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

Technical Report Series
UMCS–93–1–4

John Sargeant

United Functions and Objects: an Overview

United Functions and Objects: an Overview*

John Sargeant
Department of Computer Science
University of Manchester
Manchester U.K.
Tel. +44 61 275 6292
Fax +44 61 275 6236
js@cs.man.ac.uk

Abstract

United Functions and Objects (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.

This report concentrates on examples which illustrate various aspects of UFO, including the various encapsulation and abstraction mechanisms it provides, and the various forms of parallelism which can be exploited.

Technical Reports issued by the Department of Computer Science, Manchester University, are available by anonymous ftp from `m1.cs.man.ac.uk` (130.88.13.4) in the directory `/pub/TR`. The files are stored as PostScript, in compressed form, with the report number as filename. Alternatively, reports are available by post from The Computer Library, Department of Computer Science, The University, Oxford Road, Manchester M13 9PL, U.K.

1 Introduction

Modern computers are parallel, and the amount of parallelism is steadily increasing. Most modern programming languages still assume they are serial. This mismatch has unfortunate consequences. On a small scale, compilers must extract increasing amounts of parallelism from sequential code in order to use pipelined, superscalar processors effectively. On a larger scale, programming for multiprocessors usually requires a significantly higher degree of skill and specialisation than traditional sequential programming. This latter difficulty stems from the explicit nature of most parallel programming; the application programmer has to map a particular problem onto a particular machine.

It is obviously desirable to have an implicitly parallel programming language, in which this mapping is done by the compiler and runtime system, and is not the responsibility of the application programmer. However, the various forms of implicit parallelism explored during the 80s (functional, And/Or parallel logic, concurrent object-oriented etc.[1, 2, 3, 4, 5, 6]) have, by and large, made little impact on real use of parallel machines. Fortran is enjoying a resurgence.

One problem has been lack of convincing demonstrations of performance. Efficient large-scale implicit parallelism is hard to achieve, but the UFO project will be able to build on a considerable body of research in this area, and on some promising results, such as those obtained for SISAL[7].

A more fundamental reason, in the author's view, is that many such languages have been too narrowly focussed, and have not incorporated the best of modern programming language technology. Functional languages, for instance, form a promising starting point, since they allow parallelism naturally, guarantee determinism, and are easy to reason about. However, pure functional languages seem inappropriate for some problems, which are more naturally expressed in terms of communicating objects, each of which manages its own local state. It *is* possible to have the benefits of both styles in one language, provided that careful attention is paid to the mechanisms for encapsulating state, and that certain safeguards are provided by a static type system.

The following are important characteristics of a realistic implicitly parallel language:

Parallelism by default. It seems inappropriate to merely enhance a sequential model of computation with a few parallel constructs in a world where computers are parallel. The design of UFO reduces sequencing to a minimum: data dependence and the queueing required to guarantee coherent access to stateful objects. The underlying computational model is (extended) dataflow, rather than communicating sequential threads, although an implementation may well use such threads.

Good abstraction and encapsulation facilities. The ability to write generic, reusable code is even more important than in a sequential environment, as some parallel algorithms are inherently more complex, and easier to get wrong, than sequential ones. UFO combines abstraction mechanisms normally found in functional languages, with others from object-oriented languages.

Amenability to reasoning. It should be relatively easy to reason about programs, both formally and informally, both by the programmer and the implementation. This is aided in UFO by the presence of a useful, statically determined, pure functional subset, and a static type system which ensures that programs are type-safe, and prevents various abuses of stateful objects.

No gratuitous inefficiency. Although some small constant factor in execution speed is a reasonable trade for expressiveness, serious inherent overheads will make the language unacceptable. UFO therefore avoids, for instance, lazy evaluation, and provides a mechanism to "switch off" dynamic binding when efficiency is critical.

UFO is not a narrow "single paradigm" language. It has been influenced by a number of disparate language styles, leading to an interesting, and hopefully very useful, synthesis. The main influences are as follows:

1.1 Dataflow languages, especially SISAL

SISAL [1] is a pure functional language with strict semantics, primarily geared towards numerical computation. It can be classed as a dataflow language, in that the underlying computational model is a parallel dataflow one. Sequencing is by data dependence only; parallelism is the default.

A great deal of work has been done on optimising SISAL for conventional supercomputers, and recently substantial numeric SISAL programs have been shown to run faster on multiprocessor Crays than Fortran versions [7]. Functional languages need not be inefficient, at least for such applications. However, SISAL is quite a limited language (e.g. it has no polymorphism, data abstraction, or higher-order features; multi-dimensional arrays have to be represented as arrays of arrays). An update, SISAL2, has been defined [9] which addresses some, but by no means all, of these limitations.

The original idea behind UFO was to create a language based on SISAL which could be used for a wider range of parallel applications, by adding objects to encapsulate updatable state. In fact, UFO has gone well beyond this original idea, but still has a subset which (apart from syntactic differences) is very similar to SISAL.

1.2 Object-oriented languages, especially Eiffel

Issues of software reliability, reuse, etc. are even more critical in parallel programming than in the old sequential world. It rapidly became clear that UFO must have good encapsulation and abstraction mechanisms, and a flexible static type system. A survey of object-oriented languages rapidly showed that Eiffel [10, 11] was closest to what was required. In particular, the Eiffel type system, with its elegant combination of genericity and inheritance, instantly looked "right".[1]

[1]The author is firmly of the opinion that static typing is a Good Thing, except for a few specialised applications. By definition, the end user never sees static type errors, but may well see runtime ones!

One distinctive feature of the Eiffel type system is the use of constrained generic types. A type variable may be constrained so that it may only be bound to types which conform to the constraining type. For instance, if `Ord` is a class which defines ordering operators, then a type variable may be declared[2]

```
T <- Ord
```

and any actual type provided for `T` must conform to `Ord`. The advantage of this is that generic types can be used in much more concrete ways; any methods defined on the constraining type, such as $>$, may safely be used on values of such types.

The UFO type system is therefore heavily influenced by Eiffel, although currently the rules for redefinition on inheritance are more restrictive, in order to avoid complex global validity checking. Unfortunately, to someone brought up in the functional/dataflow world, Eiffel looks extremely imperative and serial, and so the runtime semantics of UFO are very different.

