
then so is the n-tuple (x1,x2,...,xn).

DEFINITION 2.3: D3: D3 = P(D1), the powerset of D1.

Note that D1 contains D1 x D1 as a subset. In terms of LISP, D1 is the set of all S-expressions. D2 has been used by Backus [Backus78] for his FP-systems.

In our discussion, we first assume that the set of atoms 'A' does not contain PHI, the empty set, as a member, later we consider the consequences of permitting PHI as a member of 'A'.

D1 and D2 are equivalent

The domains D1 and D2 are equivalent in the sense that each one can be used to model the other. In order to model D2 by D1 we may define

DEFINITION 2.4: The null element 'nil' is the Ø-tuple. An n-tuple is an ordered pair (x,z) where x is any element in D1 and z is an (n-1)-tuple.

In order to model D1 by D2, we represent an ordered pair simply by a 2-tuple.

Nevertheless, practical considerations may induce one to prefer one domain over the other. The discussion of such considerations is not in the scope of this paper.

In this paper we use D1 knowing that we may use sequences if we need to. By adding the bottom symbol BOT where $BOT \sqsubseteq x$, and $x \sqsubseteq x$ for all x in D1, D1 becomes a partially ordered set and thus gives rise to the flat lattice LD1. Now partial functions on D1 are defined to have the value BOT where

they do not have a value in D1 and hence may be considered total on LD1.

NOTE: The symbol '$\subseteq$' is read 'is weaker than'.

We will not go into any further detail of this matter but refer the reader to [Manna73], [Scott76], [Scott77], and [Stoy77].

By means of suitable notational conventions, elements of LD1 can be made to represent programs on LD1. Backus [Backus78] introduces a very elegant mechanism, called metacomposition, and shows by way of example that it permits one to define recursive functions without resorting to the usual recursive definitions. Williams [Williams80] describes a general algorithm that permits one to express -- by metacomposition -- functions given by mutually recursive equations.

The domain D1 does not contain functions

The elements of D1, which are atoms and pairs (or sequences) of other elements of D1, are single objects rather than sets. Hence D1 does not naturally contain functions on D1 as elements. Identifying the members of D1 with the members of D1 -> D1 does not work since the two set have different cardinalities.

3. SOME PROPERTIES OF THE DOMAIN D3

The domain D3 is the powerset of D1. The elements of D3 are sets (subsets of D1); D3, partially ordered by the subset relation, is a lattice. We read this relation again as 'is weaker than'. The bottom of the lattice is PHI, the empty set, the top is D1. D3 is closed under set union. Since D1 contains D1 x D1, D3 is also closed under the cartesian product. We assume that both operations are part of any programming system based on D3.

Relations on D1 induce functions on D3

Relations R on D1 are subsets of D1 x D1. Therefore they are subsets of D1 and thus members of D3. Each such relation gives rise to a (total) function on D3. Let X be some member of D3, then the application of R to X is defined by

DEFINITION 3.1: Application

$$R(X) = \{y \mid (y,x) \text{ in } R \text{ for some } x \text{ in } X\}.$$

NOTE: The usual notation for application (f(x)) and composition (f.g) implies that evaluation proceeds from right to left. Therefore, a function is treated as a set of pairs taken from the set (range x domain) and not from the set (domain x range).

Besides union and the cartesian product, application constitutes the third and last basic operation on D3. From the definition of application, five equivalences follow:

(3.1) $(F \cup G)(X) = F(X) \cup G(X)$

(3.2) $F(X \cup Y) = F(X) \cup F(Y)$

(3.3) $F(PHI) = PHI$

(3.4) $PHI(X) = PHI$

(3.5) If X, Y, and Z are singletons

then $(Y \times X)(Z)$ = if X=Z then Y else PHI.

Furthermore,

R(X) includes R(Y) if X includes Y and

for a given directed set LX, we obtain for the least upper bound (l.u.b.)

$$R(\text{l.u.b}(LX)) = \text{l.u.b.}(\{R(X) \mid X \text{ in } LX\})$$

Hence, functions induced by relations R are both monotonic and continuous.

