Department of Computer Science
University of Manchester
Manchester M13 9PL, England

Technical Report Series
UMCS–92–4–3

John Sargeant

UFO - United Functions and Objects
Draft Language Description

UFO - United Functions and Objects
Draft Language Description

John Sargeant
Department of Computer Science
University of Manchester
Manchester U.K.
js@cs.man.ac.uk

Abstract

UFO is a general-purpose, implicitly parallel language designed to allow a wide range of applications to be efficiently implemented on a wide range of parallel machines while minimising the conceptual difficulties for the programmer. To achieve this, it draws on the experience gained in the functional and object-oriented "worlds" and attempts to bring these worlds together in a harmonious fashion.

1 Introduction

This document is an incomplete draft description of the UFO language. This first section describes the motivation behind UFO, its principal features, and contrasts it with other approaches. The second section describes a purely functional subset of UFO. The third deals with functional arrays and loop structures, which are important for exploiting parallelism, particularly in numeric computation. Later sections deal with the additional features to do with state manipulation, and other issues such as program structure and I/O. The appendices give further examples, and the full syntax.

1.1 The problem

Despite many years of research, implicit parallel languages are not in widespread use; Fortran rules. And yet the problem of programmability of parallel machines is acute, especially as the scale of machines increases rapidly. The Japanese EM5 machine, for example[1] will have 16000 processors.

Attempts at implicit parallelism have been made via strict, lenient, and lazy functional languages, logic languages, and parallel object-oriented languages such as ABCL[8], POOL[9] and actor languages [10]. Of these, strict functional languages (SFLs) are the simplest, and therefore the easiest to implement efficiently. A notable success is SISAL[2], which for some numeric applications runs as fast as Fortran on Crays. Efficient SISAL implementations also exist on shared memory multiprocessors, and a distributed memory implementation is in progress[13].

However, the strict functional style is restrictive in many ways, for instance in dealing with interaction, state-based problems, and nondeterminism. It also has poor encapsulation properties, as the state of the computation must be explicitly threaded through the computation as function parameters. Adopting lazy semantics, as in Haskell, solves some of these problems but not all, and introduces heavy performance penalties. Furthermore, the object-oriented approach to software construction has advantages over the functional approach, in terms of adaptation to change and reusability of software components (See [4] for a very clear and thorough exposition of OO design principles). For instance, the pattern matching against constructors used in many FLs directly conflicts with notions of data abstraction.

We propose to retain the efficiency of strict functional programming, and introduce objects to solve the above problems. The resulting language is not purely functional, but the encapsulation of state provided by the objects allows the programmer to think in a functional manner much of the time. In addition, the type system provides important safeguards.

UFO objects allow many constructs to be expressed which cannot be expressed, or cannot be implemented efficiently, in a strict functional context, for instance queues, recursive array definitions, accumulators, nondeterministic merges, and semaphores. A consequence of this power is that, in the limit, all the traditional concurrent programming problems of deadlock, starvation, nondeterminism etc. can occur. We feel this is inevitable in a sufficiently powerful language, and aim to minimise the problem by providing safeguards in the language definition and encouraging the use of appropriate programming methodologies.

Many applications can be written in a functional style, with limited, well encapsulated use of state. A standard library of generic objects will be provided to minimise the need to write new non-functional code. On the other hand, UFO programs can also be written in terms of networks of autonomous communicating objects where appropriate.

1.2 Principal features of UFO

The functional part of UFO is largely derived from SISAL. The object-oriented features are heavily influenced by Bertrand Meyer's Eiffel[11], although many of them are common to other OO languages. UFO can also be viewed as a concurrent object-oriented language.

The resulting language is *not* a pure functional language(PFL), and it is therefore possible to write code which is not referentially transparent. However, the semantics are such that the implementor is allowed to execute functions in parallel exactly as in a PFL. To a large extent, the conceptual problems caused for the programmer are localised to those parts of the program which explicitly manipulate state.

The principal features are as follows:

Parallelism by default. Like SISAL and ID, UFO does not impose a sequential evaluation order on execution, beyond that required by the data dependencies, except in a few clearly defined cases.

Functional arrays. Like SISAL, UFO provides support for functional arrays, closely integrated with the loop structures.

Support for object-oriented software design. Objects with inheritance are provided to give the usual benefits of information hiding, reusability, inheritance polymorphism etc.

Encapsulation of state. Objects are also used to encapsulate state in such a way that the problems it poses in a parallel world are minimised.

Strong static polymorphic typing. As well as the usual benefits of strong static typing for software reliability, the type system gives essential information about the state properties of objects.

Multiple forms of parallelism. Programming very large parallel machines requires that all possible levels of parallelism should be available. UFO supports several forms of parallelism: fine grain dataflow (if required), functional divide-and-conquer, array/loop parallelism [1] lenient/nonstrict evaluation (if required), and asynchronous communication between objects.

[1] What I call array/loop parallelism is probably equivalent to "data parallelism", provided that the widest definition of the latter term is used. It is most certainly not limited to loops which can be vectorised, or executed in a SIMD manner.

1.3 Contrast with other approaches

Few of the individual ideas in UFO are new; in fact most are quite old. However, the particular synthesis of ideas differs substantially from all existing languages.

- Unlike pure functional languages such as SISAL, Haskell etc., UFO tackles the problem of state head-on. One consequence of this is that problems which in PFLs are solved by non-strict semantics are dealt with explicitly using objects in UFO, and so strictness is sufficient. Functional UFO code can be implemented leniently if desired. There are also facilities which allow non-strict evaluation to be programmed.

- Unlike, for instance, LISP and its derivatives, or SML, the presence of state does not impose a global sequential evaluation order on the program. Evaluation is parallel by default, except when explicitly sequentialised, either by synchronisation or within procedural methods.

- Unlike object-oriented languages, UFO retains many of the features of modern functional programming, such as higher-order functions and strong static generic typing, and allows the functional style (a function combines several values) to be used alongside the object-oriented style (a message gets sent to a single object).

- Unlike imperative languages, UFO has features designed to promote the functional style of programming, such as constructs to create arrays monolithically rather than by individual updates.

- The objects in UFO are quite similar to the traditional idea of monitors. However, unlike monitors, UFO objects are first-class citizens which are dynamically creatable and can be included in arbitrary data structures.

- UFO has many of the features of concurrent object oriented languages such as ABCL, actor languages, Concurrent Smalltalk, or POOL, but differs from these in supporting the functional style too, in the array and loop facilities, and in having both generic typing and inheritance.

1.4 Some important criticisms

1.4.1 Why yet another language?

Anyone proposing to design yet another programming language has a duty to justify this decision very carefully. I justify UFO as follows:

1. There is a clear need. Almost all real large-scale parallel programming is done in Fortran, usually exploiting only very restrictive forms of loop/array parallelism. This is surely inadequate for future very large machines and very large, complex problems.

2. No existing programming language (as far as I know) is close to meeting this need. This is reflected in the large number of differences (shown above) between UFO and other languages.

3. The design and implementation of UFO is predominately a combination of existing successful technologies. For instance exploitation of array/loop parallelism in SISAL, the object-oriented methodology of software construction, and strong static polymorphic typing, as used in many functional languages and in Eiffel.

1.4.2 Things which look like functions and aren't

A natural objection for functional programmers is that UFO functions are not pure, and so lose many of the benefits of functional programming. In fact, function definition in UFO is constrained in such a way that the user of a function is shielded from any internal state manipulation, and can use the function exactly as if it was written in a pure FL in most cases. A later section discusses this in more detail.

Consider a compiler. It is perfectly reasonable for the user of a compiler to regard it as a function from source code to object code. In fact, it's probably written in an imperative language and involves lots of state manipulation. On the other hand, if it's written in a pure FL, the writer of the compiler will also regard all its components as functions. Even in that case, there are still state transitions going on under the surface (overwriting of graph nodes in a graph reduction implementation of the FL, for instance).

UFO ensures that (to a close approximation) Functions Really Are Functions by the combination of three features:

1. Use of objects to encapsulate state.

2. Use of strong static typing to ensure that stateful values can be identified statically wherever necessary.

3. A set of restrictions on the way functions may be defined.

It is possible (although very naughty) to write contorted procedural code with all sorts of nasty, ill-controlled side effects, but the result then has to be called a procedure, not a function.

1.4.3 Deadlock, nondeterminism etc.

In the limit, a UFO program may exhibit all the classic problems of concurrent programming: deadlock, starvation, unwanted nondeterminism etc. The language restrictions which would have been necessary to avoid this would have been extremely severe. In the end, the programmer has to be careful.

The language design is, of course, tailored towards minimising the chances of such things happening. In particular, the functional subset of UFO is sufficiently expressive that much code will use that part only. Common uses of state will be packaged up in a standard library of "trusted" classes.[2]

In some circumstances, nondeterminism may be acceptable, or even desirable. For instance, the result of a set union function may be a representation of a set with the elements in a nondeterministic order. So long as the same set is always produced, and the other set operations assume no order, the nondeterminism is harmless. There is a class of algorithms which can be executed in parallel more easily if this sort of nondeterminism is allowed. An example appears in appendix A.

1.5 A note on syntax description

Fragments of syntax are given throughout this document. In some cases, they are simplifications of the actual syntax which is given in appendix B. The syntax is given in the extended BNF used by the Karlsruhe University parser generator lalr [12]. The conventions are as follows:

```
(* ... *) Comment
'token'   Literal token
[ X ]     Optional X
X || Y    Sequence of Xs separated by Ys
X+        One or more Xs
X*        Zero or more Xs
X Y       X then Y
X | Y     Either X or Y
```

In addition, the type of a construct (usually an expression) is sometimes informally included in the syntax, eg:

```
(Int)expression
```

2 Functional UFO

2.1 Simple functions, recursion etc.

The following program fragment illustrates many of the basic features of UFO.

```
type RtoR =  Real -> Real;

function trap(l, r: Real; f: RtoR ): Real
is    (r-l) * (f(l) + f(r))/2.0
```

[2]An interesting analogy is the use of pointers (references) in Eiffel. Meyer claims that explicit manipulation of pointers is usually only necessary in building basic data structures. Once these are present in the standard library, the applications programmer rarely has to worry about them.

```
function area(l,r,est,tol: Real; f: RtoR ): Real is
{
mid:Real = (l + r)/2.0;
a1 = trap(l, mid, f); a2 = trap(mid, r, f);
newest = a1 + a2
return
   if abs(est - newest) < tol
   then newest
   else area(l, mid, a1, tol/2.0, f) + area(mid, r, a2, tol/2.0, f)
   fi
}
% End of area
```

Function area integrates the function f between bounds l and r, by recursively dividing the range in half until a trapezium rule approximation (function trap) is accurate enough (i.e. successive estimates differ by less than the given tolerance, tol). Readers used to SISAL will notice that, apart from syntactic differences, the program is similar to the equivalent SISAL program.

The following features are shown:

- Like SISAL and Eiffel, UFO is strongly typed. The parameters and results of function must explicitly be given types. Constants are of unique type which can be lexically determined. The types of other values (such as a1, a2 etc.) can therefore easily be inferred, although they can be explicitly stated, as in the case of mid above.

- The body of a function is an expression. One sort of expression is the *block* construct which allows the definition of local *value names*:

  ```
  block : '{' valdefs 'return' expression '}'
  ```

  Operationally, the *value definitions* can be executed in any order consistent with the data dependencies (e.g. a1 and a2 can be evaluated in parallel, but mid has to be evaluated before either of them).

- The absence of sequential execution order means that, of course, the two calls to area can be executed in parallel.

- UFO, like SISAL, is strict (although an implementation may choose lenient evaluation for functional code). As a result, mutually recursive local value definitions don't make sense[3], and they are syntactically prevented by requiring that any use of a value name should textually follow its definition.