1.3 Pure lazy functional languages, notably Haskell

Modern pure functional languages, for which Haskell [8] is now the standard, are characterised by higher-order functions, lazy evaluation, and static type systems with strong similarities to that of Eiffel.

Initially, UFO allowed constant function values, but not full higher-order functions, as it was feared that the latter would over-complicate the type system. Early experience showed that a partial parameterisation mechanism was useful, and gave no particular problems. Examples appear below.

Lazy evaluation, however, was never an option. Although it improves the expressiveness of a pure functional language, lazy evaluation is incompatible with the presence of updatable variables, as the execution order is almost impossible to visualise. Furthermore, the semantics require normal order evaluation, which is sequential. To exploit parallelism in a lazy language, it is necessary to do strictness analysis, in much the same way as it is necessary to do dataflow analysis to extract parallelism from an imperative language. It is still unclear how successfully this can be done. Also, programming techniques for handling I/O and other "imperative" features in lazy functional languages involve deliberate serialization[19].

It is also worth noting that laziness conflicts with dynamic binding, since in order to select a method on the basis of the class of an object, it is necessary to evaluate the object before calling the method!

The effect of lazy evaluation can be simulated in UFO for those applications which really benefit from it. For instance, the `Future` class in section 6.7 can be used to build lazy data structures. The wavefront example in section 6.6 is one sort of example sometimes used to justify laziness, although in that case only non-strict (as opposed to actually lazy) evaluation is necessary.

[2] This is UFO syntax - in Eiffel the arrow is the other way round!

Haskell provides type classes, which are a systematic way of dealing with overloading, and a first step towards "proper" classes and inheritance. For instance, there is a type class `Eq` which includes all types with equality defined, and it has a subclass `Ord` of types which also define ordering. Type classes can be used to impose constraints on generic types. For instance, the type of a sorting function is:

```
sort :: Ord a => [a] -> [a]
```

"sort is of type list of `a` to list of `a` provided `a` is an instance of `Ord`". The ordering operators (< etc.) can then be used within `sort` in the knowledge that any actual type provided will have implementations of them. Notice that this is constrained genericity, much as in Eiffel.

However, the actual implementation of the operations is concentrated in the instances (actual types) at the leaves of the class hierarchy. It is not possible to inherit from an instance, and so the normal OO practice of incrementally adding implementation down the hierarchy is not possible. This is very restrictive and UFO has a more conventional inheritance mechanism.

1.4 Concurrent object-based languages, particularly ABCL

In concurrent object-based[3] languages, such as actor languages [3], POOL [5] or ABCL [4], a computation is expressed as a network of communicating objects, each of which manages its own local state. ABCL seemed the best developed of these.

There are considerable similarities between UFO (stateful) objects and ABCL objects; they provide mutual exclusion on method accesses, so ensuring coherent updating of the instance variables. Incoming messages/method calls are queued if necessary. An object may continue to execute a method after it has returned a result, and may in some circumstances accept another message before it has returned a result. An object may selectively accept some messages and not others.[4]

However, the differences are substantial; the model underlying ABCL is one of communicating sequential threads. As a result there is a distinction between different sorts of message passing ("past", "present" and "future") which is unnecessary in UFO. ABCL also has a notion of pre-emption ("express messages") which is inappropriate given the semantics of UFO stateful objects, as described below.

2 Functional UFO

This section describes a purely functional subset of UFO, showing how classes and inheritance are useful things to have, even in a pure functional language, and how these object-oriented notions can peacefully exist alongside higher-order functions. The description is informal, via a series of small examples.

[3]Following the convention that a language with objects but not inheritance is described as object-based, rather than object-oriented

[4]This last feature was directly inspired by conversations with the designers of ABCL.

2.1 Why functional?

The recommended style for writing UFO programs is to remain within the pure functional subset unless there is a clear reason to introduce stateful objects. This is in sharp contrast to the very imperative style usually adopted in OOP. The reasons for this are as follows:

- Functional programming is very simple; there is no need to distinguish between a copy of an object and a reference to it, for instance. Solution composition[5] is simply expressed as function composition.

- Parallelism is implicit, and relatively easy to exploit, as parallel calls cannot interfere with each other.

- Parallel functional programs are guaranteed to be deterministic; the result cannot depend on evaluation order.

- Functional programs are relatively easy to reason about formally. This allows simple program transformations (between serial and parallel algorithms, for instance), and allows a class of compile-time optimisations (e.g. common subexpression elimination and dataflow loop "vectorisation" [14]) to be done more easily for general cases.

- Functional programming has some neat abstraction and encapsulation mechanisms, particularly higher-order functions to capture common patterns of recursion.[6]

The efficiency argument against (strict) functional languages is no longer valid for most applications, as most of the sources of inefficiency can be removed. Sharing analysis can drastically reduce the number of copies made of data structures, as the SISAL results show. Higher-order functions can be transformed out. Garbage collection is present in most object-oriented languages anyway. The remaining inefficiencies are almost certainly balanced by the ability to exploit a small amount of parallelism in almost any application, and are overwhelmed for highly parallel applications.

Given all this, it is perfectly reasonable to write substantial programs (compilers, for instance) entirely in the pure functional style. However, the author is not a True Believer in pure functional languages: solutions to some problems are simply best expressed using stateful objects. Some parallel algorithms (for instance bounded search algorithms) may be much more efficient if stateful objects are used.

2.2 A divide-and-conquer function

One simple form of parallelism is found in divide-and-conquer algorithms which recursively split a problem into subproblems until the subproblems are small enough to be solved directly.

[5] Not problem decomposition, if we believe in object-oriented design!

[6] In fact, some object-oriented languages have comparable facilities, blocks in Smalltalk, for instance.

The following general framework for divide-and-conquer programs[7] illustrates a number of the features of UFO.

```
typevar Problem, Solution

dc( problem: Problem; primitive: Problem -> Bool;
    divide: Problem -> Array[Problem]; combine: Array[Solution] -> Solution;
    solve : Problem -> Solution ): Solution is
if problem.primitive
then problem.solve
else combine( map(problem.divide, dc(_, primitive, divide, combine, solve )))
fi
```

UFO has strong static generic typing, with mandatory type declarations for function parameters. Type inference, as used in most modern functional languages, (e.g. Haskell[8]) would be very complex in the presence of inheritance, and its benefits to the programmer would be outweighed (as it is in Haskell) by the unhelpful type error messages which would result.

