FAD, a Functional Programming Language that Supports Abstract Data Types

by

Johannes J. Martin

Department of Computer Science
Virginia Tech
Blacksburg, VA 24061

Technical Report No. CS80005-R

Keywords

Programming languages, functional programming languages, FP-systems, abstract data types, parameterized data types, generic functions, infix operators, FAD.

CR Categories: 4.22, 5.23, 5.24

Abstract

The paper outlines the programming language FAD. FAD is a functional programming system of the kind described by Backus (Backus78]. FAD supports abstract data types, parameterized types, and generic functions. A single scope rule establishes the encapsulation requirements for data type specification and program structuring. Certain syntactic additions improve program readability as compared to pure functional notation.

1. Introduction

Programming restricted to defining and applying functions has a long history. Its theoretical roots reach back to the theory of recursive functions [Kleene36], the lambda calculus [Church41], and the system of combinators [Curry58]. The first well known practical programming language of the functional type (based on the lambda calcules) is LISP [McCarthy60], and a later one is APL [Iverson62]. Both languages, although extensively used by researchers in certain areas, are not among the most popular ALGOL-type languages such as FORTRAN, COBOL, PL/I, or Pascal. However, it has been doubted whether the trade of programming is best served by languages of this conventional type. Experience shows that in particular the ambitious conventional designs that try to embrace the state of the art grow into ever larger, amorphous collections of 'features'. Although this trend may not be technically inevitable, it certainly is conspicuous in practice. Backus [Backus78] calls the conventional languages fat and flabby and points out that a 'large increase in size brings only a small increase in power'.

In addition to this disproportion of size and power, ALGOL-type languages lack mathematical properties conducive to program analysis and verification. This is due in part to their sheer size and multitude of diverse features, in part to their use of successive state transformations as the model of computation. In fact, the variable, the assignment statement, and the subroutine (in contrast to the function procedure), which are the tools used and needed for transforming the state of computation, are responsible not only for what is called side effect and

alias problems but also for rather awkward appendages to otherwise very concise and elegant description mechanisms. Compare, for example, Guttag's original algebraic specification method for abstract data types [Guttag75] with the form extended to accommodate subroutines and functions with side effects [GHM77]. Backus deals with these questions in great depth and finally concludes that only a radically different language structure can eliminate the trouble. He consequently proposes a new breed of languages called Functional Programming (FP) systems. These systems, though related to LISP and APL, are also distinctly different: simpler. The language FAD outlined here belongs into this category.

FP systems basically consist of a mechanism for defining new functions from existing ones. The foundation of any edifice of function definitions is a language defined set of so-called primitive functions.

Functions always have exactly one object as a parameter and they return one object as a result. Both objects, the parameter and the result, may be of arbitrary complexity. There are no variables and hence no assignment statements; every valid phrase is an expression or a definition.

FP languages have many attractive properties. In particular, they

- promote well structured programming,
- provide algebraic rules for program analysis
  and verification,
- do not suffer from alias or side effect problems,

- are easy to compile, and
- facilitate efficient storage management.

In addition, FAD supports

- abstract data types and allows
- parameterized types and
- generic functions.

The absence of side effects makes automatic sharing of like data objects universally possible and transparent to the user. Copying of data structures is never necessary.

Despite these and other advantages, some feel that the concise and well structured FP system programs are frequently much harder to read, to understand, and hence more difficult to design and to maintain than those written with conventional programming languages. However, the author is convinced that this chiefly syntactic problem is not too difficult to correct. It seems that Backus [Backus78] has designed his notation without extensive 'syntactic sugaring' in order to expose the conceptual simplicity of FP systems. In contrast, one of the objectives of this paper is to show that an FP system can be made into a user friendly programming language. Therefore, questions of syntax and of features serving readability are discussed.

An other important point is that FP systems are not history sensitive, that is, they have no means to save definitions or results produced by a computation and to recall them later. Therefore, they must be imbedded into an environment capable of performing these tasks for them. This

issue, addressed extensively in [Backus78], is not in the scope of this paper.