[3] They don't make sense anyway with scalars, but this becomes an issue with, for instance, arrays

- Function types and function values are allowed. Function values may be passed as parameters, returned as results, incorporated in data structures etc. (e.g. an array of RtoRs is a valid type) but there is no mechanism to construct function values at runtime.

Readers familiar with SISAL will have noticed some syntactic differences and will probably be wondering why:

1. Value definition uses =, rather than :=. This is because UFO has real, full-blown assignment, and I want to reserve := for that. A value definition really is a once-and-for-all definition of a value, and as such = seems appropriate. For similar reasons, the term "single assignment" is avoided, and "value definition" is used instead.

2. The block construct replaces **let** ... **in**. This is mainly because blocks are used in a more general context, where **let** ... **in** would have been inappropriate.

3. The syntax for specifying the result type of a function is changed from **(... returns <type>)** to **(...): <type>**. This makes type specs more uniform - attaching a type to something is always done with **:<type>** - and more concise. The notation will be shown to fit neatly when classes are introduced.

4. Names of primitive types start with an uppercase letter. This is to fit in with the convention used for class definitions.

5. The syntax has been made generally less verbose. In particular, since the body of a function is an expression the SISAL "end function" is redundant.

The builtin scalar data types are **Int**, **Real**, **Double**, **Bool**, **Char**, and **Bits**. The number of bits used to represent each of these is machine dependent, but Int and Bits values should have at least 32 bits, Reals at least 32, and Doubles should be larger than Reals[4].

2.2 Functional objects

2.2.1 Class definition

The following defines the interface of a complex number class:

```
class Complex( r, i: Real ) % Real and imaginary parts.
functions
  re:    Real;
  im:    Real;
  rad:   Real;
```

[4]Early implementations will probably not distinguish between Reals and Doubles, but "proper" Doubles will obviously be necessary for realistic numerical applications.

```
  theta: Real;
  add ( c: Complex ): Complex;
  mult( c: Complex ): Complex;
  square: Complex;
end
```

The effect of this declaration is as follows:

1. To introduce a new type, called Complex. All classes are types, but there are types which are not classes (e.g. scalars).

2. To give the types of the parameters needed to create an object of type Complex. In fact, the names of the parameters are also required. This is simply to make it easier for the class definition to be self-documenting.

3. To list the functions available on Complexes. These functions are the only way to manipulate such objects, so the representation is hidden in the usual way - it might be rectangular coordinates, or polars, or even both. Again, the names of any function parameters are included for documentation purposes, although it doesn't make much difference in this case.

2.2.2 Implementation

The implementation might look like this:

```
imp Complex( r, i: Real )
re: Real = r; im: Real = i %The instance values.

function rad: Real is sqrt( re*re + im*im )

function theta: Real is arctan( im/re )

function add ( c: Complex ): Complex
  is Complex( re + c.re, im + c.im )

function mult( c: Complex ): Complex
  is Complex( re*c.re - im*c.im, re*c.im + im*c.re)

function square: Complex is mult( self )

end %Complex
```

The implementation starts with value definitions for the instance values. Every instance of class Complex will therefore have defined instance values re and im - there's no concept of an uninitialised instance value.

An object (an instance of a class) is created by a *constructor* consisting of the class name and actual parameters, eg

```
z = Complex( x, y )
```

A function is applied to an object with

```
funcall : expression '.' identifier [ '(' parameters ')' ]
```

where the expression before the dot evaluates to an object and the identifier is a function defined on that object[5], e.g.

```
z.re
```

This is the same convention as in C++ and Eiffel. This notation, and the way the class is defined, observe what Meyer calls the *principle of uniform reference.* This says that the user of a class (a *client*) shouldn't need to know whether a particular value is stored or calculated; the syntax for getting it should be the same in either case. As in Eiffel, this is done by allowing instance values to be visible, and having the same syntax for referencing them as for parameterless function calling. They are made visible by defining them in the interface as if they were parameterless functions.

To refer to the object itself, a class uses the reserved identifier **self**, as in the (rather artificial) square function. This function also shows that the "objectname." is dropped when a class refers to one of its own functions. [6]

2.2.3 Alternative function call syntax

We appear to have different function calling syntaxes for class functions and "ordinary" ones like those in the binary integration example. Not only is this ugly, it will give us problems with generic types and higher-order functions. The solution is a simple syntactic trick. For a function f with a single parameter, a call to f with an actual parameter x can be represented as either f(x) or x.f. For a function f of $n > 1$ parameters, a call can be represented as f(x1 ...xn) or x1.f(x2 ...xn). So, for instance, a call to the complex multiply function can be represented either as:

```
x = y.mult(z)
or
x = mult(y,z)
```

In this case the second style is more natural, but this varies according to the particular function. Since this is a purely syntactic transformation, it applies to all functions, including those not defined in classes (e.g. the area function above), although the dot form is less likely to be appropriate in such cases.

[5]In general, the function may also be the result of an expression. It is syntactically constrained to be an identifier to avoid a parsing ambiguity, but we can write a "f = exp" where exp yields a function value and then use f in a funcall.

[6]In this case, mult(self) is equivalent to self.mult(self). However, for objects with state, the behaviour of a local method call like this will be different from that of the general case obj.method

Functions of different classes are allowed to have the same name (eg different sorts of numbers may have a mult function), so we are allowing a limited form of function name overloading. This will normally be resolved automatically by the type checker. For documentation purposes, it is possible to attach a type specification to an identifier (and to some other sorts of expression), e.g.

```
x: Complex = y.mult(z)

b = map( z, square: Complex -> Complex )
```

(The map function is defined below). In a few cases, such type specifications are required[7].

2.3 Further simple features

The following, more sophisticated version of Complex shows some further simple features of UFO.

```
class Complex( re, im : Real ) % Real and imaginary parts.
constructors
  Polar( rho, ang: Real )
functions
  re:     Real;
  im:     Real;
  rad:    Real;
  theta:  Real;
  square: Complex;
operators
  + ( c : Complex ): Complex;
  * ( c : Complex ): Complex;
  * ( r: Real ): Complex;

imp
  store re, im;

constructor Polar is
  re =  rho*cos(ang); im = rho*sin(ang);
endcon

function rad is sqrt( re*re + im*im );
function theta is arctan( im/re );
function square: Complex is self * self;
```

[7]Some such cases are shown below. A more formal definition of the type system is in progress.

```
operator  +( c : Complex ): Complex
  is Complex( re+c.re, im+c.im);

operator *( c: Complex ): Complex
  is Complex( re*c.re - im*c.im, re*c.im + im*c.re);

operator *( r: Real ): Complex is Complex( re*r, im*r );

end
```

The features illustrated are:

- A rather snappier class definition can be given by combining the class interface and implementation part, and omitting function headers in the implementation.
- Alternative constructors can be defined, like Polar above. The body of a constructor is a series of value definitions, which mirror the default value definition for the class (i.e. they must define exactly the same set of values) and replace the default definitions if an instance of the class is created using the constructor. Note that an object created using Polar is still of type Complex, and is still represented in rectangular coordinates. Constructor names (including the class name) may be overloaded. Two constructors are distinct if any of their parameters are of different types.[8].
- The common case where parameters to the class, such as re and im, are simply stored as instance values, is made more convenient by a keyword **store**, which avoids the need to introduce extra names and do the obvious value definitions.
- The standard infix operators can be overloaded, possibly in multiple ways within the same class. Overloaded operators retain their original precedence and associativity. The intention is that, for instance, + should be used to add different sorts of numbers, ++ to append different sorts of collections of things together, and so on.

Note that one option for the multiply operator is missing, namely "Real * Complex", since the class type is implicitly the type of the first parameter. For two class types A and B, "A op B" will be defined in class A, and "B op A" in B. In a case such as this, where one of the types is not a class type, the operator has to be defined outside a class definition, e.g.

```
operator * (r: Real; c: Complex): Complex is c * r;
```

[8]This is a provisional rule, which may be changed.

2.4 Simple arrays and loops

To be plausible for numeric applications (and probably many others too) a programming language needs arrays, and appropriate structures for manipulating them. Work on SISAL and ID has shown how to use arrays in a functional context, and how to implement them efficiently. The basic problem with functional arrays is that every "update":

```
B = update( A, I, V )
```

logically creates a copy of A, with the Ith element replaced by V. Although it is possible to do a lot of analysis to reduce the amount of copying, there is still potential for a great deal of inefficiency, and anyway programs written in this style tend to be messy and serial, because the "current" version of the array has to be threaded through the computation.

The solution to this problem is to provide structures for manipulating arrays *monolithically*, e.g. expressions which generate a whole array at a time. These are typically loop structures (as in SISAL) or array *comprehensions* as in Haskell (ID has both). UFO has loops, as they allow a wide range of structures to be expressed easily, and can be implemented efficiently[9].

Some examples of arrays are as follows:

```
a1 = [1,2,3,4,5];            % An Array[Int]
a2 = [1 to 5];               % Same as a1
empty = []: Array[Int]  % An empty array requires an explicit typespec.
three = a1[3];               % = 3
```

Arrays are the basic built-in structured data type in UFO, in much the same way as lists are in many functional languages. The main difference is that the programmer should assume that indexing is a constant time operation (in the sense that it's independent of the size of the array), while adding an element may not be. UFO has lists too (see below).

A loop is an expression, the simplest form of which has the following syntax:

```
loopexp ::= 'for' generator 'do' valdefs 'return' expression 'od'
generator ::= identifier 'in' expression
```

The body of a loop is therefore very similar to a block. e.g.

```
function sumsquares( xs: Array[Int] ): Int is
for x in xs do
y = x*x
return
  sum(y)
od
```

[9] Comprehensions have advantages too, and future versions of UFO may have them.

The expression in the **in** part (xs) evaluates to an array, and an instance of the loop body is generated for each element (x). These can all be executed in parallel. Each instance value created in the body (y) corresponds to *an array* in the **return** part. So sum is intended to be a function which takes an array of Ints and returns their sum.

This implicit conversion of loop values to arrays in the **return** part is new in UFO. SISAL has explicit constructs in the **return** part which gather arrays, sum loop values etc. The UFO approach abolishes all such constructs, and allows arbitrary functions of appropriate types to be used in the **return** part. A possible disadvantage is that users may find the implicit conversion confusing.

Note that the notation for defining arrays shown above can be used to give more familiar-looking generators, for instance the above loop can be written:

```
for i in [1 to size(xs)] do
y = xs[i] * xs[i]
return
  sum(y)
od
```

The facilities described above will be sufficient for many applications. Advanced array and loop facilities are described in section 3.

Implementation note: Much of the work in creating an optimised UFO implementation will be in array representation. Depending on how an array is used, different mappings across processors will be appropriate. There are various games to be played with descriptors. Alternative array representations will sometimes be appropriate, e.g. lists, dataflow token streams, or none at all (i.e. optimised out by sharing analysis).

2.5 Genericity

UFO, like many functional languages (and also Eiffel) allows generic types.[10] Functions may take arguments of generic type in the usual way:

```
typevar Alpha, Beta;

type AtoB = Alpha -> Beta;

function map( a: Array[Alpha]; f: AtoB ): Array[Beta]
is
for xs in a do ys = f(xs) return ys od
```

Class definitions may also be parameterised in a similar way:

[10] I've followed Meyer's convention of referring to this as "genericity" rather than just "polymorphism" to distinguish it from the polymorphism of inheritance.

```
typevar T;

class Stack[T]
functions
  is_empty: Bool;
  push(newval:T): Stack[T];
  pop: (T, Stack[T]);
  top: T;
end
```

Notice how, since this is a functional stack, push and pop return new Stacks. The implementation could be via arrays, but a list representation (see below) would be more efficient.