The divide-and-conquer function takes as arguments an arbitrary Problem, a predicate to test whether a Problem is small enough to solve directly, a function which divides a Problem into an array of smaller Problems, another which combines an array of Solutions into a single Solution, and finally the function which turns a small Problem into a Solution. A particular divide-and-conquer problem is expressed by providing `primitive`, `divide`, `combine`, and `solve` functions of the appropriate types.

The body of a function is an expression, in this case a conditional expression. If the problem is primitive, it is solved directly. Otherwise, it is split into an array of subproblems by `divide`. `dc`, with the four functions wired-in, is mapped across the array of subproblems (`map` simply applies a function to each element of an array). The resulting array of solutions is combined to give the answer.

The reader may have noticed two different function call syntaxes: an OO-style "message passing" syntax:

```
problem.primitive
```

and a conventional one:

```
combine( ... )
```

These are interchangeable via a trivial syntactic transformation; the programmer is free to use whichever seems more natural for a particular call.

Another interesting aspect is the use of partial parameterisation in

```
dc(_, primitive, divide, combine, solve )
```

[7] This formulation of divide-and-conquer is believed to be due to Ronan Sleep[17]

this gives a function of type Problem $\rightarrow$ Solution. Originally, UFO did not have this, but experience in building the first UFO-in-UFO subset compiler showed that it was often useful. Without it, for instance, different versions of the map function are required if the function being applied has additional arguments.

In a lazy functional language, extracting parallelism from `dc` requires strictness analysis to determine that `combine` must evaluate the whole of its input list. Failing this, serial, normal order evaluation will occur. For this reason (amongst others) UFO has strict evaluation. Lazy evaluation can be explicitly programmed, for those applications for which it is really useful.

2.3 A simple list class

Many of the features of functional UFO can be illustrated via a simple list class, whose definition begins as follows:

```
** Functional list class
class List[Alpha]( private h: Alpha; private t: List[Alpha] )
  head : Alpha is
    nullcase : error( "Head of null list!\n" )
    otherwise : h;

  tail : List[Alpha] is
    nullcase : error( "Tail of null list!\n" )
    otherwise : t;
```

The first line above declares both the `List` class, and a constructor function `List( h, t )` which can be used to build elements of it. Alpha is a type variable: as in Eiffel (and Haskell) the component type of a list is arbitrary, but it must be consistent within a particular list. `h` and `t` are stored as instance values (not instance *variables*: their values cannot be updated), but since they are declared private, a client of the class cannot access them directly. This implements data abstraction, here used only to provide "protected" `head` and `tail` functions which attempt to fail gracefully if the list is empty.

In contrast to most languages, `null` is not a special value with its own type, or with no type at all; each (non-scalar) type has its own distinct null value.[8] This allows methods to be called on null objects; null is detected by the `nullcase` clause, if present, otherwise a default `nullcase` signals an error. This not only allows more graceful failure in cases such as the above, but also frequently allows recursive algorithms to be written more neatly, as the next few examples will show. Every type also has an associated error value, generated by the `error` function.

The rest of the list class is as follows:

```
  map ( f: Alpha -> Beta ) : List[Beta] is
```

[8]This can be implemented as a pointer to a "null implementation" of the class

```
    nullcase : null
    otherwise : f(h) ::t.map( f );

  foldr ( op: (Alpha, Beta) -> Beta; id: Beta ):Beta is
    nullcase : id
    otherwise : op( h, t.foldr( op, id ));

  filter( p: Alpha -> Bool ):List[Alpha] is
    nullcase : null
    otherwise : if p(h) then h :: t.filter( p ) else t.filter( p ) fi;

  ** Append
  operator ++ ( other: List[Alpha] ): List[Alpha] is
    nullcase  : other
    otherwise : h :: (t ++ other)
end ** class List

** Cons
operator :: ( h: Alpha; t: List[Alpha] ): List[Alpha] is List( h, t );
```

`map`, `foldr` and `filter` are standard higher order functions. `map` applies function f to every element of the list, producing a new list. `foldr` "folds" up the list (from the right, or far, end) by combining successive elements using `op`, to produce a single result element. `filter` returns all the elements of the list which satisfy some predicate. Such functions encapsulate common patterns of recursion, and are heavily used by functional programmers to avoid messy explicit recursions.

Infix operators, here cons (`::`) and append (`++`), may also be defined. Cons is defined outside the class itself because the class is implicitly the first argument to every method defined within it, and the list is the second argument to cons.

These functions (apart from `::`) show how the base case of a recursion is typically to a null value, and this is caught by `nullcase`. Without this, every user of `map` would need to remember to test for `null` before calling it. The definition of `map` itself would have to be:

```
map ( f: Alpha -> Beta ) : List[Beta] is
  f(h) :: if t.is_null then null else t.map( f ) fi
```

In most FLs, `map`, `foldr` and `filter` are defined only on lists. However, the concepts of mapping, folding, and filtering apply to arbitrary collections. UFO allows function names to be overloaded on the first argument type (i.e. the class type when the function is defined within a class).

There is no separate interface declaration for a class, since a simple tool can be used to extract the interface information:[9]

[9]This idea is directly stolen from Eiffel.

```
** Functional list class
class List[Alpha]( h: Alpha; t: List[Alpha] )
  head : Alpha
  tail : List[Alpha]
  foldr ( op: (Alpha, Beta) -> Beta; id: Beta ): Beta
  filter( p: Alpha -> Bool ):List[Alpha]
  ** Append
  operator ++ ( other: List[Alpha] ): List[Alpha] is
end ** class List

** Cons
operator :: ( h: Alpha; t: List[Alpha] ): List[Alpha]
```

`map` provides an opportunity for parallelism, as all the applications of `f` can be made in parallel and are guaranteed not to interfere with each other. It will be shown later that this guarantee is maintained even if the elements of the list are stateful. An example of the use of `map` in the compiler is to apply a typechecking function to a list of definitions:

```
class Topdef ** Top-level definition class
  typecheck : (Topdef, List[Error]) is ...
end

** To typecheck a list of Topdefs...
check_all( defs : List[Topdef] ): List[(Topdef, List[Error])] is
  defs.map( Topdef| typecheck )
```

Since `typecheck` may be overloaded, it is necessary to disambiguate it. On a normal call, this is done according to the type of the first argument. When used as a function value, this is not possible. It would be possible to infer the type (easily in this case, since the type of `defs` gives it away) but in general this inference is very complex in the presence of inheritance. It was decided to require all global function values to be explicitly disambiguated, using the *type name* `Topdef|`.