PHI denotes "undefined"

It also follows from the above that a partial function on Dl (that is a special kind of a relation) that is not defined for some value x in Dl, gives rise to a function on D3 that assumes the value PHI for {x}. There-

fore, with functions on D3 that are induced by relations on D1, PHI is for D3 what BOT is for LD1.

The set of pairs that represents a relation that gives rise to a function F on D3 is called the representative set of F. F itself is said to be in D3 (as well as on D3). Deviating from [Scott76], we do not identify elements of D1 with the finite elements of D3. This keeps the functions in D3 particularly simple as we shall see in the next subsection.

Having the empty set represent the condition 'undefined' leads to a quite natural intuitive model. A process that computes a function produces a set of results. While the process is running, the set is empty. Upon termination, the process adds one or more results to the set. Thus, if the process never produces a result, then the set remains empty.

Not all functions are in D3

One might conjecture that all functions on D3 are also in D3. Although this is correct, as we shall see, for all necessary functions and functionals, below we give an example which shows that in general the conjecture is false. However, the following theorem gives us a criterion to determine whether or not a given function is in fact in D3.

THEOREM 3.1:

Assume that (1) F: D3 -> D3 is a function on D3 and

(2) F distributes over union, that is,

(For all X,Y in D3) $F(X \cup Y) = F(X) \cup F(Y)$.

Then there is a relation R on D1 such that for all X in D3

$$F(X) = \{y \mid yRx \text{ for some } x \text{ in } X\}$$

Proof:

Let aRb iff (a in F({b}))

$$X = \bigcup_{x \text{ in } X} \{x\}$$

$$\rightarrow F(X) = \bigcup_{x \text{ in } X} F(\{x\}) \qquad \text{// by (2) //}$$

$$\rightarrow F(X) = \{y \mid y \text{ in } F(\{x\}) \text{ for some } x \text{ in } X\}$$

$$\Leftrightarrow F(X) = \{y \mid yRx \text{ for some } x \text{ in } X\} \quad \text{Q.E.D.}$$

4. FUNCTIONAL FORMS

Functional forms or functionals, which are a special kind of function, map functions and/or other members of D3 to functions. For functions that are members of D3, a functional is a mapping of the form

FF: D3 x (D3 x (...)) -> D3,

Since functions in D3 have the form F:D3 -> D3, most functional forms (those with more than one parameter) are apparently not in D3. The trouble is superficial because any n-tuple of members of D3 can be packaged into D3 by means of the cartesian product. For example, the functional 'composition' computes a new function from two given ones. Using functional (rather than infix) notation, we express this by

h = comp(f,g).

Since pairs of sets are not members of D3, a pair of functions such as (f,g) is not in D3. However, the cartesian product (f x g) is. Therefore, the modified composition function

h = comp(f x g)

is indeed a mapping from D3 to D3 and may thus be in D3.

By virtue of this packaging step, functionals become ordinary functions on D3 except that some of them may not themselves be members of D3. We may

use our criterion to decide this question. By theorem 3.1, we know that a function is in D3 if it distributes over union.

Using this result, we find that the functionals composition, construction, if-then, and constant are in D3. Rather than proving this by applying theorem 3.1 (which is straightforward), we give two definitions for each functional:

(i) the interaction of the functional with the application operator and

(ii) the representative set of the functional.

Both definitions have the same effect, but the second type can only be given for members of D3.

A. Composition

(4.1) (i) (comp(F x G))(X) == F(G(X))

(ii) comp == {((z,x), ((z,y),(y,x))) | x,y,z in Dl}

B. Construction

(4.2) (i) (cnstr(F x G))(X) == F(X) x G(X)

(ii) cnstr == {(((z,y),x), ((z,x),(y,x))) | x,y,z in Dl}

C. Condition

(4.3) (i) (if (P x F))(X) == if P(X) then F(X) else PHI

(ii) if == {((y,x), ((T,x),(y,x))) | x,y in D1}

NOTE: We have chosen the if-then conditional rather than the usual if-then-else form because the former is strict while the later is not. A function is strict if it maps PHI to PHI. If f = PHI is a function, then f(x) is PHI for all x. Thus, a strict functional maps the function that is always PHI to itself. Functions specified by the if-then-else construct can be defined with the above if-then construct by

if P then F else G == if(P x F) U if(~P x G)

D. Constant

(4.4) (i) (const(Y))(X) == Y

(ii) {((y,x), y) | x,y in D1}

Notational conventions and primitive functions

In order to improve the readability of the examples below, we write

	f.g	instead of	comp (f x g),
	[f,g]	"	cnstr(f x g),
	if p then f	"	if(p x f),
and	if p then f else g	"	if(p x f) U if(~p x g).