The valid uses of a value of variable type T are:

- Use on the lhs of a value definition, where the rhs expression is also of type T.
- Use on the rhs of a value definition, where the lhs value is of type T.
- As a parameter to a function of the appropriate type.
- As the result of a function of the appropriate type.
- As input to certain functions which are defined for all types, such as (shallow) copy and equality test. These operations are yet to be fully defined.

Implementation note: Languages with this feature usually have the property that all values can be represented by bit patterns of the same size (usually 32 bits). This may not be the case in UFO, e.g. because we have double-precision reals. I claim that this is not a serious problem. Future architectures may in any case solve the problem by making everything 64 bits. Assuming this is not the case, there will be a small number of possible sizes (e.g. 32, 64, 96 bits). In the worst case, it is necessary to generate, for a function with generic parameters, a version corresponding to each possible size for each type variable (e.g. 9 versions of array_map). Versions not actually used can be thrown away (at worst) at link time [11]. Since such functions are usually small, and rarely have more than two type parameters, I don't consider this to be a serious problem.

2.6 Inheritance

OOLs typically have quite sophisticated inheritance mechanisms. Eiffel shows how this can be reconciled with strong static generic typing.

There are two major decisions to be made in designing an inheritance mechanism:

[11] Without separate compilation, the number of versions can be restricted to those which are actually used. In general, UFO programs can be separately compiled, but will often be much more efficient if compiled as a whole.

1. Single or multiple inheritance.

2. Static or dynamic binding.

Multiple inheritance has advantages, and I have no objection in principle to having it in UFO. However, it introduces complexity into the language and its implementation, so it seems better to leave it out for the moment. It may well appear in future versions.

Dynamic binding is desirable because it allows inheritance to provide a form of polymorphism, as well as simply a way of reusing code. There is a fairly straightforward, reasonably efficient, implementation (at least in the case of single inheritance). UFO has dynamic binding, but "switches it off" in a number of circumstances. The mechanism for achieving this restriction is to allow inheritance in such circumstances, but in such a way that the child class is not type-compatible with its parent. There are a number of cases in which we may want to do this:

- When maximum efficiency is crucial[12].

- When the child class introduces state, whereas the parent was stateless. We will see later that we need to be able to identify statically which objects have state.

- When the child has an incompatible interface to the parent.

The general philosophy can be described as follows. If inheritance is used to describe an "is_a" relationship between parent and child, then polymorphism and dynamic binding are appropriate. If it is used simply to reuse some implementation code, they may not be. For instance, if Penguin inherits from Bird[13] we would expect to be able to have a collection of Birds which include Penguins. However, if we implement a Stack by inheriting from List, some of the List operations may be inappropriate for Stacks, the interfaces will look quite different, and having a collection of Lists some of which are actually Stacks doesn't seem very useful.

The features will be described via the usual geometric shapes example.

2.6.1 Simple static inheritance

Suppose we have the following classes:

```
type Angle = Real;

class Point( x, y: Real )
functions
  x: Real;
  y: Real;
  translate( shift:Point ): Point;  % x and y shifts taken from "shift".
```

[12] Remember we are competing with Fortran, not Smalltalk!

[13] Suitably redefining functions to do with flying.

```
  rotate( about:Point; by:angle ): Point;
  distance( start:Point ): Real;
end

% Implementation of Point is irrelevent, but obvious anyway.

class Polygon( vertices:Array[Point] )
functions
  translate( shift:Point ): Polygon;
  rotate( about:Point; by:angle ): Polygon;
  perimeter: Real;
  area: Real;
end

imp Polygon
  store vertices; num_vertices = size(vertices);

function  translate( shift:Point ):Polygon is
Polygon(
  for v in vertices do
  newpoly = v.translate( shift )
  return newpoly
  od
  )
% ... etc.
end % Polygon
```

The usual next step is to define a Rectangle class inheriting from Polygon.[14]

```
class Rectangle( vertices: Array[Point] ) inherits Polygon
% translate, rotate, perimeter, area inherited from Polygon.
constructors
  Rectangle( botl, topr: Point )
functions
  diagonal_length: Real;
endclass

imp Rectangle
  store vertices;
  num_vertices = 4;
  % lengths of horizontal and vertical sides.
```

[14]Mario Wolczko has pointed out various errors in, and grotty features of, this example, and in particular that a cleaner object-oriented design would have an abstract polygon class from which both Rectangle and the concrete Polygon inherit. My cheap and nasty design serves the current purpose, though.

```
  vside: Real = distance(vertices[1], vertices[2]);
  hside: Real = distance(vertices[2], vertices[3]);

% Non-default constructor, for rectangle orthogonal to axes.
constructor Rectangle( botl, topr: Point ) is
  vertices = [Point(botl.x, botl.y), Point(botl.x, topr.y),
              Point(topr.x, topr.y), Point(topr.x, botl.y)]
  num_vertices = 4;
  % lengths of horizontal and vertical sides.
  vside: Real = topr.y - botl.y
  hside: Real = topr.x - botl.x;
end

% Reimplementation of area & perimeter, simpler than for Polygon.
function area is hside * vside;
function perimeter is (hside+vside)*2;
% And the new function.
function diagonal_length is sqrt( hside*hside + vside*vside );
endimp
```

However, there is a problem here which doesn't occur so often in standard OOLs. Translate and rotate return Polygons as results rather than updating the object being translated/rotated, because we're functional. If we translate or rotate a Rectangle, the result is a Polygon - a perfectly good Polygon, but it's lost its rectangleness! Being forced to reimplement such functions would throw away much of the advantage of inheritance. A solution is to introduce a keyword **Mytype** which stands for the type of the class under consideration. So, for instance, translate becomes:

```
function  translate( shift:Point ):Mytype is
Mytype(
  for v in vertices do
  newpoly = v.translate( shift )
  return newpoly
  od
  )
```

and this will produce a Polygon when used in Polygon and a Rectangle when used in Rectangle. Note that **Mytype** does not imply dynamic binding; it is simply a syntactic shorthand for writing out the code for (eg) translate once in each subclass, with the class name substituted.

In fact, a similar case occurs in Eiffel. The solution used is to allow type declarations of the form **like** <anchor> where <anchor> is (effectively) some instance value, or **current**. **like current** gives exactly the effect of **mytype**.

In general, an inheriting class can do the following:

- Introduce new instance values.
- Change the rhs expressions initialising instance values.
- Introduce new functions, operators, and constructors.
- Reimplement existing functions, operators, and constructors.

2.6.2 Deferred classes

Most OOLs have a way of defining classes which leave some or all of the implementation to their descendents. Such classes are called deferred classes in Eiffel, or abstract superclasses in Smalltalk. There seems to be no problem with introducing such things into UFO, e.g.

```
deferred class Shape
  translate( shift:Point ): Mytype;
  rotate( about:Point; by:angle ): Mytype;
  perimeter: Real;
  area: Real;
end
```

In Eiffel, if any function of a class is deferred, the whole class is said to be deferred, and it's illegal to create an object of that class. This makes sense in the example above, but in general it seems a bit restrictive. Some syntax is required for deferring a particular function rather than the whole class:

```
  translate( shift:Point ): Mytype; deferred;
```

A (provisional) UFO view is that if a class is declared as deferred, then instances of it cannot be created. However, if only certain functions are deferred, instances can be created, and the default implementation for deferred functions is to return an error value of the appropriate type.

2.6.3 Dynamic binding

The standard example continues as follows. Suppose we want to write a function to decide which of two Shapes is bigger:

```
function bigger( s1, s2: Shape ): Bool is s1.area > s2.area
```

If we're allowed to write this, the implementation must work out at runtime which version of area to call. If not, we have to write a different version of bigger for each possible combination of actual Shapes. For reasons discussed above, UFO allows dynamic binding, but with some restrictions.

Implementation note: With single inheritance, there is a fairly simple scheme to implement dynamic binding with small overhead. Each class has a descriptor, which is an array of function pointers. Each object has a pointer to its class descriptor, and the function to be called is selected via a statically determined offset. Meyer claims that Eiffel uses this scheme too, at minimal cost, although it is less obvious how to do it in the presence of multiple inheritance.

For small objects implementing simple data structures, the cost (especially in terms of space and the consequent cost of copying) may still be unacceptable. To avoid this cost, a class may be declared **closed**, e.g.

```
closed class X
% All the usual stuff
endclass
```

Inheritance from a closed class is still allowed, but with the following restrictions:

- Functions of a closed class cannot be deferred.
- If a closed class C inherits from a non-closed class O, or vice-versa, C and O are not type compatible.
- If a function of a closed class C is reimplemented in a descendent class D, C and D are not type compatible.

This ensures that functions on a closed class can always be statically bound, and no time or space overhead is necessary.

2.7 Other data structures

2.7.1 Records and unions

In the area of recursive data structures, functional and object-oriented languages have very different views of the world. FLs typically have algebraic data types, with tuples and union types expressed via constructor functions and usually accessed via pattern matching rather than named selectors (although the latter can be trivially programmed). SISAL has a rather more conventional-looking approach, with record and union types, and explicit selectors. There is a degenerate form of pattern matching, namely the **case** construct. In OOLs, on the other hand, "records" are classes with explicit selectors, (i.e. trivial methods) and the notion of a union type is not explicitly present. The uses of union types in FLs are effectively covered by two completely different mechanisms in OOLs:

- The most common case, a union of a nil value and some tuple (e.g. lists, simple trees) is dealt with by null references to objects.

- Other cases, where a data type may have a number of related forms, are dealt with by inheritance.

The pure functional style has several attractions. Recursive data structures have very simple, clear, concise, natural definitions; an internal data structure for a compiler, for instance, can be defined in a way very similar to a BNF grammar. Pattern matching allows clear, concise descriptions of operations on such a data structure, obtained by case analysis. The trouble is, although this seems to work well as a programming-in-the-small technique, it becomes brittle when used on a larger scale. For instance, in the compiler example, a small change in the grammar may cause widespread changes in the program, because many of the patterns matched against are no longer valid. I've seen this at first hand, and it seems incompatible with the requirements of software maintainability, reusability etc. Indeed, some functional programmers now seem to prefer to introduce selector functions.

The questions for UFO are:

1. What is the equivalent of a record/tuple?

2. Do we have union types as such, and if so what do they look like?

For question 1, adopting the OO approach in full seems appropriate. For instance, the complex number example above declares a "record type" but also packages in the operations on the type.

For a recursive example, consider a binary tree:

```
type AtoB = Alpha -> Beta;

closed class Tree[Alpha](this: Alpha; left,right: Tree[Alpha])
functions
  this:  Alpha;
  left:  Tree[Alpha];
  right: Tree[Alpha];
  map ( f:AtoB ): Tree[Beta];

imp
  store this, left, right;

  function map is
    Tree( f(this),
          if is_null(left)  then left.map( f )  else null:Tree[Beta] fi,
          if is_null(right) then right.map( f ) else null:Tree[Beta] fi
         )
end
```

Note that (unlike, e.g. SISAL records) UFO class definitions can be recursive. To deal with the nil case, we associate with every class type T a value **null**:T. The simplest way to use nulls is to say that any attempt to call a method of **null**:T will generate an error. This deals with e.g. taking the head or tail of an empty list. Explicitly testing for a null can be done with a standard function is_null.

This does, however, lead to rather messy code, as in the map example, since any use of a function like map (including the recursive calls within the definition) has to explicitly check for null. A solution is to allow calls on null objects, provided the function specifies a result for the null case, e.g.

```
function map is
  nullcase:  null: Tree[Beta];
  otherwise:  Tree( f(this), left.map( f ), right.map( f ) )
```

This seems to be worthwhile, as it tidies up map a lot and makes it look more like the usual neat functional version. Obviously, the nullcase can't refer to any instance values!