In fact, the typechecking function has a second parameter which is the global namelist. It would be possible to have a separate mapping function for this case, but this would rapidly get messy. A more elegant solution can be obtained by use of partial application:

```
defs.map( Topdef| typecheck( _, nlist ) )
```

where since `typecheck` has type

```
( Topdef, Namelist ) -> (Topdef, List[Error])
```

the result of the partial application is a function of type:

```
Topdef -> (Topdef, List[Error])
```

as required for the `map`. Note that, since the first parameter is omitted in the partial application, explicit disambiguation is still required.

2.4 Inheritance

In UFO, classes with inheritance can be used even in pure functional programs. The prototype compiler is such a program and uses classes and dynamic binding extensively. Using static *typing* along with dynamic *binding* ensures that there is always an appropriate method to call at runtime. The argument for this is eloquently argued by Meyer in [10].

A simplified fragment of the compiler will illustrate class definitions and inheritance:

```
deferred class Expression( spec: TypeSpec) inherits Treenode
print: String is deferred
typecheck( cxt: Context ): (Expression, List[Error]) is (self, null)
end

class IfExp( ... condexp, thenexp, elseexp : Expression ) inherits Expression
  print is
     "if "   ++ condexp.print ++ "\n"
  ++ "then " ++ thenexp.print ++ "\n"
  ++ "else " ++ elseexp.print ++ "\nfi\n"

  typecheck is
    { condexp_out, cond_errs = condexp.typecheck( cxt );
      ** ... and so on
      ** Combine the errors from the various checks
      combined_errs = cond_errs ++  ...
      return (IfExp( result_type, condexp_out, thenexp_out, elseexp_out ),
              combined_errs )
    }
end ** Class IfExp
```

A deferred class[10], such as `Expression`, is one which serves as a superclass for other, concrete, classes but can't itself have instances. Individual methods of a deferred class may be deferred, like `print`, in which case the implementation is left entirely to subclasses, or a default implementation may be provided, as in the case of `typecheck`. The idea of `typecheck` is that it takes an expression, and a context (e.g. the namelist structure defining names in scope) and returns a new expression decorated with types, and a list of any errors found.

The parameters to a class (e.g. `spec` for `Expression`) are stored with the class as instance values. Unless they are declared `private`, these instance values can be accessed by code outside the class, in the same way as parameterless functions e.g.

```
t = some_exp.spec
```

[10]Also known as a virtual class, or abstract class in other object-oriented languages

IfExp is a concrete subclass of Expression. Typically, the class parameters for a subclass are an extension of those for its parent. This is indicated by the ... notation in the header.

`print` simply constructs a string representing the expression, in UFO syntax. This recursively calls other `print` functions on the component expressions. The appropriate `print` function is called at runtime by dynamic binding. The fact that there is an appropriate `print` function is guaranteed at compile time by static typing.

The body of `typecheck` is a *block*, which provides an inner scope, and contains a series of value definitions for local names such as `condexp_out`. These are *not* assignments; each name is given a value exactly once. The result of a block is the expression following `return`. The first part of the result of `typecheck` is an `IfExp`, built by calling a *constructor* giving actual values for the class parameters. Since the result type of `typecheck` is the tuple type `(Expression, List[Error])`, the result of type `(IfExp, List[Error])` is valid.

2.5 Dynamic binding vs. pattern matching

The only real conflict between functional and OO styles concerns the way in which selection is done. Modern functional languages rely heavily on pattern matching. It is intuitively appealing, and allows simple, concise definitions, and reasoning by case analysis. It is traditionally regarded as a Good Thing by functional programmers: "Pattern matching is one of the cornerstones of an equational style of definition; more often than not it leads to a cleaner and more readily understandable definition than a style based on conditional equations...it also simplifies the process of reasoning formally about functions" [15](p. 37).

For instance, `map` in Haskell is written:

```
map [] f = []
map (h:t) f = f h : map t f
```

and the expression example might be written:

```
type Expression = (Exp, TypeSpec)

data Exp =  ConstExp String
          | IfExp Expression Expression Expression

...

print (exp, spec) =
 case exp of
  Constexp str -> str
  IfExp condexp thenexp elseexp -> .... etc
```

Data types are defined by a combination of type synonyms (often for tuple types like `Expression` above) and algebraic data types. Operations on them are defined by pattern matching and explicit selection using case expressions. As explained above, the same effect is achieved in UFO by using named selectors (i.e. public instance values) for components of structures, and selection between alternatives is done by dynamic binding.

Pattern matching conflicts with data abstraction because, in order to pattern match on a data structure, its concrete representation must be known. Programs which use pattern matching heavily are painful to change, since changing a data structure causes missing cases and broken patterns to be spread across the program. Many, but not all, of these are detected statically. The trend among functional programmers seems to be to rely less on pattern matching and to program named selectors.

Dynamic binding gives better adaptation to change, since when a new subclass is added the additional methods required are localised within that subclass. Existing selectors are not affected, and missing cases can be detected statically as deferred methods not implemented in all subclasses.

In UFO, the special case of matching an empty list, or other null object, is handled by the `nullcase` construct or by explicitly testing with `is_null`. Other cases are handled by dynamic binding.

The above argument is an extension of that presented by Meyer against case statements in [10]. It suggests that achieving a UFO-style synthesis of Logic Programming with OOP would be much more difficult, due to the total reliance of LP on the pattern matching of unification.