The rest of the paper is organized as follows:
Section 2 gives an overview of FAD, section 3 introduces two mechanisms added to enhance readability, local naming of selectors and infix operators. Section 4 describes the type definition and checking apparatus. It follows the conclusion (section 5), appendix A with two sample programs written in FAD, and appendix B with the formal specification of most of both FAD's syntax and semantics.

2. Overview of FAD

In structure and style, this section closely follows Backus' description of FP systems. Aside from smaller notational differences, which are mostly concessions to the ASCII character set, FAD deviates more substantially from Backus' system by introducing

(i) the notion of a set of items,

(ii) the distinction between items and objects,

(iii) the encapsulation mechanism,

(iv) the type mechanism, and

(v) the already mentioned facilities for naming selectors and using infix operators.

(i) Items are arranged to form sets. These sets are described by item expressions, a system similar to regular expressions. However, sets of items are not regular sets. Sets of items represent both data and functions.

(ii) Whereas items constitute the raw material from which all objects are constructed, objects themselves are pairs of two items called type and value.

(iii) Definitions of functions or data can be encapsulated. With a single scope rule, this mechanism provides the import and export facilities needed for the definition of abstract data types.

(iv) Abstract data types can be declared and their proper use is enforced by the language translator.

Throughout the rest of the paper, comments injected into formal definitions are set off by '//'. The following initial description ignores types and deals in turn with

- item expressions and naming,
- functions,
- functional forms,
- encapsulation.

2.1 Items, item expressions, and naming

All entities in FAD, data as well as functions, are sets of items. Using sets rather than single items as basic units allows treating complex objects, such as functions or carriers of data types, and simple objects, such as numbers or character strings, by the same mechanisms. The result is homogeneity and simplicity.

These sets of items are denoted by item expressions. Items and item expressions are based on a predetermined set of atoms. This set contains numerals, character strings, the boolean values T and F, the null sequence NIL, and possibly some other special symbols.

2.11 Items

Items, the building blocks of item expressions, are either atoms or sequences of items, that is, lists of items enclosed in angular brackets. Commas may be used as separators, if confusion could otherwise arise. NIL denotes the null sequence <>.

Examples of items // a,b,c, ... are atoms //

NIL;

a;

<b>;

<<NIL>,b,<a,c>>.

Formally, items are defined as follows:

item:

(1.1) Atoms including NIL are items,

(1.2) If X is an item, then so is <X>,

(1.3) If X and <W> are items, then <X,W> is an item,

(1.4) There are no other items.

Note, that <X,W> = <X>, if <W> is NIL. The set of all items is sometimes called 'item'.

2.12 Item expressions

The universe of discourse is the set of item expressions, called IE. An item expression denotes a set of items. IE consists of expressions for all finite and certain infinite sets of items.

There is no symbol in FAD that denotes the empty set. However, the empty set may occur as the result of the application of a function to an item or as the value of certain recursive definitions. The meta-symbol PHI will be used to refer to those expressions that denote the empty set. Therefore, PHI is considered to be an item expression although it is not a symbol of the FAD language.

The expressions for finite sets are now defined.

IE:

(2.1) PHI is in IE,

(2.2) all atoms including NIL are in IE.

Suppose A, B, and <C> are in IE then so are

(2.3) <A> // sequence of length 1 //,

(2.4) A|B // union //, and

(2.5) <A,C> // sequence //.

Notes: (i) Conceptually, the sequence of two expressions denotes the set of all sequences that can be formed by appending members of the set denoted by the second expression to members of the set denoted by the first one.

(ii) If A denotes a set S, then <A> denotes the set obtained by surrounding all members of S with angular brackets.

(iii) According to the above definitions, atoms also qualify as items, and items qualify as item expressions, (hence atoms qualify as item expressions). Nevertheless, a notational distinction is not necessary since the proper interpretation is always clear from the context.

The members of IE that denote infinite sets are defined later.

Expressions that denote the same set are considered equivalent. The following list gives the basic equivalences among members of IE.