Implementation note: Note that I have not restricted the use of nullcase to statically bound classes. It is possible to use it with dynamic binding *because nulls are typed.* The implementation trick is to represent **null**:T as a pointer to a "null implementation" of T where all calls except nullcase ones generate errors. This notion of a null implementation may be useful with deferred classes too.

Returning to question 2, given all this do we need union types? This is still unclear, but since there is currently no clear case for including them, they will be left out, at least for the moment.

2.7.2 Lists

A list class can be defined in a similar way to the tree:

```
closed class List[Alpha]( h: Alpha; t: List[Alpha] )
constructors
  List( a: Array[Alpha] );
functions
  hd: Alpha;
  tl: List[Alpha];
  array: Array[Alpha];
  map ( f: AtoB ): List[Beta];
... etc.
operators
  ++ ( self; l: List[Alpha] ): List[Alpha];      % Append
  :: ( v: Alpha; l: List[Alpha] ): List[Alpha];  % Cons
end
```

Collections, such as lists, will typically have a constructor which takes an Array argument. This allows such collections to be specified using the array syntax, e.g.

```
l = List( [1 to 42] )
```

and to be conveniently returned from loops. Likewise, such classes will typically include a function which returns an array of values. So for instance, the map function on lists could be written

```
function map( l: List[Alpha]; f: AtoB ): List[Beta] is
for x in array(l) do y = f(x) return List( y ) od
```

Implementation note: It should be possible for lists and other collections to have alternative representations (e.g. dataflow token streams) where appropriate. This may mean that the definition will have to be frozen, so that the implementation can know about it.

SISAL cognoscenti should note that lists are still strict, so are not the same as SISAL streams. The term "stream" is avoided in UFO, hopefully to help avoid confusion in this area.

2.7.3 Strings

There are several ways of viewing strings in programming languages:

- As array[character] (e.g. Pascal, C, SISAL).
- As list[character] (e.g. most FLs)
- As a special primitive data type, with a set of operations built in. (e.g. ID)
- As an ordinary class, but with some syntactic sugar for literal strings (e.g. Eiffel).

I originally preferred the third option for efficiency reasons, but it now looks like they can be defined as an ordinary class, although as for Lists the interface may have to be frozen for efficiency reasons. It makes sense for strings to be functional things, as they are usually scanned or appended together, rather than selectively updated.[15]

The interface of class String is similar to that for List, although there will be some extra operations. The syntactic sugar is that:

```
"String"
```

```
is equivalent to:
```

```
String( ['S', 't', 'r', 'i', 'n', 'g' ] )
```

[15]Interestingly, in Eiffel strings have the potentially nasty property that assignment of strings is a sharing of references rather than a copy, and the shared string may be updated. We avoid this by keeping strings stateless.

Implementation note: In my experience, programs making heavy use of functional strings (e.g. a compiler) do many appends and therefore are slow. The usual representations of arrays and lists are not optimised for append, and the result can be very expensive. The sort of efficient implementation I have in mind is to represent small strings as packed arrays of Bits and large ones as trees of small ones. An append then just creates a new tree node. As well as making append efficient, this gives some scope for parallelism.

2.8 Functional I/O

The strictness of UFO means that purely functional I/O will not be sufficient to cover all cases. However, as far as possible, reasonable facilities will be provided to encourage the writing of purely functional programs whenever appropriate.

The appropriate set of features is not yet decided. A minimal set of I/O facilities (functional and otherwise), suitable for UFO0, is described later.

2.9 Other features

2.9.1 Null and error values

Every type T has an associated error value, **error**:T. Every class type T has an associated null value **null**:T. Their presence can be tested using the standard functions is_error and is_null.

The error semantics of UFO are derived from SISAL. Roughly, given a function or operator f with result type T, if an error occurs within f (e.g. bound check failure, divide by zero, attempt to access via null value etc.) or if any of the parameters to f are error valued, the result will be **error**:T. There are rather more complicated rules for arrays and loop structures. An implementation may choose not to implement the error semantics, and terminate with a suitable error message at the point the error originally occurs.

Implementation note: As mentioned above, it is useful to implement **null**:T as a pointer to a "null implementation" of T, which implements nullcases and generates suitable error messages on other accesses. On a tagged architecture, error values will probably be distinguished from valid ones by a tag bit. On an untagged machine, proper implementation of error values (especially for scalars) would probably be expensive, hence the dispensation for implementations to opt out if they wish.

2.9.2 Nesting and scope

Nesting of implementations, or of functions, is disallowed. Nested implementations would be against the spirit of the OO approach. Anybody who has written a lot of, for instance, Pascal knows that if you nest your functions you usually end up unnesting them.[16]

When the name of a value appears in an expression, the order of the search for the definition is as follows:

[16] In other words, they are more bother than they are Wirth...

1. Local value definitions, within the immediately enclosing block or loop body, if any.
2. Value definitions in each enclosing block in turn, if any.
3. Parameters of the enclosing function.
4. Instance values and functions of the enclosing class.
5. Methods of other classes. This may produce multiple answers due to overloading, which is resolved using type information.

For instance, in the map function:

```
function map( a: Array[Alpha]; f: AtoB ): Array[Beta]
is
for xs in a do ys = f(xs) return ys od
```

xs may happen to be an instance of a class with a method f, but the parameter f takes precedence.

2.9.3 Identifiers and namespaces

Identifiers have the usual form. Case is significant. Type (or class) names and value names occupy the same namespace so must have disjoint names. A suggested convention is for type and class names to start with an uppercase letter, and function/value names to be all lowercase with underscores separating words.

3 Arrays

Readers who have no special interest in array computations may wish to skip this section on first reading, as the features it describes are orthogonal to those covered in the rest of the document.

In UFO, expressions returning an array determine both the dimensionality and the bounds of that array value. These are not part of the type, as in some languages, but are properties of the particular value of type **Array**[T]. Each dimension of an array has an upper and a lower bound, both of type Int, and these may be supplied as expressions. Note that arrays are rectangular, and that a 2-D array containing values of type T has the overall type **Array**[T], and not **Array**[**Array**[T]].[17]

[17] The latter can be used for ragged arrays, triangular matrices, skyline matrices, and suchlike.

3.1 Defining values of type Array[T]

The general syntax for array constructors is

```
arrayvalue : '[' [bounds '|'] [arraycontents] ']'

bounds : bound ( ',' bound )*

bound :  (Int)expression [ '..' (Int)expression]

arraycontents : expression (',' expression)*
              | (Int)expression 'to' (Int)expression
                     ['step' (Int)expression]
              | 'all' expression
```

The array contents provide the values for the elements of the array in the form of a sequence, or an arithmetic progression, or as a single value repeated as many times as necessary. These values are used to fill the array whose shape is described by the bounds part. The step in the arithmetic progression may be positive or negative, and defaults to 1 if omitted. The progression will be empty if the step is 0 or has the wrong sign for progression towards the final value, e.g.

```
a = [3 | 1, 2, 3, 4, 5]     % lowerbound 3, upperbound 7
b = [3..7 | 1 to 5 ]        % same as a
c = [5 | 0 to 10 step 2 ]   % 6 values
d = [1..10 | all 0 ]        % 10 zeros
e = [1..10 | 0 ]            % Illegal
```

Each comma in the bounds part increases the dimensionality of the array, starting from 1-D when no commas are present. Each dimension has a lower bound and an upper bound, written before and after '..' respectively. If the upper bound is omitted (together with its '..'), it defaults to have the same value as the lower bound, except for the rightmost upper bound in the bounds, which defaults to match the number of elements provided. Where upper bounds are present, the number of elements provided must exactly match the bounds given.

Multi-dimensional arrays are filled in "index order", i.e. "by row", which means with later subscripts changing more rapidly than earlier ones.

For example :

```
a = [1..3, 1..2 | 1 to 6 ]      % First row is [1,2] etc.
b = [1..3, 1 | 1 to 6 ]         % Same as a
c = [2, 3 | 42 ]                % A 2D array of one element
d = [2, 3 | 1 to 10 ]           % 10 elements, all in row 2.
```

3.2 Loops and Arrays

Another way of producing an array value is as the result of a loop. In the **return** clause of a loop, any value of type T defined in the loop is a value of type **Array**[T] and contains all the values held by the loop value during the loop. There are two forms of loop, namely **while** loops and **for** loops.

The syntax of a **while** loop is :

```
whileloop :    ['initially' valdefs] 'while' (Bool)expression
                 'do' statements 'return' expression 'od'
```

while loops test the value of the Boolean expression at the start, and this is preceded by an **initially** clause which gives initial values to identifiers used in the test and in the body of the loop (the statements between **do** and **return**). Identifiers initialised in this clause (and only such identifiers) can have values for the next iteration provided for them by preceding them by **new**. This new value is also accessible in expressions in the body of the loop, again by using the prefix **new**. Identifiers assume their new values immediately before executing the test, so neither the test condition nor the **return** clause, which is considered following the failure of the test, may contain the keyword **new**. For example :

```
a)        a = initially x = 1
                while x < 10 do
                new x = x + 1
                return x
                od
```

has the effect of setting a to [0 | 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. Note that both the initial and final values of x are included, and that the lower bound is 0, corresponding to the initial value. If a value is returned which was not mentioned in the **initially** clause, the resulting array has lower bound 1. E.g.

```
b)        b, c = initially x = 1
                while x < 5 do
                y = x * x
                new x = x + 1
                z = new x
                return (y, z)
                od
```

has the effect of setting b to [1 | 1, 4, 9, 16], and c to [1 | 2, 3, 4, 5].

The syntax of **for** loops is

```
forloop : 'for' generators 'do' valdefs 'return' expression 'od'

generators : generator | generator 'join' generators
```

```
                        | generator 'cross' generators
                        | generator 'dot' generators

generator : identifier 'in' (Array)expression
```

for loops use one or more loop control identifiers, each with a corresponding generating expression. A generating expression is an array, and the loop control identifier takes each of the values in the array within an evaluation of the loop body. **while** loops always return 1-D arrays, but, depending on the generators used, **for** loops can return arrays of any dimensionality. Thus

```
c)        a = for id1 in gen do
                ...
                id2 = exp : T
                return id2
                od
```

produces an **Array**[T] with the same dimensionality as the generator and with the same bounds. When more than one loop control identifier (and corresponding generator) is present, they are combined using one of three mechanisms, viz. **join**, **dot**, and **cross**.

```
d)    for val1 in gen1 join  val2 in gen2 do
        ...
        id = exp : T
        return id
        od
```

produces a 1-D **Array**[T] with the same lower bound as gen1. The body is evaluated with val1 taking each value from gen1 and, for each value of val1, val2 takes all possible values from val2. For example :

```
for i in [1 to 4] join j in [i to 4] do
   k = i+j
   return k
   od
```

produces the Array [2, 3, 4, 5, 4, 5, 6, 6, 7, 8].
Notice that val1 can be used in gen2. The result is not multi-dimensional because it may not be rectangular.

```
e)          for val1 in gen1 cross val2 in gen2 do
              ...
              id = exp : T
              return id
              od
```

produces an **Array**[T] of dimensionality equal to the sum of the dimensionalities of the generators, and with bounds corresponding first to those of gen1 and then to those of gen2. val1 cannot be used in gen2, and hence the result is rectangular. The evaluation of the loop body occurs for val1 taking each possible value from its generator, and, for each of these, val2 taking each value from its generator.