Further, we use infix notation for the usual arithmetic and logical operations by defining

f op g == op.[f, g]

and apply the usual precedence rules.

In the examples below, the following functions are assumed to be given as primitives. The definitions are subject to the distributive rule 3.2

F(X U Y) = F(X) U F(Y).

For all x in D1:

id({x}) == {x} // the identity function //

hd({x}) == if (x = (a,b)) then {a} else PHI // head //

tl({x}) == if (x = (a,b)) then {b} else PHI // tail //.

The functional "while" is not in D3

Next we give the promised example of a function (functional) not in D3.

while (P x F) == if ~P then id

U if P then (while(P x F)).F

If 'while' were in D3, then it should be true that

while (P x (F U G)) = while (P x F) U while (P x G)

The following counter example shows that this is not always so. Suppose F={(2,1)}, G={(3,2)}, and P={(true,1),(true,2),(false,3)}.

Then

while(P x (F U G))({1}) = {3}

whereas

(while(P x F) U while(P x G))({1})

= while(P x F)({1}) U while(P x G)({1}) = PHI

Thus, 'while' is a function on D3 that is not in D3.

There are three methods to specify a function

Functions may be specified

1. directly by enumerating their representative sets, for example

xor == {(F,(T,T)), (T,(T,F)), (T,(F,T)), (F,(F,F))},

2. by combining primitive functions (given a priori) and/or otherwise specified functions by means of functional forms, for example

abs == if id > const(Ø) then id else -.id,

3. by recursion, that is, by the least fixed point of a functional, for example

* == if tl=const(Ø) then const(Ø)
else hd + *.[hd, (tl-const(1))].

Strict linear functionals are in D3

In [Backus79] Backus introduces the concept of linear functional forms. These are defined as follows.

The functional E is linear if there is a functional El such that for all functions p, f, and g:

E(if p then f else g) = if El(p) then E(f) else E(g)

The following relationship exists between linear forms and functionals in D3.

Strict linear functionals are in D3.

The proof of this claim is given in [Martin81].

Functionals in D3 are the only ones needed

Functionals such as 'while' and some others are not linear nor are they in D3. However, the functions they compute are in D3 since they can be expressed as fixed points of continuous functionals that are in D3. For example, the function while(p x f) is the least fixed point of the functional t(w) defined by

t(w) == (if p then w.f) U (if ~p then id)

Therefore, the ususal constructs for 'while' etc. can be interpreted as simple macro mechanisms that generate the required recursive definition for the function denoted by the construct.

Nondeterministic processes can be specified

Suppose a value y is specified by

$$y = (f(1) \cup f(2))(\{x\})$$

where one of the functions f1 or f2 may not terminate, that is yield PHI. Here y assumes a non-PHI value if at least one of the functions yields a non-PHI value. Thus y is specified nondeterministically. More complex nondeterminisms may be specified by larger expressions and by the involvement of recursive definitions.

NOTE: For a successful implementation of such a device on a one-processor computer system, it is necessary to let alternate processes take turns of finite duration, for otherwise a successfull process may never have a chance to run.

5. D3 WITH THE ADDITIONAL ATOM PHI

The pairs in D1 contain as components only elements of D1; hence they never contain PHI. If we add PHI to the set of atoms, then pairs of the form (PHI,x) or (y,PHI) are possible.