A|PHI = PHI|A = A;

A|A = A;

A|(B|C) = (A|B)|C = A|B|C;

A|B = B|A;

<A,(B|C)> = <A,B>|<A,C>;

<(A|B),C> = <A,C>|<B,C>;

<A|B> = <A> | <B>.

Thus, the identity element for union is PHI. Union is idempotent, associative, and commutative. Sequencing distributes over union. Note that

<PHI,A> = <A,PHI> = PHI and <PHI> = PHI,

thus sequencing is PHI preserving.

2.13 Naming

Item expressions and modules (see below) can be named. Names are atomic symbol strings which obey the usual rules of identifiers. The expression that associates a name with an object is called a definition and has the form:

name == thing_to_be_named

A name that occurs in an expression may be replaced by the right-hand side

of its definition.

Example

XYZ == <<NIL>,a,b>;

UVW == <c,XYZ,<f>>;

The XYZ that occurs in the right-hand side of UVW's definition may be replaced from XYZ's definition. Thus, UVW represents <c,<<NIL>,a,b>,<f>>.

Names such as XYZ or UVW are called defined atoms. FAD defines certain atoms a priori. These are called primitive atoms (objects).

2.14 Item expressions for infinite sets

Expressions that denote infinite sets can now be defined as recursive formulas.

Suppose R, a member of IE, contains one or more occurrences of the atom S yet undefined, then the definition

S == R

defines a member of IE denoted by S.

There is a third category of expressions in IE: functions. Functions will be introduced in section 2.2.

Examples of expressions for infinite sets

(i) With the primitive atom 'int',
the set of all stacks for integer numbers could
be defined by

```
stack == NIL | <int,stack>
```

which expands to

```
stack = NIL | <int,NIL> | <int,<int,NIL>> | ...
```

By repeated substitution, arbitrarily many items of an infinite expression are generated.

Note: The example of a stack, used throughout the rest of the paper for demonstrating encapsulation, type checking, and the like, has been chosen for its simplicity and publicity. It is sufficient for showing how types, fixed and parameterized, are introduced and handled but it is much too simple to hint at the power of the language.

(ii) The set of all binary trees that accommodate real or integer numbers at their nodes may be defined by

tree == NIL | <tree, (real | int), tree>

2.2 Functions

Functions are the third and last type of expressions in IE. Functions are sets of pairs of items where each pair defines the mapping for one item. For example, a function which contains the the pair

<yi, xi>

has the value yi, if it is applied to the value xi.

Note: The conventional notation for application, f(x) or f:x, and composition, f(g(x)) or f.g:x, implies a flow of processing from right to left. The order of range and domain values in the pairs <y,x> has been chosen to agree with this convention.

This set of pairs, which is usually infinite, is called the representative set of the function; a function is given by its representative set.

Functions may be specified directly as item expressions using the means described above or by combining existing functions through functional forms. FAD provides an initial pool of primitive functions as a starting point. Some of the primitive functions and the functional forms are

introduced below.

Functions map items into items. The application of a function f to an item x is expressed by

f:x

If the representative set of f does not contain a pair <yi,xi>, then f is said to be undefined for xi. A function is, for example, undefined for a value x, if the application f:x leads to a non-terminating computational process. Although non-terminating processes can not simply be ruled out, it is convenient for the analysis of programs to assume that a function has a special value ('undefined') where the defining process encounters an error condition or does not terminate. When the value 'undefined' is also allowed to be part of the domain of functions, some care must be taken to ensure that the resulting total mappings are monotonic. For details see, for example, [Manna73].

In FAD, the desired extension of partial functions is brought about by reinterpreting functions as mappings from sets of items to sets of items rather than from items to items. This is accomplished by the following definition of application.

(3.1) f:(x | y) = f:x | f:y

(3.2) (f | g):x = f:x | g:x

(3.3) <y,x>:z = if x = z then y else PHI

From (3.1) follows that f:PHI = PHI, because

f:x = f:(x | PHI) = f:x | f:PHI

thus f:PHI is a subset of f:x and since this relation is independent of x, it follows that f:PHI = PHI. q.e.d.