Thus the value [lb1 .. ub1, lb2 .. ub2 | **all** exp] can also be produced by :

```
f)    for i in [lb1 | lb1 .. ub1] cross
          j in [lb2 | lb2 .. ub2] do
        el = exp
        return el
        od
```

given that the expression cannot have referred to the indices. However this can also be used in the more general case where the expression depends on i and/or j.

```
g)         for val1 in gen1 dot val2 in gen2 do
        ...
        id = exp : T
        return id
        od
```

produces an **Array**[T] of dimensionality equal to the dimensionalities of the generators (which must be the same), and with bounds corresponding to the intersection of those of the two generators. The evaluation of the loop body only occurs for values of val1 and val2 in positions within this intersection of the generators. Thus

```
h)        for i in [0 | 1, 2, 3, 4] dot
                  j in [10, 20, 30, 40] do
                k = i + j
                return k
                od
```

produces an array [12, 23, 34].

i) When the array produced by the loop is 1-D, it is possible to use a form of loop which is a hybrid between a **while** loop and a **for** loop. This uses generators, like the **for** loop, but is preceded by an **initially** clause, and values defined in that may be redefined using **new**, as in the **while** loop. For example, the following sets val to the sum of the elements in a, where a is a 1-D **Array**[Int] :

```
        val = initially ans = 0
                for el in a do
                new ans = ans + el
                return last(a)
                od
```

where last is a function which returns the last element of a 1-D array.

3.3 Operations on Arrays

a) Standard functions on arrays

There are interrogation functions on arrays, all of which return integers:

```
lower(a,dim)     - lower bound of a in dimension dim
upper(a,dim)     - upper bound of a in dimension dim
dimsize(a,dim)   - size of a in dimension dim
                    i.e. upper(a,dim) - lower(a,dim) + 1
size(a)          - total number of elements in a
dim(a)           - number of dimensions in a
```

There are two functions which return arrays derived from array values, but with different bounds :

```
set_lower(a, dim, newlb)
     - returns an array with the lower bound in dimension
       dim changed to newlb (and the upper bound correspondingly)

set_upper(a, dim, newub)
     - returns an array with the upper bound in dimension
       dim changed to newub (and the lower bound correspondingly)
```

b) Slicing and subscripting :

Subscripting an **Array**[T] produces a value of type T - whatever the dimensionality of the array. Thus if the number of subscripts is not equal to the dimensionality an **error** : T is produced. To create an array from part of an array, a range may be used instead of a subscript, producing what is known as a "slice" of the original array. Syntactically, subscripting is just a special case of slicing.

The syntax of a slice generator is :

```
slice : (Array)expression '[' slicebounds ']'

slicebounds : slicebound (',' slicebound )*

slicebound : (Int)expression
           | [(Int)expression] '..' [(Int)expression]
```

The number of slicebounds must be equal to the dimensionality of the source array. If there are any dot-pairs in the construct, the result is an array, with dimensionality equal to the number of dot-pairs, and otherwise the result is a scalar. A slicebound without a dot-pair is an indexing of the source array in that dimension, and the index must be within the relevant bounds. Similarly both lower and upper values, i.e. those given either side of a dot-pair, must be within bounds, and will form the new bounds of the resulting array. Either (or both) values may be omitted, in which case they default to lower or upper bounds as appropriate.

Thus, if b is a 1-D array:

```
b[i]              % Ordinary indexing, returning the (i+1-lower(b,1))th
                  % element of b, if in range, error otherwise.
b[lb..ub]         % 1-D array with bounds lb..ub and values from b
b[..upper(b,1)-1] % a copy of b with the top element removed.
```

If a is a 2-D array:

```
a[i,j]  % Ordinary indexing.
a[i,..] % 1-D array with bounds [lower(a,2)..upper(a,2)] and values from
        % the i'th row of a.
a[..,i] % 1-D array with bounds [lower(a,1)..upper(a,1)] and values from
        % the i'th column of a
```

The last function mentioned earlier is simply

```
function last( a: Array[T] ) : T is a[size(a)]
```

d) There is a built-in function :

```
index(a : Array[T], i : Int) : T
```

which returns the element of a at index position i, whatever the bounds or dimensionality of a. It therefore defines an overall index ordering on an array, and should not be confused with the ordinary indexing operation shown above. Index positions start at 1, and increase in steps of 1 up to size(a). As mentioned when describing array constructors, in the ordering of the elements the later subscripts increase more rapidly than earlier ones.

A related function:

```
flatten(a : Array[T]) : Array[T]
```

returns the 1-D array with lower_bound 1 containing all the elements of a in index ordering, whatever the shape of a.

e) There is an infix operator ++ between arrays which generates an array by concatenation. Viz.

```
a ++ b
```

is a 1-D array with lower_bound = lower_bound(a,1), and upper_bound = upper_bound(a) + size(b). If the arrays are not 1-D, they are flattened first.

f) There is an infix operator # between arrays which generates an array by substitution. Viz.

```
a # b
```

The dimensionalities of the arrays a and b must match, and the bounds of b must be included within those of a, otherwise an error is generated. For example

```
a # [2, 2 | 5]
```

produces a 2-D **Array**[Int] with bounds and elements the same as those of a, (which must also be a 2-D **Array**[Int]), except in position [2,2], where 5 is used.

```
[0..10, 0..10 | all 0 ] # [1..9, 1..9 | all 1 ]
```

produces a 2D array with 1s in the middle and 0s around the edges.

g) There is a built-in function :

```
shaped_like(a : Array[T]; b : Array[U]) : Array[T]
```

the result of which is an array with the dimensionality and bounds of b, but with the values (in index order) of a. The number of elements in a must exactly match those in b, or an error is generated.

E.g. to get a 2-D array from an **Array**[**Array**[T]], a, which is known to be rectangular, where a and all its elements are 1-D :

```
shaped_like(Catenate(a),
   [lower(a,1)..upper(a,1), lower(a[1],1)..upper(a[1],1)|]
```

where Catenate is a function defined as

```
function Catenate(a : Array[Array[T]]) : Array[T] is
initially ans = [] : Array[T]
for row in a do
new ans = ans ++ row
return ans[upper(a,1)]
od
```

which produces a 1-D array concatenating the elements of a.

Another example of the use of shaped_like is to produce an array with additional dimensions, for use with # (see above). Thus a 1-D array, b, can be turned into a 2-D array, using

```
shaped_like(b, [5,1..total_size(b)|])
```

which can then be used to overwrite the 5th row of a 2-D array, a, as in :

```
a # shaped_like(b, [5,1..total_size(b)|])
```

Overwriting the 5th column of a instead would be :

```
a # shaped_like(b, [1..total_size(b),5|])
```

4 Interlude

Before we jump over the cliff in the next section, it's worth reflecting that functional UFO is quite a respectable language in its own right. It has most of the features of SISAL (a language in which quite a lot of numeric code has been written) and most of the features of, say Hope+ (a language in which quite a lot of symbolic code has been written) together with better facilities for data abstraction etc. than either of them. It seems reasonable to claim, therefore, that use of stateful objects, as described below, should be very much the exception rather than the rule.

Most of the concepts of object-oriented programming work perfectly well in a pure functional context. An interesting line of research is therefore to push this fusion as far as possible. This would involve introducing lazy evaluation, runtime function creation etc., and also multiple inheritance. This could be done by adding these features to functional UFO, but it would probably be better to start from Haskell.

However, we believe that pure FLs will always be too restrictive for widespread general-purpose use. The real world is full of state, working perfectly well in parallel. So here we go...

5 State

5.1 Objects with assignment and state

A counter class can be defined as follows:

```
stateful class Counter( initial: Int )
functions
   read: Int;
procs
   inc( n: Int  ): Int;

imp
state
   i := initial

% Increment the counter and return the new value.
proc inc is { i := i + n fby return i };
function read is i;
end % Counter
```

The keyword **stateful** warns the user that a Counter is not just a constant value, and tells the implementation to check various restrictions on its use (described below) and to avoid certain optimisations.

A Counter has an *instance variable*, i, which is initialised and modified by assignment, denoted ':='. The initialisation is compulsory. An instance variable may only be assigned-to within a **proc** (procedural method) of the class. The body of a **proc** is an expression. Statements (such as assignments) are incorporated into expressions by the block construct, the full form of which is:

```
block : '{' statements [['fby'] 'return' expression] '}'
```

Statements within a block are separated either by semicolons or by the keyword **fby** (followed by)[18]. Statements are still sequenced only by data dependency except that if two statements are separated by **fby** they are executed sequentially[19]. The result of the block is the expression following **return**, if present. The functional block construct is simply the special case where all the statements are value definitions and a result is always returned.

Procs and functions will collectively be referred to as *methods*.

An object with state behaves like a (dynamically created) monitor; it guarantees exclusive access to its instance variables. So if two calls are made in parallel to inc, or one to inc and one to read, they will be serialised in some undefined order (in the second case, the result will therefore be nondeterministic). However, the implementation may allow multiple calls to read in parallel.

The counter might be used as follows:

```
function funnyadd( a, b: Int):Int is
{
  { c = Counter( 0 ); c!inc(a); c!inc(b) }
  fby
return c.read
}
```

A stateful object can be created and bound to a value name in the same way as a purely functional one. The syntax for proc calling is the same as (the dot version of) function calling, except that the dot is replaced by a pling. The following constructs may appear either as expressions or as statements:

1. A **proc** call.
2. The block construct.
3. The loop construct.
4. The **if** ... **then** construct.

Otherwise, there is no overlap between expressions and statements. When used as statements, a block or loop need not necessarily have a **return** part, or **if** an **else** part.

[18] This use of **fby** is totally unrelated to the use of the same keyword in "Lucid The Dataflow Programming Language".

[19] In fact, even in this case, the sequencing constraint is quite weak - see later

Such constructs notionally return a value of type **Void**, as does a **proc** declared with no result type. However void values cannot be used in value definitions or assignments, or passed as parameters, or returned from functions.

The two calls to inc may be made in parallel, and may be serviced in either order. However, the **fby** ensures that the read will occur after both incs have been done.

Note that funnyadd is a perfectly good, referentially transparent, function; it always returns a + b. Although it manipulates state internally, there is no way for that state to "persist" between calls. Furthermore, most of the properties which make this so can be determined statically, merely from the fact that the parameter and result types contain no state.[20] Identifying cases where state can be completely encapsulated in this way is important, because it gives conceptual clarity for the user of the function and freedom for the implementation to perform various optimisations.

Although funnyadd is a silly example, it is easy to find more sensible ones, e.g.

```
function fancysort( unsorted: Array[Real] ): Array[Real] is
% Fancy super-efficient quicksort using in-place update

or..
function compile( sourceprog: String ): String is
% Object-oriented compiler
```

Also note that the serialisation of access to stateful objects means that deadlock can easily occur[21], for instance if a proc in object A calls a proc in object B which calls a proc (or function) in A again. In general, this has to be the responsibility of the programmer, language restrictions strong enough to prevent it would have to be extremely severe.

5.2 References

Two questions raised by the introduction of state are as follows:

1. What does value definition, A = B, or assignment, A := B, mean when A and B are stateful? Is B copied, or does A merely become another reference to it? (Of course in the functional case, we don't care, but now we certainly do.)

2. Do we have a notion of explicit pointers, i.e. where something of type pointer-to-x is different from something of type x?

The proposed answers are the same as in Eiffel, namely:

1. Value definition, A = B, or assignment, A := B, of non-scalar values *copy references only.*

[20] The one bit which can't is that it might have been nondeterministic; this is discussed later

[21] In fact, the weak sequencing rules described later mean that deadlock is harder to generate than the above would suggest. UFO does not require the implementation to detect deadlock, although some such mechanism would obviously be a useful optional facility.