Now the relation 'is weaker than' must be redefined. First, since PHI, the bottom of D3, is weaker than all other objects, D1 is now partially ordered and by itself a lattice. Secondly, it is desirable that this lattice is not flat but that ordered pairs with components that are PHI are weaker than those without. For example,

$$(\text{for all } a,b)\ (a, \text{PHI}) \sqsubseteq (a, b).$$

Before, 'weaker' was synonymous with 'subset of' defined by

$X \subseteq Y$ iff (for all x in X there is a y in Y) such that x=y.

Now we have

DEFINITION 5.1: "is weaker than"

1. (for all x in D1) $PHI \subseteq x$
2. (for all x in D1) $x \subseteq x$
3. $(u, v) \subseteq (x, y)$ iff $(u \subseteq x)$ & $(v \subseteq y)$
4. (for all X,Y in D3)
   $X \subseteq Y$ iff (for all x in X there is a y in Y) such that $x \subseteq y$.

The atom PHI adds the following improvements to the system. Consider the function 'or', which may be defined by

or == {(T,(T,T), (T,(T,F)), (T,(F,T)), (F,(F,F))}

and consider the test

if (f=const(∅)) or (a/f=const(1)) then ...

If we assume that f(x)=∅ implies a/f(x)=PHI and consequently (a/f(x)=const(1))=PHI, then f(x)=∅ leads to

if (T or PHI) then ...

The value of 'T or PHI' is undefined according to the definition given for 'or', however, logically we would expect the value 'T' since, as soon as one operand of 'or' has the value 'T', the value of the other one does not matter anymore; the result is 'true' anyway. A simple modification of the application operator accommodates cases of this kind.

DEFINITION 5.2: Application

Let F and X be elements of D3 and x, y, and z elements of D1, then

$$F(X) = \{y \mid (y,z) \text{ in } F \ \& \ z \sqsubseteq x \text{ for some } x \text{ in } X\}.$$

To continue our example, we redefine 'or' as follows

or == {(T,(T,PHI)), (T,(PHI,T)), (F,(F,F))}.

Now 'or' applied to (T,T), (T,F), (T,PHI), (F,T), or (PHI,T) yields 'T',
applied to (F,F) it yields F, and
applied to (F,PHI) or (PHI,F) it yields PHI.

Other logical functions can be specified in a similar fashion. Also consider the constant function (some_constant, PHI) and the functional

const == {((y,PHI), y) | y in Dl}. Further, the projections 'hd' and 'tl', if applied to an ordered pair of the form (x, PHI) or (PHI, x) give

hd(x, PHI) = x,

hd(PHI, x) = PHI,

tl(x, PHI) = PHI,

tl(PHI, x) = x.

With this, we can define, for example, multiplication by

* == {(Ø,(Ø,PHI)), (Ø,(PHI,Ø))}
U (if tl > const(Ø) then hd + *.[hd, tl-const(1)])

where '+' and '-' are assumed to compute PHI if one of their operands equals PHI.

The fact that the definition of the application operator has been modified does not alter the basic equivalences for 'application' given by 3.1 - 3.5 except for 3.3. F(PHI) = PHI is no longer true, that is, functions are not anymore necessarily strict. This is so because the representative set of a function may now contain elements of the form (y,PHI) yielding y for all values to which the function may be applied.

Now we may assert that all functions have the form

f U {c(1),...,c(n)}

where f is strict and the c(i) are constant. The other properties of domain D3 are unaffected. However, the new definition of application makes it impossible to define a function that could test a given value for equality with PHI although PHI, which is now an atom, may freely be used in expressions. Suppose an attempt is made to construct a predicate 'eq' that tests whether an item equals PHI by including the pair (T,(PHI,PHI)) into the representative set of 'eq'. Of course, this 'eq' returns 'T' if applied to (PHI,PHI). However, it also returns 'T' if applied to (a,b) for any a and b.

This is a very desirable property. A test for equality between a given value and PHI must not be able to compute 'T' if X = PHI and 'F' otherwise because such a test would amount to a solution of the halting problem of Turing machines. On the other hand, if the result of a function does not depend on a particular parameter, then the value of this parameter should not matter at all even if it is undefined.