Because of (3.3), a so extended function assumes the value PHI, if the original function is undefined. So, PHI represents 'undefined'.

Example of a function

The function 'xor' is the set expressed by

(<F,<F,F>> | <T,<F,T>> | <T,<T,F>> | <F,<T,T>>)

Hence, xor:<T,F> =

<F,<F,F>>:<T,F>
| <T,<F,T>>:<T,F>
| <T,<T,F>>:<T,F>
| <F,<T,T>>:<T,F>

= PHI | PHI | T | PHI = T.

While functions in FAD are always total, for most items in IE, the value of any given function is most likely equal to PHI. Now, with the set D of all items for which the value of a given function f is not PHI, the significant domain D' of f is defined as the set of all subsets of D.

D and D' have the following properties.

(i) D is the l.u.b. of a lattice whose poset is D'.

(ii) For every expression s in IE, there exists an element t in D' such that

$$f{:}s = f{:}t \quad \text{and} \quad f{:}(s-t) = PHI.$$

The lattice over D' is, of course, also a semilattice with respect to union. Since application distributes over union, functions are morphisms of the semilattices that constitute their domains. Therefore, the range R' of a function f is also a semilattice; its l.u.b. is R = f:D.

Functions, though, do not distribute over intersection. For example, let f:a = f:b = c, then the intersection of f:a and f:b is c, whereas the intersection of a and b equals PHI and f:PHI = PHI (not c). Therefore, a function is not a morphism for the complete lattice over D'. With respect to intersection, functions, however, have a property that is similar to but weaker than the distributive law: monotonicity. Montonicity is defined for functions with partially ordered domains and ranges. Let x and y be members of D' than the intersection of x and y is a subset of x as well as

of y. Thus

$$x \cap y \subseteq x \quad \text{and} \quad x \cap y \subseteq y$$

The relation $\subseteq$ is, of course, a partial ordering of D'. A function f over a partially ordered set is called monotonic iff

$$A \subseteq B \Rightarrow f(A) \subseteq f(B)$$

Thus, if f is monotonic then

$$f(x \cap y) \subseteq f(x) \text{ and } f(x \cap y) \subseteq f(y)$$

THEOREM:

Functions in FAD are monotonic.

PROOF:

$A \subseteq B$ implies that there is a C such that $A \mid C = B$.

$f{:}B = f{:}(A \mid C) = f{:}A \mid f{:}C$

Thus, $f{:}A \subseteq f{:}B$. q.e.d.

The Fixed-Point Theory of recursive programs (see, for example, [Scott70] or [Manna73]) presupposes monotonicity for extended partial functions; FAD functions meet this requirement.

Inasmuch as each semilattice (R' as well as D') is the set of all subsets of its l.u.b. and, hence, completely specified by it, the l.u.b. can be used to represent these semilattices. FAD takes advantage of this for the specification of carrier sets of data types (see section 4).

The above specifications allow a function to be the union of other functions. If this occurs, it is assumed that all components of the function are evaluated simultaneously. This facilitates the writing of non-deterministic procedures. It is envisioned, to let the programmer put an upper bound on the number of results computed by such a program. For example, he may specify that only one result is wanted. In this case, the program terminates as soon as the first result is found. If no such bound is given, the program terminates when all partial processes terminate. Therefore, the whole program will terminate, only if all component programs terminate; however, if a bound is specified, the program may terminate and produce results, even if some component processes should not terminate.

2.21 Definition of primitive functions

Instead of enumerating the representative set, one can specify a function f by defining the expression f:x for all items x. This alternate method is frequently more convenient.

In the following definitions, conditional expressions are used. The notation is self-explanatory. It is assumed that X,Y,Z and <W> are items.

Recall that <Y,W> = <Y> if <W> = NIL.