2. Explicit pointers are not necessary.

The following implements a FIFO queue.

```
% A one-way link. Note that only the next field is updatable.
closed stateful class Link[Alpha]( v: Alpha )
functions
  first: Alpha;
  next: Link[Alpha];
procs
  set_next( y: Link[Alpha] );

imp
  first = v
state
   next := null:Link[Alpha]

proc set_next is { next := y };
end % class Link

% The queue class itself.
stateful class Queue[Alpha]
functions
 queue_head: Alpha;         % Return the first value in the Q.
 queue_length: Int;     % Return current length of the Q.
procs
  enqueue( val:Alpha );     % Queue the value at the back
  dequeue: Alpha;           % Dequeue the value from the front of the Q.
end % class Queue

imp Queue
state
  head := null:Link[Alpha]; tail := null:Link[Alpha]

proc enqueue( val:Alpha ) is
{
  newlink = Link( val );
  if is_null(head)
  then
    { head := newlink; tail := newlink }
  else
    { tail!set_next(newlink) fby tail := newlink }
  fi
}
```

```
proc dequeue: Alpha is
{
  result =
  if is_null(head)
  then error:Alpha
  else
    {
      p = head fby
      head := head.next fby
      if is_null(head) then { tail := null:Link[Alpha] } fi
    return
      p.this
    }
  fi
return result
}

function queue_head: Alpha is
if is_null(head)
then error:Alpha
else head.first
fi

function queue_length:integer is
initially
i = 0;
p = head
while not is_null(p) do
  new i = i + 1;
  new p = p.next
return last(i)
od

end % Queue
```

This illustrates a number of points:

- Link is specified to be a closed class. It should be no less efficient than a C struct.
- Null values and the is_null test are used pretty much as in the functional case.
- Only the instance variables head and tail can be assigned to. Other values within the implementation of Queue, such as newlink, are just ordinary local values defined by value definition in the usual way.

- The rhs of the definition of newlink constructs a new Link. All the other assignments and value definitions of Links just move references around.
- When used as a statement, an **if** construct need not have an **else** part, if the **then** part returns a **Void** result. However, the **then** part is still syntactically an expression, hence the enclosing block in the example in dequeue.

5.3 Sequencing and termination rules

In functional UFO, constructs such as **if** ...**then** and blocks are simply expressions. In full UFO they also act as control constructs, and have an operational aspect[22]

Defining the semantics of this operational aspect is tricky, because we have to balance several factors:

- The rules need to be simple so programmers can understand them.
- They should restrict parallelism as little as possible.
- The implementation should be able to choose whether to spawn parallel tasks or not. In particular, serial execution should be efficient.
- As far as possible, we want to avoid situations where programs deadlock, or are nondeterministic, under some evaluation orders but not others.

The basic rules are as follows:

1. The queuing for access to a stateful object is **FIFO**.
2. A method call (proc or function) on a stateful object is *safe* **as soon as it has been queued**.
3. In general, a construct is safe as soon as all method calls it makes are safe, and all assignments to, or uses of, instance variables of the enclosing object have been completed.
4. The order of execution of statements in blocks and loop bodies is constrained only by data dependencies between the value names[23], except that if two statements are separated by **fby**, the second statement does not begin execution until the first is safe.
5. All method calls (notionally) create separate, asynchronous, parallel processes.
6. A method releases exclusive access on a stateful object as soon as it is safe.

[22]An analogy is the declarative and procedural meaning of Prolog programs.
[23]NB: Not the instance *variables*!

Implementation note: FIFO was a likely implementation anyway, because although it is slightly less efficient than LIFO, it probably yields more parallelism. On a machine where messages are guaranteed to arrive in the order they were sent, a call is safe as soon as the message is sent. In most cases, the implementation can decide on how much of the available parallelism to exploit. The requirement that method calls behave as asynchronous parallel processes requires that even a serial implementation will have to provide pseudo-parallel execution in some cases, to prevent unnecessary deadlock. Such cases are expected to be relatively rare (they typically involve use of **wait**, described later).

For instance, in the funnyadd example:

```
function funnyadd( a, b: Int):Int is
{
  { c = Counter( 0 ); c!inc(a); c!inc(b) }
 fby
return c.read
}
```

the two incs are queued in some indeterminate order. The **fby** ensures that the read queues after them, but the actual synchronisation is done in the queue on the counter, not by the code in funnyadd.

In the queue example:

```
    { tail!set_next(newlink) fby tail := newlink }
```

the **fby** ensures that the set_next message is queued at the object referred to by tail before the reference is reassigned.

These rules have a number of interesting consequences. They give UFO may of the properties of a concurrent object-oriented language such as ABCL.

- The sequencing is very weak. Writing S1 **fby** S2 only means S1 is safe when S2 starts executing; S1 need not have terminated (in fact it need not ever terminate). To get stronger sequencing, it is necessary for S1 to return a value needed by S2, or to explicitly synchronise using **wait** (described later).
- A method may continue executing after it has returned a result.
- A method may release the lock on an object before it returns a result. Suppose a method x!mess1 ends with **return** y!mess2. As soon as mess2 is queued, (and any other safety criteria are met) the lock is released, and the task of returning the result to the original caller is effectively *delegated* to y. This avoids deadlock in a number of cases.

An example of UFO programming in concurrent OOP style appears in appendix A.

5.4 When is a function not a function?

A crucial issue in the design of UFO is the question of the precise distinction between procs and functions. The presence of stateful objects makes the distinction quite subtle in general, but it is important to ensure that functions really "behave like functions":

- For conceptual clarity of programs. If something is called a function, a user should be able to use it under the assumption that it is exactly that.
- For purposes of optimisation.

Something "behaves like a function" if the user can assume:

- No sequencing constraints; the function can be used in arbitrary expressions and hence be evaluated in parallel with other functions.
- No side effects.
- Referential transparency; if y = f(x), a reference to y and another call of f(x) are identical if the xs are identical.

Roughly, there seem to be 3 serious possibilities:

1. Allow the programmer to decide when to write **function**, rather than **proc** (apart from a few basic restrictions such as disallowing assignments within function bodies). The implementation then has to decide which ones are really functions for optimisation purposes.
2. Restrict 'function' to mean only SISAL-like functions, i.e. no state manipulations anywhere in the call graph.
3. Develop a set of (static) rules to identify functions which allow state manipulations internally, but which behave like pure functions to the user.

The first approach is ok from the implementation point of view (although it may result in unnecessarily conservative implementations), but is likely to cause problems because a user of a "function" cannot rely on it actually being one. The second is very restrictive; something with internal state cannot be a function even if that internal state is completely invisible to any user.

So, let's explore the third option. First some definitions:

- A class (or object of that class) has *explicit state* if it has instance variables.
- A class (or object) has *implicit state* if it has instance values of a stateful type.
- A class (or object) is *stateful* if it has explicit or implicit state, and *stateless* otherwise.

- A structured type (or value of the type) other than a class (e.g. an array) is stateful if it has a component type which is stateful, and stateless otherwise.

UFO objects provide controlled access to explicit state, but this is only a partial solution. State is very pervasive. Implicit state anywhere in a data structure means that, for instance, a reference to the structure behaves differently from a copy of the structure.

A method can fail to be a function because:

- It causes some externally-visible side-effect.
- Its result contains variables (because then a reference to the result and a copy of it are different things).
- It is nondeterministic (in terms of the result it produces, or its termination properties, or both).

Let's try to formulate a weak, statically checkable, set of restrictions which allow something to be called a function.

Let f(a1..ak) be a method of class C, with argument types t1..tk, and result r of type tr. Given a value definition

```
x = c.f(a1..ak)
```

then we want y and z to be identical in:

```
y = c.f(a1..ak);  z = x
```

We have to assume that a1..an are not being modified in parallel with the calls; as shorthand for this we will say a1..an are *fixed*. The full set of inputs to f includes the instance variables and instance values of c. However, the instance values behave like extra parameters, so we can represent the call as

```
f(v1..vm,a1..an)
```

where v1..vm are the instance variables and a1..an are the explicit parameters plus the instance variables.

First attempt:

R1: Assignment restriction. f does not assign to any instance variable of C.

R2: Output restriction. r is stateless

R3: Input restriction. a1..an are also stateless.

R4: Determinism restriction. For a particular fixed a1..an, f always produces the same r.

R5: Termination restriction. For a particular fixed a1..an, f always terminates or always fails to terminate.

R1 is clearly necessary and easily checkable (you need to check for assignments in other methods called locally by f, as well as the body of f itself, otherwise trivial).

R2 is also clearly necessary. If r has state, y is a separate object from x, with its own state, whereas z is another reference to x. It is easy to check provided that we can tell statically whether tr is stateful or stateless. This involves some restrictions on genericity and inheritance, discussed below.

Note that R2 implies that the constructor for a stateful class is a proc, not a function.

R3 is where things get tricky. It's easy to check, but it seems too strong. It's quite reasonable for a1..an to have state, so long as f (or anything it calls) doesn't alter that state. So what we really want is:

R3': Input modification restriction. Any state in a1..an is not modified by f.

In other words, a1..an are "read only". The trouble is, this is undecidable statically (due to aliasing etc.) and expensive to determine even at runtime. The extent to which we can get close to R3' statically depends on the sophistication of the implementation. A first step is to check all assignments in the call graph of f against the types t1..tn. If there are no assignments to values of these types, R3' must be satisfied. No doubt closer approximations can be achieved, e.g. by abstract interpretation.

An engineering compromise is required. The following are possibilities:

1. Stick with the original R3, and check it statically.[24]

2. Use R3', check as much as possible statically, but leave the ultimate responsibility with the programmer.

3. Use and enforce R3'. A full UFO implementation will therefore (have an option to) plant code to check at runtime of it can't prove statically that the restriction is satisfied.

I currently favour the third option. It gives the input modification restriction the same sort of status as array bounds checking. Although it's obviously much more complicated than that, we can also do a lot more statically than is usually done with array bounds.

R4 is too restrictive. There are cases, such as nondeterminsitic merges, and set operations, where the concrete result of a method may vary nondeterminsitically but its meaning is not affected. Such methods should be allowed as functions if they satisfy the other restrictions. Of course the writer of a nondeterministic "function" has a duty to document it carefully!

On the other hand, nondeterministic termination is clearly unacceptable for a function, and so R5 is necessary. Since it can't be checked statically (or dynamically in any sensible way as far as I can see) it has to be the responsibility of the programmer.

[24]Actually, a slightly stronger restriction is needed. v1..vm should also be stateless internally, as otherwise a really malicious "function" writer, although not allowed to assign to one of them, could take a reference to one and hack it internally.

5.4.1 Restrictions on genericity and inheritance

The restrictions above mean that it is necessary to tell statically, at least in some circumstances, whether a particular value is stateful or not. This obviously has an impact on the forms of polymorphism we allow.

Taking genericity first, generic *inputs* to a function present no problem as the function has no way to modify them anyway. Generic outputs are a potential problem as they may violate the output restriction. However, it is the *caller* who is violating the restriction, so this probably has the status of a warning (the restrictions are, after all, only there to protect the caller). Usually this can easily be checked; if an actual generic parameter used in the output type is (or could be) stateful, a warning is generated.

Inheritance is a bit trickier. Suppose class Pure is stateless, but class Dirty, which inherits from Pure, adds state, either explicitly via instance variables, or implicitly by adding stateful instance values. Now an object of class Pure might actually be a Dirty. We cannot allow code using Pure to become illegal (or worse, legal but incorrect!) just because somebody inherited from it. Note that it's *not* a dynamic binding problem; even if Pure is a closed class, we can no longer, for instance, ignore the difference between a reference to a Pure and a copy of it, as code using Pures was previously allowed to do. Worse, all classes with Pure instance values are similarly affected!

The simplest solution is to ban this sort of inheritance altogether. Stateful and stateless objects would then live in separate "worlds". This is very restrictive, and fortunately a weaker restriction works. Inheritance of this form is allowed, but the child is not type-compatible with its parent, so, for instance, if P is a Pure and D is a Dirty, the assignment P := D or value definition P = D are not allowed. We therefore get some of the benefits of inheritance without being able to mix up stateless and stateful objects in a dangerous manner.

A similar problem occurs with generic type parameters to classes. List[T] is supposed to be a pure class - but what about List[Counter]? A solution is to disallow such types, but generate very similar ones by inheritance:

```
stateful class SList[T]: inherits List[T]
end class
```

SList is identical to List, except that it is stateful. The previous restriction ensures that Lists and SLists cannot be mixed up, as they are not type compatible.