2.6 Multiple inheritance

Multiple inheritance (MI) provides extra functionality at the cost of extra complexity in the language and its implementation. The initial intention was to stick with single inheritance in the first version of UFO (UFO1), perhaps introducing MI later. However, it now looks likely that a limited form of MI will appear in UFO1.[11]

Consider the following functions on lists:

```
** Compare two lists for equality
eqlist( l1, l2 : List[T] ): Bool is
  if l1.is_null and l2.is_null then True
  elseif l1.is_null or l2.is_null then False
  elseif l1.head != l2.head then False
  else eqlist( l1.tail, l2.tail )
  fi

** Convert a list to a string, separating elements by 'sep'.
printlist( l : List[T]; sep : String ): String is
```

[11] The content of this section is much more tentative than that of most of this report.

```
if l.is_null then ""
else l.head.print ++
  if l.tail.is_null then "" else sep ++ l.tail.printlist( sep ) fi
fi
```

To ensure type safety, the type variable `T` in the first function must be constrained to ensure that equality is defined, and in the second case so that `print` is defined, i.e.

```
typevar T <- Eq
typevar T <- Printable
```

Since many types will be both comparable for equality and printable, with single inheritance, `Eq` and `Printable` would have to be the same class, or one a descendent of the other. This is clearly unsatisfactory.

A possible solution is to allow a restricted form of MI, where all parents except one of a class must be *shadow classes*. A shadow class is one which contains no implementation: no instance values or variables, and all its methods are deferred. If a concrete class inherits (say) `Eq`, then it (or a concrete ancestor) must define equality.

A less restrictive version of MI would be to allow abstract classes with methods but no instance values or variables. At the time of writing this is still being considered.

3 Arrays and loops

Another form of parallelism is that generated from arrays and loops, often known as data parallelism, although that term usually means a rather more restrictive form than that which UFO provides.

The array and loop constructs of UFO are largely inspired by SISAL[1]. The SISAL project has shown that functional arrays can be efficient, provided that suitable optimisations, particularly sharing analysis to avoid unnecessary copying, are performed by the compiler. Substantial numeric programs written in SISAL have been shown to run faster on multiprocessor Crays than Fortran versions[7]. The key to success is to avoid low-level programming in terms of individual updates, and instead to write loops which generate whole arrays at once.

Arbitrary SISAL1 programs can easily be translated into UFO. Most such programs are numeric, although some of the examples chosen here are not. SISAL2.0 [9] has very sophisticated array handling facilities, although they are possibly rather too complex, and UFO provides a smaller number of general-purpose features, at the cost (possibly) of more optimisation work for the implementation.

The `map` function on arrays is written:

```
map( a: Array[T]; f : T-> U ): Array[U] is
   for x in a do y = f(x) return y od
```

`for` loops are generated by arrays and create arrays.[12] The effect of a loop ranging over integer indices, as in

```
for i in [1 to n]
```

is achieved by having syntax for arrays containing arithmetic progressions. In practice, of course, such an array will not be stored by a good implementation. A name defined in the body of the loop, such as `y`, corresponds to the name of an array in the `returns` part, with one element for each instantiation of the loop. The loop cycles are independent, so all the calls to f may be evaluated in parallel.

The following program , adapted from an example in the SISAL2 reference manual[9] "plays" Conway's "game of life". A colony of cells live on a grid. If a cell has more than 3 neighbours it "dies of overcrowding". If it has less than 2, it "dies of loneliness". If an empty square has exactly 3 neighbours, a cell is born there.

```
type Cell = Int
type Grid = Array[Cell]

dead is 0
alive is 1

newcell( g:Grid; i,j: Int ): Cell is
{
neighbours = G[i+1, j-1] + G[i+1, j] + G[i+1, j+1]
           + G[i,   j-1]             + G[i,   j+1]
           + G[i-1, j-1] + G[i-1, j] + G[i-1, j+1]
return
  if neighbours > 3 or neighbours < 2 then dead
  elseif neighbours == 3 then alive
  else G[i,j]
  fi
}

nextgrid( g: Grid ):Grid is
{
x = dimsize(g, 1);
y = dimsize(g, 2)
return
  for i in [2 to x-1] cross j in [2 to y-1] do
  newg = newcell( g, i, j)
  return  newg
  od
}
```

[12] We are currently considering using the loop syntax to express iteration over user-defined sequenceable types, and possibly providing an "array comprehension" syntax, as a shorthand for simple loops.

Although UFO has true multi-dimensional arrays, there is only one `Array` type. The shape of an array is, like its bounds, a property of a particular array value, and in general is determined dynamically. This simplifies handling of arrays in that functions (such as `map`) can be written which iterate over an array irrespective of its shape. It does, however, mean that indexing an array with the wrong number of indices may not be caught until runtime! In practice, it is expected that is will be possible to determine the shape of arrays statically most of the time, certainly for typical numeric applications.

The `newcell` function simply calculates whether a cell will be dead or alive on the next iteration according to the rules. `nextgrid` calculates the next version of the whole grid. The generators determine the shape of the result array. Combining two arrays in a `cross` product generator, as in this case, creates a result array whose dimensionality is the sum of the dimensionalities of the generating arrays, in this case 2D. A name defined in a generator, or in the loop body, such as `newg`, becomes the name of an array with one element for each instantiation of the loop when referred to in the `return` part of the loop. So

```
return newg
```

returns a 2D array of cells. Of course all the loop bodies may be evaluated in parallel.

The example is incomplete in two respects. Firstly, `nextgrid` returns a smaller grid than the one it's given; we need to put the boundary back on. Secondly, we need to set up the grid in the first place, preferably without having to "draw" the whole of it, or to do messy individual updates. This can be done as follows:

```
glider : Grid is
{ o = dead;
  x = alive
  return [2..4, 2..4 | o, o, x,
                       x, o, x,
                       o, x, x ]
}

iterate( n: Int) : Grid is
initially
  emptygrid = [1..100, 1..100 | all dead];
  i = 0;
  grid = emptygrid # glider
while i < n do
  new grid = emptygrid # nextgrid(grid);
  new i = i + 1
return last(grid)
od
```

The glider function sets up the standard "glider" example on a small grid. This is "stamped" onto a larger grid using the map overwrite operator, #

```
grid = emptygrid # glider
```

A # B is A except that where A and B intersect, the elements of B are substituted. The same technique is used to put the boundary back on the result of `nextgrid`. Sharing analysis can be expected to eliminate the extra arrays.

The body of function `iterate` is a loop controlled by a test rather than a generator. It shows how loop values (`grid` and `i`) can be used to express data dependence between loops. Note that, despite the presence of the imperative looking:

```
new i = i + 1
```

this is still pure functional, referentially transparent code. There is one `new i` for each instance of the loop; `i` cannot be reassigned, and neither can `new i` once it has been given its value. This *single assignment rule* is characteristic of dataflow languages, which have to deal with numeric problems and so frequently use loops rather than recursion. It will also have an important part to play in the semantics of stateful objects.