2.22 Examples of functions

Length

```
 lth:X == if (X = NIL)              then 0;
          if (X = <Y,W>)            then 1+lth:<W>;
            otherwise                  PHI.
```

Selector functions

```
 s(i) == if (X = <Y,W> and i = 1    then Y;
          if (X = <Y,W> and i > 1   then s(i-1):<W>;
            otherwise                  PHI.
```

Tail

```
 tl:X == if (X = <Y,W>)             then <W>;
            otherwise                  PHI.
```

Prefix

```
prfx:X == if (X = <Y,<W>>)          then <Y,W>;
          otherwise                    PHI.
```

Suffix

```
sufx:X == if (X = <<W>,Y>)          then <W,Y>;
          otherwise                    PHI.
```

Identity

```
id:X == X.
```

Atom

Let A be the set of all atoms,

```
atom:X == if (X is in A)            then T;
          if (X is in item-A)       then F;
             otherwise                 PHI.
```

Equality

```
eq:X == if (X = <Y,Z>) and (Y = Z) then T;
        if (X = <Y,Z>) and~(Y = Z) then F;
          otherwise                   PHI;
```

Test for NIL

```
isnil:X == if (X = NIL)          then T;
             otherwise                F.
```

Set

```
set:X == if (X = <Y,W>)          then Y | set:<W>;
           otherwise                 PHI.
```

Intersection

```
&:X == if (X = <Y,Z>) and (Y = Z)  then Y;
        otherwise                   PHI.
```

Arithmetic and logical functions

op:X == if (X = <Y,Z>) and Y and Z are good operands
then Y op Z;
otherwise PHI.

Constant functions

Let A be a numeral, a quoted character string, or any other atom that denotes some value other than a function either by convention or by definition then

A:X == value(A) // 'value' is left as an intuitive notion. For example, the value of the numeral 3 is the number 3 //

Thus, if a non-function A is used as the left operand of the application operator, it is interpreted as the function <value(A),item>. Recall that 'item' is the set of all items.

There are many more functions that manipulate item expressions; they are described in the preliminary reference manual for FAD.

2.4 Functional forms

A functional form is an expression denoting a function. This function depends on parameters, which are item expressions imbedded in the functional form. Hence, a functional form is a function distinct from other functions only because of its syntax.

However, not all functional forms are members of IE. Functional forms in IE are called simple, those not in IE are called essential. Essential functional forms treat sets (functions) as a whole, not element by element. Therefore, the distributive law for application does not hold for these forms, they can not be represented as sets of pairs. Nevertheless, they are continuous functionals and, hence, have unique least fixed points [Manna73]. In the following description, the symbols f,g,p denote functions (p denotes predicates), e denotes some expression such that [e] is a valid functional form, the symbols <W>,X,Y,Z denote item expressions (<W> explicitly a sequence).

The following are the definitions of some functional forms.

2.41 Simple functional forms

Composition

```
(f.g):X == f:(g:X)
```

Construction

```
[f]:X == <f:X>

[f,e]:X == prfx:<f:X,[e]:X>
```

Condition

```
(if p then f else g end):X == if p:X = T  then f:X;
                              if p:X = F  then g:X;
                              otherwise PHI.
```

Case

```
(case f of Z end):X == if (Z = Y::g,U)
                        then (if f:X = Y
                                then g:X
                                otherwise (case f of U):X );
                      if (Z = Y::g)
                        then (if f:X = Y
                                then g:X);
                      otherwise PHI.
```

For example,

```
case switch of
  sw1 :: [f1.g1, ... ],
  sw2 :: f3.f4,
  end: Y
```

evaluates to f3.f4:Y if switch:Y = sw2.