5.5 Flags and Flagged values

5.5.1 Flags

One thing which is missing from the above examples is the notion of waiting for an event to happen, as occurs in I-stores or semaphores ("conditional synchronisation" in traditional concurrent programming terminology). To achieve this, we introduce *flags*. A flag roughly corresponds to a presence bit in an I-store.

A value of type Flag can appear only as an instance variable, and therefore must be confined within the implementation of a stateful class. The only operations allowed on a Flag are:

- Assignment of one of two values, **set** or **clear**.
- The builtin procedure f!wait. If f is set, this does nothing. If it isn't, the process which called the wait is suspended, until it becomes set. In order to allow it to be set, the exclusive access on the object is revoked. When the procedure setting the flag is safe, the waiting one is restarted. If more than one process is waiting, an arbitrary one runs, and the others wait until the chosen procedure is safe. Note that if it resets the flag, the others will continue to wait.

For instance, a general semaphore can be implemented as follows:

```
closed stateful class Semaphore( n:unsigned )
procs
  sem_signal;
  sem_wait;

imp
state
 count := n;
 q :Flag := if n==0 then clear else set fi ;

proc sem_signal is
{
  n := n + 1 ;
  if n==1 then { q := set } fi
}

proc sem_wait is
if n>0
then { n := n - 1 }
else { q!wait fby q := clear }
fi

end % class Semaphore
```

5.5.2 Flagged values

The following defines a class which attaches a flag to a data value of arbitrary type:

```
closed stateful class Flagged[T]
functions
```

```
  read: T;              % Read when value becomes available.
procs
  write( v:T );         % Write, but only if no value currently present.
  overwrite( v:T );     % Write, whatever
  extract: T;           % Destructive read.

imp
state
  flag: Flag := clear;
  val: T := error:T

function read is
{ flag!wait fby return(val) }

proc write is
if flag==clear
then { flag := set ; val := v }
else fatal_error( "Write to flagged value with PB set!" )
fi

proc overwrite is { flag := set ; val := v }

proc extract is
{
  flag!wait fby
  flag := clear  fby
  return val
}

end % Flagged
```

Readers familiar with dataflow will recognise Flagged as describing exactly the operation of an I-store location. The intention is that Flagged will be a builtin type. On machines with I-stores a Flagged value will be a single word, and the hardware will handle everything. On other machines, the implementation above can be used, although it will be much less efficient. Note that the underlying system has to provide a single primitive, wait.

5.5.3 Non-strict evaluation

Flagged values can be used to simulate non-strict evaluation. For instance, the following is a solution to the classic "wavefront" problem:

```
type Nonstrict_array = Array[Flagged[Real]];
```

```
type func = ( Real, Real, Real ) -> Real;

function wavefront( m,n: Int; initial: Real; f: func ): Array[Real]
is
{
% Initialise the array
a : Nonstrict_array =
  for i in [1 to m] cross j in [1 to n] do
  emptyij: Flagged[Real] = Flagged;
  return emptyij
  od
return
% Do the wavefront, and convert to an ordinary array for function result
  for i in [1 to m] cross j in [1 to n] do
  if i == 1 or j == 1
  then { a[i,j]!write( initial ) }
  else { a[i,j]!write
  ( f(a[i,j-1].read, a[i-1,j-1].read, a[i-1,j].read)) }
  fi;
  aij: Real = a[i,j].read
  return aij
  od
}
```

(Note that, despite everything which goes on internally, wavefront trivially obeys restrictions R1-R3 above. As the programmer, I only have to ensure determinism and termination, which in this case is a matter of getting the indices right.)

However, this very fine-grain use of Flagged values is unlikely to be acceptably efficient except on machines with appropriate hardware support. On other machines, it is probably better to use flags only for more traditional concurrent programming style problems, and to tackle problems such as the above by rewriting the algorithm.

6 Other features

6.1 I/O

There is a considerable amount of work still to do on I/O. The following proposes some minimal I/O facilities (suitable for UFO0).

A program is a function or proc, whose input and output types are scalars or strings. (This restriction needs to be lifted for UFO1).

Functional file input is provided by a class RFile (Readonly File):

```
class RFile ( name: String )
functions
```

```
  ok: Bool;
  contents: String;
endclass
```

Creating an RFile attempts to open the file with the given name. If creation succeeds, ok will be TRUE, and contents will be a string consisting of the contents of the file, otherwise ok will be FALSE and contents will be **error**:String.

Procedural I/O is provided by a more general class File:

```
stateful class File ( name: String; mode: Char ) inherits RFile;
functions
  eof: Bool;
procs
  write( s: String );
  getc: Char;
  ungetc;
  readline: String;
endclass
```

The mode will of course be r or w, and ok and contents work as before. Various sorts of read operations are possible, but it seems best to provide character-by-character and line-by-line input, and let the fancy stuff be functions on Strings, at least for UFO0.

6.2 Clusters

This section outlines my current ideas on the higher-level structure of UFO programs, and what the notion of a complete "program" is. These ideas are still tentative.

The Eiffel story is as follows: A "system" is a runtime entity which is constructed by assembling a set of classes from some "universe", i.e. all the classes available in some situation, and designating one of them as the "root". Classes are therefore the only things which are global. "Clusters" are informal groupings of closely related classes. A system is put together, and naming conflicts are resolved, by the environment software. Classes within a cluster are not allowed to have the same name. There is no language feature corresponding to the notion of a cluster - it's purely to do with the environment.

The current UFO proposal is to have a notion of "clusters", which is similar to the extent that it is a related group of things used in constructing systems. However, this is a real language feature, which looks like the following:

```
'cluster' identifier
['uses' identifiers]
'export' identifiers
definitions
'end'
```

where the identifiers in the **uses** part are the names of clusters, and those in the **export** part are the names of things defined in the definitions. The extent to which a cluster is a completely self-contained reusable component is determined by what's in the **uses** part. Probably some standard libraries should be assumed by default. A cluster will consist of one or more files, and the specification of a system will consist of a list of clusters, and for each cluster a list of the files which hold it.

The interesting question is what sort of things can be defined (and exported) in the defintions. The Eiffel answer is classes only, but we can make (progressively weaker) arguments for other things as follows:

Types. We have types which are not classes (e.g. function types). We also need to be able to define type variables (probably - they are implicit in Eiffel, but I don't think I like that), although they don't need to be exported.

Functions. I think there's clearly a case for functions which aren't methods of classes, and for being able to construct libraries of such things, e.g.

```
cluster differential_equations
uses real_maths
export predictor_corrector, runge_kutta, ...
```

Procs. There may occasionally be a need to use procs outside classes. The sort of thing I'm thinking of are things which break the input modification restriction in subtle ways (e.g. by doing File I/O) or things which break the output restriction (e.g. by returning a File descriptor).

The other question is what constitutes the "root" or "main program". The current suggestion is that it should be any function or proc, with appropriate input and output types, exported by any cluster. The actual root is nominated when the system is constructed.

7 Further work

As stated at the beginning, this document is not a complete description of the language. The following are the main points to be addressed in producing a complete description:

1. The type and inheritance mechanisms need to be more fully defined, and their relationship clarified.

2. Various standard functions applicable to all objects should be considered, e.g. equality, shallow copy, deep copy.

3. Full I/O facilities (functional and otherwise) need to be defined. We should consider whether we want to have some notion of persistent objects.

4. Notions like what constitutes a complete program, how classes map to files, and what the environment looks like need to be sorted out.

5. The error semantics needs to be properly defined (or thrown away!). Error handling in general needs more attention.

6. The various restrictions and semantic issues to do with stateful objects and functions need to be gone over very thoroughly.

7. The issue about union types and case statements has not yet been resolved.

8. The array features need more work, and in particular the dynamic shaping of arrays needs critical assessment.

9. A search should be made for other minor features which should be included.

UFO is quite a large language, and a full implementation (even a serial one) will take some time. We also intend to write much of the project software (including the compiler) in UFO. To ease this process, we should define a subset of the language (UFO0) for an initial bootstrap implementation. UFO0 needs to be suitable for writing a compiler (in a style compatible with the way the eventual UFO in UFO compiler will be written). Features likely to be omitted from UFO0 include multidimensional arrays, some other array/loop facilities, dynamic binding, much of the checking of constraints on stateful objects, and flags. As these facilities are independent, the progression to UFO1 (the language of this document) can be done in progressive stages. If the project is successful, there should eventually be a UFO2 which incorporates the experience gained with UFO1.

The current state of the project is that we have a parser, which accepts the syntax given in the appendix. It therefore includes most of the features described above, and most of the examples have been checked using it. A type checker will be added in the near future. Both the parser and the type checker will be for full UFO1. It is hoped to complete a serial UFO0 implementation, generating C, within the next few months. Implementation of UFO1, especially parallel implementations, will be much larger tasks.

Acknowledgements

Wim Böhm of Colorado State University was closely involved in the origins of UFO. Chris Kirkham has read many previous drafts of this document, and made numerous suggestions, as well as doing much of the work on the arrays section. Mario Wolczko provided invaluable input from the object-oriented perspective.

References

[1] S. Sakai, Y. Koadama, Y. Yamaguchi **Architectural Design of a Parallel Supercompuer EM5** JSPP 91.

[2] A.P.W. Böhm, D.C. Cann, D.H. Grit, J.T. Feo, R.R. Oldehoft, **SISAL Reference Manual** language version 2.0., Computer Science dept. Colorado State University, 1991.

[3] R.S. Nikhil, **ID Reference Manual**, CSG memo 284, MIT Laboratory for Compuer Science, 1988.

[4] B. Meyer, **Object-oriented Software Construction**, Prentice Hall, 1988

[5] M. Ben-Ari, **Principles of Concurrent Programming**, Prentice Hall 1982

[6] V. Sarkar, D.C. Cann, **POSC: a Partitioning and Optimising SISAL Compiler.**

[7] D. A. Turner **The semantic elegence of applicative languages** Proceedings of the 1981 ACM conference on Functional Programming Languages and Computer Architecture.

[8] A. Yonezawa (ed) **ABCL: an Object-Oriented Concurrent System** MIT Press computer systems series, 1990.

[9] P. America, **POOL-T: A Parallel Object-Oriented Language**, in A. Yonezawa & M. Tokoro (eds) Object-Oriented Concurrent Programming, MIT Press computer systems series, 1987

[10] G. Agha **Actors: A Model of Concurrent Computation in Distributed Systems** MIT Press series in artificial intelligence, 1986.

[11] B. Meyer **Eiffel the Language**, Prentice Hall 1992.

[12] J. Grosch & B. Vielsack **The Parser Generators Lalr and Ell**, Gesellschaft für Mathematik und Datenverarbeitung mbH, Forschungstelle an der Univerität Karlsruhe, 1991.

[13] M. Haines and A. P. W. Böhm **Towards a Distributed Memory Implementation of Sisal**, Scalable High Performance Computing Conference, April 1992.