Array indexing is a special case of slicing, which can be used to select arbitrary (rectangular) subarrays of an array. For instance, an alternative way to calculate the number of neighbours is

```
neighbours = sum( G[i-1..i+1, j-1..j+1] ) - G[i,j]
```

Assuming `sum` is a function which sums the elements of an integer array.

As another example, the following loop generates an array of n random numbers in the range 0 to 1:

```
rands( init_seed: Int; n: Int ): array[Real] is
initially
  seed = init_seed;
  i = 0
while i < n do
  new seed = (13849 + 27181 * seed) mod 65536;
  new i = i + 1;
  rand = new seed / 65536.0
return rand
od
```

4 Asynchronous communicating objects

4.1 Why state?

It is worth reflecting that functional UFO is 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 often be unnecessary.

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 could be done by defining UFO to have lazy semantics, or by replacing type classes by more conventional classes and inheritance in Haskell. [13]

However, the author believes 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. UFO therefore has stateful objects, and the design is targeted towards minimising the problems this causes, both by the way the runtime behaviour of objects is defined, and in the type system.

4.2 A simple example

The following defines a random number generator class:

```
stateful class Random( init_seed: Int ) is
  { initial seed = init_seed }

  ** Return a random real number between 0 and 1
  proc nextrand : Real is
  do
    new seed = (13849 + 27181 * seed) mod 65536;
    return new seed / 65536.0
  od

  ** Return a random integer between lower and upper
  proc rand_between( lower, upper : Int ): Int is
  do
    range = upper - lower + 1;
    return lower + trunc(!nextrand * range)
  od
end ** Random
```

A stateful class typically has one or more *instance variables*. It may still have functions, which may refer to the current values of those variables, but it will also have *procedures* which may update them. The body of such a class starts with an *initialisation block*, which gives initial values for the instance variables.[14]

[13] This would require a solution to the conflict between laziness and dynamic binding pointed out above, though.

[14] It may also define constant instance *values*. These differ from parameterless functions in that they are created once when the object is created, rather than being repeatedly evaluated. In functional UFO, this only matters semantically in terms of termination properties, but if an instance value refers to a stateful object, the difference is critical.

Procedure calling is distinguished by the use of `!`, rather than the dot used for function calls. The body of a `proc` is similar to the body of a loop, and this similarity is deliberate. The single assignment rule applies, in that each procedure body may update each instance variable at most once. The distinction between `seed` and `new seed` is exactly that in the loop example.[15] Of course, in this case, it does not maintain referential transparency, as calls to a `Random` object's `nextrand` procedure may occur from anywhere with a reference to that object, in a nondeterministic order. What it *does* do is to allow parallelism to be exploited within the body of a procedure, while maintaining internal consistency with respect to the instance variables; all references to `seed` within the same method invocation will see the same value. This scheme removes the need for any statement sequencing even within procedure bodies; *sequencing is still by data dependence only.*

This, of course, leaves the problem of how interference *between* method calls to stateful objects is prevented. This mechanism is as follows: such method calls (procedures and functions) are held in a FIFO queue. When it gets to the head of a queue, a method "locks" the object. Unlocking occurs as soon as all instance variables which are updated have been given their new values. There are three sets of instance variables to consider:

1. R, the set of (old) instance variables referred to by the method.
2. U, the set of instance variables updated by the method, if it's a procedure, by statements `new x = ...` in the top level of the procedure body. R and U may or may not intersect.
3. NR, the set of new instance variables referred to by the method, i.e. expressions of the form `new x`. $NR \subseteq U$

U is statically obvious, as are upper bounds for R and NR, which is all that's required. To ensure coherence, a method starts by taking (reference) copies of its R variables, and uses these copies for all R references. The values at the start of the method's execution are therefore seen, whether or not it (or some other method executing concurrently) updates them.

Completion of the assignments to the U variables triggers unlocking, which therefore does not necessarily imply either termination or the production of a result. On unlocking, a procedure may need to take copies of its NR variables, if it still needs to refer to them, to ensure that the correct `new` value is seen.

This very weak sequencing has several consequences:

- A function call locks the object only transiently, to take copies of its R variables. It therefore causes very little serialisation and cannot cause deadlock. Arbitrary numbers of functions can execute in parallel, and their execution does not prevent the execution of a procedure.

[15]This idea is not as novel as we first thought. Arvind's managers, dating from 1978, work the same way. However, he used explicit merging of input streams.

- A procedure call unlocks the object at the earliest possible moment, when coherent access to the instance variables is guaranteed. The call may not return its result until much later.

- Conversely, it is possible for a method to return a result before it has terminated, since it may make procedure calls which do not return values necessary to produce the result. This and the previous property are usually considered characteristic of a "concurrent object-oriented" language.

- The distinction between "synchronous" and "asynchronous" method calling/message passing is rather meaningless in this context, since we are not dealing with communicating sequential threads at the language level. An implementation in terms of such threads will need to make some calls asynchronous in order to preserve the semantics. In general, such an implementation will need to ensure that it creates no spurious data dependencies which could cause deadlock; even a sequential implementation may need to operate in a pseudo-parallel manner.

- Since locking exists only for long enough to ensure coherent access to instance variables, and sequencing is only by data dependence, it follows that the only way that deadlock can occur is by cyclic dependencies between instance variables. The hope is that this should be relatively easy to reason about.

4.3 Introducing extra sequencing

The main problem with this approach is that it leaves the programmer with too little control over sequencing in some cases. Examples where this may be a problem are:

- If we want to ensure coherence of a whole substructure referred to from an object, not just the instance values. This is the traditional "granularity of locking" problem: UFO provides a very fine granularity, from which coarser grain solutions may be programmed.

- If we want to ensure that one event "happens before" another, even if they are not data dependent. Controlling the order of display in a user interface might be an example.

- If we want to reason about some fairness property of an algorithm, not just termination/deadlock freedom, or if an unconstrained solution has too much nondeterminism to be reasoned about effectively.