In order to facilitate the construction of ordered pairs that contain PHI, the functional 'cnstr' must be modified for the following reason. Consider the expression (f & g)(x) = &.[f,g](x). With the original definition of 'cnstr', namely

$$(\text{cnstr}(F \times G))(X) = F(X) \times G(X),$$

$$f(x)=\text{PHI or } g(x)=\text{PHI implies } (f \,\&\, g)(x)=\text{PHI},$$

which is precisely what the addition of PHI to the set of atoms was to prevent. The following definition of construction eliminates the problem:

$$(\text{cnstr}(F \times G))(X) = (F(X) \cup \{PHI\}) \times (G(X) \cup \{PHI\}).$$

This corresponds to the representative set with the pairs

$$\text{cnstr} == \{(((z,y),x), ((z,u),(y,v)) \mid (x,y,z \text{ in } D1) \ \& \ (u \subseteq x) \ \& \ (v \subseteq x)\}$$

Note that this definition changes into 4.2 if PHI is removed from D1 and consequently '$\subseteq$' becomes '='.

The representative set of 'condition' must be modified similarly to

$$\text{if} == \{((y,x),((T,u),(y,v))) \mid (x,y \text{ in } D1) \ \& \ (u \subseteq x) \ \& \ (v \subseteq x)\}.$$

6. CONCLUSION

The domain D3 with PHI as an atom chosen over D1 as the domain for a (functional) programming language has the following advantages.

1. Not only the denotations of functions but the functions (mappings) themselves are members of the domain.

2. Representing the 'undefined' condition by the empty set leads to a very natural model of computational processes.

3. All functions and functional forms are strict or unions of strict functions and constant functions.

4. Since all members of D3 are sets rather than single items, nondeterministic processes can be specified in a natural way without the creation of a new concept.

5. The application operator smoothly facilitates the proper evaluation of expressions that involve the bottom PHI. No special rule is necessary to forbid the test for PHI; the test is impossible to construct although the symbol PHI may be used freely. Also, expressions such as (PHI v true) or (Ø * PHI) are evaluated to yield their proper results -- namely 'true' and 'Ø' respectively -- and are not defined to be PHI.

6. Another advantage of the domain D3 is that it facilitates the introduction of data types in a quite natural way. This issue not treated here is discussed in some depth in [Martin81].

Acknwoledgments

I like to express my thanks to Charles Feustel, who critically read the paper and in many discussions helped me to eliminate several rough edges of the mathematical presentation.

Further, I am very grateful to John Backus and John Williams, who after reading an early version of [Martin81] were kind enough to spend two days with me discussing a number of issues and patiently pointing out inconsistencies and shortcomings. Their criticisms and help with [Martin81] had a profound influence on this paper, too.

Finally, I thank Gyorgy Revesz, who has pointed out to me that some way should be found to handle expressions of the form (Ø * PHI) properly.

REFERENCES

Backus78 Backus, John W. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," CACM 21 8 (August, 1978)

Backus79 Backus, John W. "On extending the concept of program and solving linear functional equations," IBM report, August, 1979

Hayden68 Hayden, S. and Kennison J.F. Zermelo-Fraenkel Set Theory Charles E. Merrill Publishing Company, Columbus, Ohio 1968

Manna73 Manna, Zohar et al. "Inductive methods of proving properties of programs," CACM 16 8 (August, 1973)

Martin81 Martin, Johannes J. "Typed functional programming systems," submitted to TOPLAS, ACM

Plotkin76 Plotkin, G.D. "A powerdomain construction," SIAM J. Computing 5, No. 3 (1976)

Scott76 Scott, Dana S. "Data types as lattices," SIAM J. Computing 5, No. 3 (1976)

Scott77 Scott, Dana S. "Logic and programming languages," CACM 20 9 (September, 1977)

Stoy77 Stoy, Joseph E. Denotational semantics: the Scott-Strachy approach to programming language theory, The MIT Press, Cambridge, Massachusetts 1977

Strachy73 Strachy, Christopher "The varieties of programming languages," Proceedings of the International Computing Symposium, Cini Foundation, Venice (1972)

Williams80 Williams, John H. "Formal representations for recursively defined functional programs," IBM report, July, 1980