A Further examples

A.1 Closure under laws

This example is part of the "paraffins program" taken from a classic D.A. Turner paper[7]. It is one of the nicest examples of functional programming I know.[25]

The example uses functional UFO only. Notice the use of function values, e.g. in the Set class to allow a particular Set to determine how equality is defined.[26] The main

[25] The whole of the paraffins problem has been written in UFO, and presents no particular problems. However, the full program text, plus the explanation of the problem, plus discussion of various aspects of the solution, makes it quite long. It is intended that a separate document will contain it, and other larger examples of UFO programs

[26] In the full paraffins program, the equivalence function defined below is itself used to generate sets of paraffins.

function, closure_under_laws calculates the transitive closure of a set or arbitrary values under a set of arbitrary laws, provided as a set of functions. The algorithm is potentially parallel if the generated set is reasonably large.

```
%  Assume a (functional) set class.
class Set[T]( from: Array[T] )
constructors
  % Construct a set given a user-specified equality function. The
  % implementation will squirrel this away and use it in member etc.
  % instead of the default.
  Set( from: Array[T]; equals: ( T, T ) -> boolean );
functions
  empty: boolean;
  add( newel:T ): Set[T];
  member( val: T ):boolean;
  intersection( s: Set[T] ): Set[T];
  union( s: Set[T] ): Set[T];
  difference( s: Set[T] ): Set[T];
  array: Array[T];
end

type Funset[Alpha] = Array[ Alpha -> Alpha ];

% a and b are equivalent if b is a member of the equivalence class of a.
function equiv( a, b: Alpha ): Bool is
member(equivclass(a,paralaws),b)

% The equivalence class is generated by transitive closure
function equivclass( a: Alpha; laws: Funset[Alpha] ): Set[Alpha] is
closure_under_laws( laws, Set( [a] ), Set( [a] )  )

% The idea is to apply each of the laws to each of the new elements
% from the previous application, until no new elements are generated.
function closure_under_laws( laws: Funset[Alpha];
                             accumset, newset: Set[Alpha] ): Set[Alpha] is
if newset.empty
then accumset
else
  { possibles =
       for f in laws join xs in array(newset) do
       ys = f(xs)
       return Set(ys)
       od;
     newones = difference( possibles, accumset )
  return closure_under_laws( laws, union( accumset, newones ), newones )
  }
```

```
fi

% e.g. if the following two laws on arrays are used...

% swap the first two elements
function swap( x: Array[T] ): Array[T] is
if size(x) < 2
then x
else [x[2], x[1]] ++ x[3..]
fi

% Rotate first and last elements
function rotate( x: Array[T] ): Array[T] is
if size(x) < 2
then x
else x[2..] ++ [x[1]]
fi

% ... the result is all permutations of the input array.
```

A.2 A nondeterministic parallel problem solver

This is a "concurrent OOP" example, although interestingly function values are again heavily used for generality. Three "standard components" are defined:

1. A nondeterminsitic Merger, useful for various producer-consumer type problems. This makes use of the Queue and Semaphore classes defined earlier[27]
2. An "active" Processor object which, when activated, repeatedly takes items from a source Merger, processes then according to some function (provided as a parameter, of course) and sends them to another, until it gets deactivated.
3. A general-purpose problem-solver function, which nondeterministically finds a solution from an array of candidates, if one exists. It sets up input and output Merger queues, and a number of Processors, and runs them until a solution is found or there are no more candidates. In the former case it stops the Processors to avoid unnecessary computation.

A plausible use for such a thing, finding a prime number from among a large number of big numbers which might be prime, is then a one-liner. Obviously, many more sophisticated merging and distribution primitives are possible, and a good UFO library should have a good range of such "standard parts".

[27]A true object-oriented programmer would probably inherit the Queue, and possibly the Semaphore too.

I would expect a good UFO implementation to distribute the Processor objects sensibly across the physical processors, without the need for explicit placement by the programmer.

```
% Nondeterministic merger, for producer-consumer type problems. Uses the
% queue, and a semaphore to block if the queue is empty. Could easily turn
% it into a classic buffer by blocking the other way if the queue gets too
% long.

stateful class Merger[T];
procs
  put( x:T );
  get: T;

imp
  s = Semaphore(0);
  q: Queue[T] = Queue;

proc put is { q!enqueue(x) fby s!sem_signal };

proc get is { s!sem_wait fby return q!dequeue };

end

type TtoU = T -> U;

% An ``active'' object which, when activated, repeatedly takes something out
% of a queue, processes it, and adds it to another queue. It can also be
% deactivated, although since there is no preemption, this won't happen
% until the current message has been processed. Notice the generality
% obtained by providing the processing function as a parameter.

stateful class Processor( source: Merger[T]; dest: Merger[U]; f: TtoU );
procs
  activate;
  deactivate;

imp
  store source, dest, f;
state
  go := FALSE;

proc activate is
{ go := TRUE fby self!continue }

proc continue is
```

```
if go then
  {
  x = source !get;
  y = f(x) ;
  dest!put(y) fby
  self!continue % Queue behind any waiting deactivate.
  }
fi

proc deactivate is { go := FALSE };
end % class Processor

% Returns the first solution it finds from the array of candidates provided,
% or a null result if none found. It does this by creating ‘numprocs’
% Processors, each of which tries to find solutions until one is found, in
% which case it kills them off.
typevar Candidate, Solution;

function findasolution( candidates: Array[Candidate]; numprocs: Int;
         null_solution: Solution; try: Candidate -> Solution): Solution
is
{
  % Create input and output Mergers.
  input_queue : Merger[Candidate] = Merger;
  output_queue : Merger[Solution] = Merger;
  % Create the processors. Assume function try which takes a candidate
  %  and returns a solution if it can and the null solution otherwise.
  procobjs =
    for i in [1 to numprocs] do
    p = Processor( input_queue, output_queue, try );
    p ! activate;
    return p
    od;
  % Feed in the Candidates one end...
  for cand in candidates do input_queue!put( cand ) od;
  % .. and see what comes out at the other.
  result =
  initially
    found = FALSE;
    numresults = 0;
    res = null_solution;
  while numresults < size(candidates) and res == null_solution do
    new res = output_queue!get;
    new numresults = numresults + 1;
  return last(res)
  od;
```

```
  % If a Solution was found, the processors may still be going and we
  % need to tidy up...
  if result != null_solution
    then
    for i in [1 to numprocs] do procobjs[i]!deactivate od
    fi
return result
} % end of findasolution

% For instance, to find a prime from an array of possible primes:
function findaprime is
  findasolution( possprimes, n, Bignum(0), checkprime )
% where checkprime is a function which takes a Bignum and returns the
% number of it's prime, and Bignum(0) otherwise.
```

B Syntax

The syntax is given in the extended BNF used by the Karlsruhe University parser generator lalr [12]. The conventions are as follows:

```
(* ... *) Comment
'token'   Literal token
[ X ]     Optional X
X || Y    Sequence of Xs separated by Ys
X+        One or more Xs
X*        Zero or more Xs
X Y       X then Y
X | Y     Either X or Y
```

The following words are reserved:

```
all and Array Bits Bool Char class closed constructor constructors
cross div do dot Double else elseif end error deferred fby fi for
function functions if imp in inherits initially Int is join mod
Mytype new not null nullcase od operator operators or otherwise
proc procs Real return self state stateful step store then to type
typevar while
```

The syntax is as follows (currently this does not include the full array/loop syntax, or the clusters proposal).

```
(* This gives the precedence and associativity of the operators.
  The list is in order of increasing precedence *)

OPER
```

```
  RIGHT '->'
  LEFT '++' '::' 'and' 'or'
  NONE '==' '!=' '<' '>' '<=' '>='
  LEFT '+' '-'
  LEFT '*' '/' 'div' 'mod'
  NONE 'not'
  NONE '['
  NONE ':'
  NONE '.' '!'

RULE

(* A program is a collection of class definitions and implementations, and
``free functions'' which do not belong to a class. There may also be
type and type variable definitions. *)

  program: topleveldef +  .
  topleveldef : classdef | classimp | functiondef | typedef | typevardef
              | procdef .
  classdef : [ 'closed' | 'deferred' ] [ 'stateful' ] 'class' id
             [generics] [formals] ['inherits' id] [';']
             methoddecs [ 'imp' implementation ] 'end' [';']  .
  classimp : 'imp' id [generics] [formals] [';'] implementation 'end' [';'] .
  generics : '[' ids ']' .
  methoddecs : [ 'constructors' methoddec+ ] [ 'functions' methoddec+ ]
               [ 'procs' methoddec+ ] [ 'operators' opdec+ ] .
  methoddec : id header [';'] .
  opdec : op header [';'] .
  header : [formals] [ ':' typespec ] .
  formals : '(' ( typedids | 'self' ) || ';' ')' .

(* Some generally useful bits and pieces: comma-separated identifiers,
  typed comma-separated identifiers, semicolon-separated statements *)
  ids : id [ ',' ids ] .
  typedids : ids ':' typespec .
  statements : statement ( separator statement )* [separator] .
  separator : ';' | 'fby' .
  expressions : expression ( ',' expression )* .

(* An implementation consists of optional type and type variable definitions
(which will be local to the implementation), followed by value definitions
for the instance variables, possibly initialisation of instance variables,
and then the methods. *)

  implementation : typevardef* typedef* [storedparams] [valdefs ]
                   ['state' [storedparams] vardefs] methoddef* .
```

```
storedparams : 'store' ids ';' .
methoddef : (functiondef | procdef | condef|opdef) [';'] .
functiondef : 'function' id [formals] [':' typespec] 'is' body  .
procdef : 'proc' id [formals] [':' typespec] 'is' body .
opdef : 'operator' op [formals ':' typespec] 'is' body .
condef : 'constructor' id [formals] 'is' [storedparams] [valdefs]
           ['state' [storedparams] vardefs] 'endcon' .
body : expression
      | 'nullcase' ':' expression [';'] 'otherwise' ':' expression .
valdefs : valdef ( ';' valdef )* [';'] .
valdef :  lvalue  ( ',' lvalue )*  '=' expression .
lvalue : 'new' id [':' typespec]
        | id [':' typespec] .
vardefs : assignment ( ';' assignment )* [';'] .
assignment : id [ ':' typespec ] ':=' expression .


typevardef : 'typevar' ids ';' .
typedef : 'type' id [ '[' ids ']' ] '=' typespec ';' .
typespecs : typespec || ',' .
typespec : basictype
           | 'Array' '[' typespec ']'
           | id [ '[' typespecs ']' ]
           | typespec '->' typespec
           | '(' typespecs ')'  .
 basictype :  'Int' | 'Real' | 'Double' | 'Char' | 'Bool' | 'Bits' .

expression : lvalue
           | expression '+' expression
           | expression '-' expression
           | expression '*' expression
           | expression '/' expression
           | expression 'div' expression
           | expression 'mod' expression
           | expression '++' expression
           | expression '::' expression
           | expression 'and' expression
           | expression 'or' expression
           | expression '==' expression
           | expression '!=' expression
           | expression '<' expression
           | expression '>' expression
           | expression '<=' expression
           | expression '>=' expression
           | constant
           | block
```

```
           | loopexp
           | ifexp
           | arrayexp [':' typespec ]
           | '(' expressions ')' [':' typespec ]
           | funcall
           | proccall
           | 'not' expression
           | expression '[' expressions ']'
           | basictype '(' expression ')'
           | opfun [':' typespec ]
           | 'self'
           .
expression : '-' expression  PREC '*' .

constant :  intconstant | realconstant | charconstant
            | stringconstant | boolconstant .

opfun : '(' op ')' .

op : '+' | '-' | '*' | '/' | 'div' | 'mod' | '++' | '::' .

loopexp : [loopcontrol] 'do' statements [ 'return' expression ] 'od' .
ifexp : 'if' expression 'then' expression
         ('elseif' expression 'then' expression)*
         [ 'else' expression ] 'fi' .

arrayexp : '[' [expressions] ']'
         | '[' expression 'to' expression ['step' expression ] ']' .

statement : valdef | assignment | loopexp | ifexp | proccall | block .

loopcontrol : ['initially' valdefs]
              ( 'while' expression | 'for' generators ) .
generators : generator | generator 'dot' generators
            | generator 'join' generators
            | generator 'cross' generators .
generator : id 'in' expression.

block : '{' statements ['return' expression] '}' .

funcall : expression '.' id [ '(' expressions ')' ]
          | id '(' expressions ')' .

proccall : expression '!' id [ '(' expressions ')' ] .
```