Constant

```
"X:Y == X.
```

This form allows to introduce a function or a sequence as a constant. For example

```
["atom,id]:5 = <atom,5>  rather than  <T,5>
```

2.42 Essential functional forms

Insert

```
/f:X == if (X = <Y>)    then Y;
        if (X = <Y,W>) then f:<Y,/f:<W>>;
           otherwise     PHI.
```

Apply to all

```
@f:X == if (ISNIL:X = T)        then  NIL;
        if (X = <Y,W>)         then <f:Y,@f:<W>>;
           otherwise             PHI.
```

While

```
(while p do f end):X ==
                if p:X=T  then (while p do f end):(f:X);
                if p:X=F  then X;
                otherwise PHI.
```

2.5 Examples of function definitions

```
sqr == *.[id,id]

stack == NIL | [id,stack]
```

The second function defines an infinite expression of 'stacks'. The parameter of the function determines the type of objects that are to be stacked. For example

```
stack:int = NIL | <int,stack:int>
```

2.6 Encapsulation

Item expressions, functional forms as well as groups of definitions may be encapsulated by writing

MODULE things to be encapsulated END

The unit created is called a module; modules may be named by

id == MODULE ... END.

The scope of identifiers, items, and functions defined within a module is the surrounding module. More than one definition may define the same identifier as long as no two such definitions are in each others scope. If the scope of a definition is contained in the scope of another one defining the same identifier, but not vice versa, then, throughout its scope, the inner definition overrides the outer one. In any case, references to different definitions are always distinguished, even if the names are identical. Hence, it is impossible, for example, to cheat the typing mechanism by creating several definitions for the same carrier identifier. The attempt to identify a type name with the wrong definition would be detected as illegal. These are the only scope rules needed for encapsulating programs or defining data types. Consider the following example.

```
stk == MODULE
          stack == MODULE NIL | <int,stack> END;
          newstk == (...); push == (...); pop == (...);
          top    == (...);
       END
```

The elements of the set 'stack' are known throughout the module stk, that is, they are known to the functions newstk, push, and so on; outside of the module, they are not known, however, the functions newstk etc. are. Outside of the module, the identifier 'stack' can not be redefined, but the make up of the stack elements is invisible.

The scope rules constitute a refinement of the substitution rules sketched above. Now, these may be stated as follows.

Atomic character strings fall into one of the following three categories.

(i) A symbol x occurs at a place where it is not within the scope of a definition for x: here, x is PHI, unless x has a value by convention as, for example, numerals do (see 2.22, on constant functions). Depending on the case, x is called an undefined atom or a constant.

(ii) A symbol x occurs within the scope of a definition for x but the occurrence is not also in the scope of the right-hand side of the definition (because this is further encapsulated): here, x denotes itself and it may not be used on the left-hand side of another definition; it is called an unavailable atom.

(iii) A symbol x occurs within the scope of a definition for x and this occurrence is also in the scope of the right-hand side of the definition: here, x may be substituted by the right-hand side of the definition; it is called a defined atom.

3. Local naming of selectors and infix operators

The readability of strictly functional programs is frequently rather poor. As an example, consider programs that evaluate polynomials by means of Horner's method:

$$P(x) = ((((An*x + An-1)*x + An-2)*x \ldots)*x + A0$$

In a conventional programming language such as PASCAL the algorithm could be expressed as follows.

```
function POLY (X:real; COEFS:list):real;
   var RES:real;
 begin RES := 0.;
   while ~ISNIL(COEFS)
     do RES := RES*X + FIRST(COEFS);
        COEFS := TL(COEFS);
     end{while};
   POLY := RES;
 end{POLY};
```

In the functional language presented this far, the algorithm appears quite differently. With the input

<X, <An, An-1, ..., A0>>

the program

```
poly == s(1)
        .while ~.isnil.s(3)
           do [+.[*.[s(1),s(2)],s(1).s(3)],s(2),TL.s(3)]
           end
        .[0,s(1),s(2)];
```

would compute the desired value.

But by all its conciseness, this program is certainly not an example of good readability. The mechanisms described next allow to write the program as follows:

```
poly ==
   res {res,x,coefs}
   .while ~.isnil.coefs
      do [res*x + s(1).coefs, x, tl.coefs] end
   {res, x, coefs}.[0, x, coefs] {x, coefs};
```

After becoming used to the functional style, programmers find this program at least as readable as the PASCAL version.