The following, very artificial, example shows how extra sequencing is achieved.

```
stateful class counter( init: Int )
  { initial count = init }
```

```
  proc inc( n: Int ): Void is do new count = count + n od

  read : Int is count
end

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

`sillyadd` is intended to return `a + b`, but since statements are not sequentialised, the read may occur before either or both increments. The key to fixing this, while retaining weak sequencing, is to notice that, once a set of method calls are queued, they are sequenced, so it is only necessary to control the order of queueing, *not* to wait for termination. The extra sequencing is achieved by *introducing extra data dependencies*. The syntax for expressions and method calls is extended as follows

```
'after' exp1 exp2

call ['ack' name]
```

The `after` clause delays the evaluation of exp2 until after that of exp1. The `ack` clause returns the named value *when the call has been queued.*[16] Since the queues are FIFO, the order of queuing determines the order of execution (except when conditional message acceptance, discussed below, is used, in which case this ordering is only partial). So the example is fixed as follows:

```
sillyadd( a, b: Int) : Int is
{ c = Counter(0);
  c!inc(a) ack inc1; c !inc(b) ack inc2
  return after (inc1, inc2)  c.read;
}
```

Note the use of a tuple to collect acknowledgements. Of course this only works because of strict evaluation.

Unlocking of an object can be delayed simply by introducing an extra variable (called, by convention "unlock") which is not assigned until the required conditions have been met.[17] This can be used to control granularity of locking.

[16]No assumption is made about messages being received in the order they are sent. However, on a machine where this is true, the implementation can optimise out some of the explicit acknowledgements this construct requires

[17]A smart implementation may notice that **unlock** is a write-only variable, and so not bother to store it! It may be better to give **unlock** a special status, like **accept**, below, but in principle it's just an ordinary instance variable.

4.4 Conditional message acceptance

In the above example, the method using the counter controlled the order in which it queued messages. It is more in line with object-oriented philosophy to allow an object to control its own queue as far as possible. UFO, like other concurrent object-oriented languages, therefore has *conditional message acceptance* (CMA). A procedure may specify a new message acceptance list.[18] The next method to execute on the object will be the first in the queue which is on the list.

Consider the simplest possible stateful class:

```
stateful class Loc[T]( )
  { initial val = error:T }

  read is val;

  proc write( v: T ) : Void is do new val = v od
end
```

This behaves as a simple, updatable store location, and can be implemented very efficiently as such. It obviously needs to be used with extreme caution in our highly parallel world!

A safer sort of variable location is one which behaves like a location in an I-store in a dataflow machine [12, 13], i.e. it is written once only, and reads queue until the write has occurred:

```
stateful class ILoc[T]( ) inherits Loc
  ** val and read unchanged
  { initial accept = << write >> }

  proc write( v: T ) : Void is
  do new val = v; new accept = << read >> od
end
```

The pseudo-variable `accept` determines the current set of acceptable messages. Its type is a special type `Message_acceptance_list` (doing it this way ensures that updating the list has the same consequences as other updates - a function can't do it, for instance. It also allows the list to be set conditionally). Initially, only writes are accepted, so reads are blocked until a write has been done. In this very primitive I-store location subsequent writes are blocked for ever. More sophisticated versions can be obtained by inheritance, e.g.

```
stateful class ILoc_with_extract[T]( ) inherits ILoc
  ** Initialisation and read as before
```

[18] A function may not do so, because it would be altering the state of the object.

```
  proc write( v: T ) : Void is
  do new val = v; new accept = all but << write >> od

  proc extract : T is
  do new accept = << write >> return val od
end
```

`extract` doesn't bother to "delete" `val`, as the next thing to get in will be a `write`.

On a machine with hardware support for I-store style operations, these classes can still be transformed into something very efficient.

CMA provides a general conditional synchronisation mechanism, arguably cleaner than (for instance) semaphores. In doing so, it complicates the, previously very simple, story about deadlock given above. The queuing mechanism becomes more complex, although research in implementing, for example, ABCL [4], which has the same problem, suggests that it is not serious.

5 The type system

UFO has strong static typing with genericity, overloading, and inheritance polymorphism. There are enough mandatory type declarations, and the rules about redefinition on inheritance are sufficiently conservative, to ensure that only relatively simple type checking (with only trivial inference) is required.

The type system is not the subject of this report, but one aspect should be mentioned, namely the way in which it gives safeguards in the handling of stateful objects.

Two requirements are imposed statically:

5.1 Any reference to an updatable object can be detected statically

This is based on the fact that only objects of stateful classes can be updated. A *class type* is a type of the form $\mathrm{C}[\mathrm{AG}_1..\mathrm{AG}_n]$ where C is the class name and $\mathrm{AG}_1..\mathrm{AG}_n$ are actual types corresponding to the generic parameters (if any) in the class definition. The class C is said to be the *determining class* for the type. In what follows, the distinction between classes and types is sometimes blurred for brevity.

An *updatable type* is defined as follows:

- A class type whose determining class has procedures.
- A class type with an updatable instance value.
- An array whose base type is updatable.
- A restricted generic type $\mathrm{G} \leftarrow \mathrm{U}$, where U is an updatable type.

An unconstrained generic type is not considered updatable, even though it could be instantiated to an updatable type, because the updating procedures cannot be used in that context.

One restriction is required. Normally, if C and D are class types, and D inherits from C, then D conforms to C. However, if D is a stateful class and C is not, the conformance no longer holds. Such inheritance is still legal, and can be used to reuse implementation, but a reference to a C cannot at runtime actually refer to a D.

5.2 A function may not modify its arguments

This is not merely a parameter passing convention. We require that, if an object is passed to a function, no part of the structure referenced from that object can be modified by the function. This enables the programmer and the implementation to reason about functions much as in a pure functional language, even in the presence of stateful objects.

This is achieved as follows. For an updatable type T, there is a corresponding type Readonly[T], defined as follows

- If T is a class type C[T_1..T_n], Readonly[T] is C'[Readonly[T_1]..Readonly[T_n]], where C' is the class obtained by deleting all procedures of C and converting the types of all instance values and variables of C to be Readonly.
- A restricted generic type G←C becomes G← Readonly[C].
- Array[T] becomes Array[Readonly[T]]

The idea is as follows:

1. When a value of updatable type T is passed as an argument to a function, it is statically converted to Readonly[T]. This means that the function can't directly update it, because its procedures have disappeared.
2. T conforms to Readonly[T] (so that the argument passing works), but Readonly[T] does not conform to T. This prevents a Readonly[T] from being converted back to an ordinary, updatable, T, in various sneaky ways, e.g. by passing it through an "identity function" returning a T or binding it to a value name of type T.

So, for instance in the random number generator, both `nextrand` and `rand_between` must be declared as procedures. `nextrand` modifies `seed` directly. Within `rand_between` the call `!nextrand` would be illegal if `rand_between` were a function, because it is equivalent to `self!nextrand`[19] and `self`, as a `Readonly[Random]` could not be an argument to `nextrand`.

Although this is expected to make reasoning about functions easier, it does not provide full referential transparency.

[19] As far as the type system is concerned. The effect of omitting `self` is to avoid going through the queuing mechanism again

1. A function may return a result of updatable type, so that calling a function twice with the same arguments is different from sharing the result of one call. This can be prevented by a static restriction, but such a restriction would be overly severe.

2. Since functions can create and manipulate stateful objects, they may have nondeterministic behaviour. In some cases, this is desirable. For instance a function taking the intersection of two large sets using some clever parallel algorithm might reasonably return the elements of the result set in a nondeterministic order. There is no way that the type system can be expected to distinguish "reasonable" from "unreasonable" nondeterminism.

As an example of the usefulness of the Readonly restriction, consider the `map` function defined earlier. Since the parameter `f` must be a function, it cannot modify the list elements. The result of a `map` is therefore deterministic, even if the elements of the list are (possibly shared) references to stateful objects.

6 Further examples

A variety of further examples of UFO programs follow. The first two are purely functional, one numeric, one text processing. The remaining examples demonstrate stateful objects, CMA etc. They include classic problems such as wavefronts and dining philosophers, and also a "nondeterministic parallel problem solver".

6.1 Matrix multiplication

Since the earlier example of arrays and loops was not numeric, a more typical numeric example should be included.