3.1 Local naming of selectors

The pure functional program is hard to read especially because of the accumulation of selectors and the absence of a structural description of intermediate results. Both problems are eliminated by the local naming of selectors.

```
E{id1,id2,...,idn}
```

denotes that the expression E expects a sequence of n members as its parameter and that it refers to the first member of the sequence by id1 rather than s(1), that it uses id2 instead of s(2) and so forth. Thus the notation {id1, id2, ...} is a shorthand for

```
id1 == s(1); id2 == s(2); ...
```

The scope of this definition is limited and extends only to the left. Intuitively, the definition is in effect for as far to the left as the s(i) refer to the same items.

Multiple levels of selection may also be specified. For example

```
a{x,y,{z,a}} = s(2).s(3)
```

Formally, the mechanism is specified as follows. Lower case letters denote identifiers, upper case letters arbitrary combinations of identifiers and balanced braces.

```
(i)    a{X}     == a{X}1
(ii)   a{}i     == a
(iii)  a{b,X}i   == if (a = b)      then s(i)
                                    else a{X}i+1
(iv)   a{{A},X}i == if (a{A}1 = a)  then a{X}i+1
                                    else a{A}1.s(i)
```

Furthermore,

```
f.g{u}:X   == f:(g{u}:X),

[f,g]{u}:X == [f{u},g{u}]:X

(f|g){u}:X == (f{u} | g{u}):X

if p then f else g end{u} == if p{u} then f{u} else g{u} end
```

etc.

3.2 Infix operators

The second modification of the purely functional programming style is the

introduction of infix operators. In a way, it seems that infix operators are quite contrary to the concept of functional programming systems. However, the comma used to denote sequencing or the vertical bar for set union are, in fact, infix operators. Thus, the concept may as well be extended to other ones. For example, while

[f,g]:X denotes the sequence <f:X, g:X>,

(f op g):X denotes the operation op:<f:X,g:X>.

If infix operators are supposed to be a useful device, means must be provided to specify them, unless they were to be restricted to some fixed set of built-in operations. Means for specifying operators differ from those for functions only because they must inform the language parser about the precedence and association rules.

The notation in FAD for this purpose is

op(c o a) == ...

where c and o set the priority of the new operator 'op' in relation to an existing operator 'o'. The symbol 'c' is '=','<', or '>' depending on the priority desired for 'op'.

If '=' is specified, 'op' and 'o' have the same priority, if '<' is specified, the priority of 'op' is less than that of 'o' but greater than the next smaller priority of some other, already existing operator. The

meaning of '>' is analogous to '<'.

The last symbol, 'a', may be 'L' (left) or 'R' (right) and indicates the desired associativity.

Example:

The definition

```
*(> + L) == ...
```

makes '*' a left-associative operator with a priority greater than that of '+'.

4. Types and type checking

4.1 Types

Types are algebras. A simple type algebra has one carrier set, some auxiliary sets, and a collections of operations (functions). The carrier set usually gives the type its name.

Sometimes, a type algebra may have several carrier sets and define several types, namely one for each carrier. Such types are called interrelated.

Again, the names of the types are those of the corresponding carriers.

In recent years, much research has been done in this area. A rather representative collection of papers can be found in [Yeh78].

FAD provides the programming facilities necessary for dealing with data types as algebras.

Definition

An object is a pair

<C, x>

where C denotes the l.u.b. of the carrier of a type and x is a member of the carrier. Therefore, x must be a subset of the set given by C.

There are certain primitive types provided by the language. Among these are the types

char, bool, int, real

With these, new types can be constructed by specifying carrier sets and primitive operations. The specific structure of the new type's carrier set, which is described by the right-hand side of a set definition, is normally hidden from the user by encapsulation so that instances of the type can only be manipulated by the primitive operations specified. This

hiding has another effect. As described in section 2.6 on 'encapsulation', an atomic item, such as a type identifier, denotes only itself unless its definition is fully visible from where it is being referenced. Therefore, the user sees type identifiers as unavailable atoms not as sets. If the type is parameterized (see 4.11 below) such that the actual type appears as a function application, then the application can not be evaluated in the user's module but remains an expression of the form type:parameter. This expression can be compared with others for equality and it can be passed down into other modules where evaluation may be possible.

Examples of objects

```
<char, ('a'|'b')>

<int, 5>

<stack:int, <5,NIL>>
```

Objects may be named. The example below defines 'stack' as the name for an infinite set of the type 'type'. The outermost angular brackets of the right-hand side of a definition are dropped for readability. Nonetheless, the right-hand side is considered a sequence, namely the pair <type,value>.

```
stack == type, MODULE NIL | <int,stack> END
```

The type of an object is not necessarily simply a primitive or defined type as shown above. Consider the result computed by the functional form

[id*id,id]:x

which is <x*x,x>

If x is the integer 4, then the object computed has the form

<<int,int>,<16,4>>

Here the type of the object is a sequence of two types and, hence, the pair structure of the object is visible. Consequently, the type checking mechanism will allow the use of selectors for seperately accessing the 16 and the 4. Such a type is called weak whereas types referred to by unavailable atomic identifiers (see section 2.6) are called strong. Strongly typed objects can only be processed by certain operations called the primitives of the type. Objects with weak types can be analysed with selectors and other sequence operations.

4.11 Parameterized types

A parameterized type is a collection of disjoined types that all have the same set of primitive operations. An example is the parameterized type 'stack'. Its members are stacks that differ by the types of objects they stack. Frequently, there is a primitive operation for the creation of

particular instances of the type. For example with the proper stack primitive NEWSTK, an instant of a new (empty) stack for, say, integer objects would be created by

NEWSTK:int,

a stack for real numbers by

NEWSTK:real.

In FAD, the carrier set for such a type is defined by a function, for example by:

stack == NIL | [id,stack]

The application stack:int defines the set

NIL | <int, stack:int>

which expands to

NIL | <int,NIL> | <int,<int,NIL>> | . . .

Parameterized types are very useful because they permit to specify a large family of types by a single definition. For a complete specification of the parameterized data type 'stack', see the appendix.

4.12 The type of a function

Recall that a function is a set of pairs <y,x> of items where the elements y and x of the pair are members of the l.u.b.'s of the function's range (R) and domain (D), respectively. Hence, a function is a subset of <R,D>. It makes good sense to call the set <R,D> the type of such a function,

(i) because functional forms applied to functions over the same pair <R,D> have certain closure properties,

(ii) because a function is a subset of <R,D> thus <R,D> is indeed the l.u.b. of the carrier set of the type.

Furthermore, functions should be strong types in order to disallow the manipulation of their representative sets by anything but functional forms and the application operation. This establishes the complete seperation of the specification of functions from their implementation. It ultimately justifies to think of functions as sets of pairs but to implement them as algorithms that compute only the pair needed at the time of invocation.

Finally, since most functional forms can be applied to all functions without regard of their actual domains and ranges, it is better to look at all functions as one parameterized type rather than (infinitely) many specific types.

For example, the function that converts real numbers into integers by truncation would be defined as

```
TRUNC == function:<int,real>, . . .
```

A typed function is called a functional object.

Examples for definitions of functional objects

```
SQR   == function:<real,real>,id*id;

TOP   == function:<int,stack>, s(1);

STACK == function:<type,type>,(NIL | [id,STACK]).
```

Where necessary for distinction, functional objects are capitalized throughout the rest of this paper.

4.2 Type checking

In order to decide whether an application of the form

F:X

is legal, one must determine whether the type of the object

X = <C, x>

is compatible with the domain of the function

F = <function:<R,D>,f>.

If so, the object

<R, f:x>

should be computed, otherwise

F:X = PHI

In the simplest case, both C and D are strong types or sequences of strong types. Therefore, compatibility exists if and only if C=D. Suppose now that D is a set of items such as int|real and C is int. Clearly,

F = <function:<R,(int|real)>,f>