```
typevar Num <- Numeric

matmult( a, b : Array[Num] ): Array[Num] is
{
  for       i in [lower(a,1) to upper(a,1)]
      cross j in [lower(b,2) to upper(b,2)] do
  cij =
    for a_element in a[i,..] dot b_element in b[..,j] do
    c_element = a_element * b_element
    return sum(c_element)
    od
  return cij
  od
}
```

Points of interest:

- The function will cope with matrices of any numeric type. However, this need not imply dynamic binding, as the implementation can generate specialised versions of `matmult` for different instantiations of the type variable, particularly for Ints and Reals.

- The standard functions `upper` and `lower` inspect the bounds of an array in a particular dimension.

- The inner product loop uses a `dot` product generator, which generates an array shaped according to the intersection of the two generators. In this case, if the matrices obey the "domino rule", the two slices will be the same size. If not, a smaller (possibly empty) result array will be produced.

- The use of slices avoids any array accessing in the innermost loop.

- The implicit coercion of loop values to arrays in the `return` part takes a little getting used to. It may be useful to define a dummy identity function `array_of` so that, for instance, the outer loop can end with `return array_of(cij)`.

A general version of the `sum` function may be written:

```
sum( a: Array[Num] ): Num is
  foldr( a, Num| + |Num, Num|zero )
```

Note the explicit disambiguation required for `+` and `zero`. In practice, an implementation should specialize versions of such functions for `Int`s and `Real`s.

6.2 Cross reference generation

The following program is a transliteration of a Haskell version written by David Lester.[20] It illustrates typical pure functional programming style using higher-order functions.

The problem is: given a piece of text (e.g. a program), and a list of words deemed to be "exceptions" (e.g. the keywords of the programming language), generate a listing of all words in the text, excluding the exceptions. Each word should appear once in the output list, together with the numbers of all the lines on which it appears in the text. The words should be sorted, as should and the list of line numbers for each word.

The solution uses an ordered binary tree to store the words, each with its list of line numbers. Points of interest are the use of `foldr` to recursively insert all the words into the tree, and the use of partial application (`enter(l,_,_)`) to create a function which enters a number of words with the same line number.

```
xref( words, exceptions: String ): String is
{
input_lines = words.lines.map( alphas );
```

[20]Before that it was a C program by Ian Cottam. Before that, a VDM spec and Pascal programming exercise in [18]!

```
ids_per_line = zip( List([1 to input_lines.length]), input_lines );
tree = ids_per_line.foldr( entries, null : Tree );
without_exc = exceptions.lines.foldr( delete, tree )
return without_exc.display
}

entries( tree: Tree; line: (Int, List[String])): Tree is
{
l, ws = line
return ws.foldr( enter(l,_,_), tree )
}

alphas( line: String ): List[String] is
if line.is_null then null
elseif line.hd.isAlpha
then
  { word, rest = line.span( isAlphanum )
    return word.map( toLower ) :: rest.alphas
  }
else line.tl.alphas
fi

Leaf is null:Tree

class Tree( word: String; lnums: List[Int]; l, r : Tree )
enter( n: Int; w: String ):Tree is
  nullcase : Tree( w, List1(n), Leaf, Leaf );
  otherwise :
     if w < word then Tree( word, lnums, l.enter(n, w), r )
     elseif w > word then Tree( word, lnums, l, r.enter(n, w) )
     else Tree( word, n :: lnums, l, r )
     fi

delete( w: String ): Tree is
  nullcase : Leaf;
  otherwise :
     if w < word then Tree( word, lnums, l.delete(w), r )
     elseif w > word then Tree( word, lnums, l, r.delete(w) )
     else Tree( word, null, l, r )
     fi

display: String is
  nullcase : ""
  otherwise: l.display ++ word.display(lnums.unique) ++ r.display
end ** class Tree
```

```
display( word: String; lnums: List[Int] ): String is
if lnums.is_null then ""
else word ++ printlist( lnums.map(print), ", " )
fi

main( words: String ): String is xref( words, readfile( "exceptions" ) )
```

The program[21] took 9 seconds to xref a 3500 line file on a SPARC2 machine. This was about 3 times slower than a C version: a little disappointing but acceptable for a prototype implementation.

6.3 Dining philosophers

The following is a solution to the classic dining philosophers problem using a boring but sensible solution from [16]: only four of the philosophers are allowed in the room at once.

```
** Enumerated type with one element, for signalling completion.
enum Signal { sig }

class Philosopher( doorman: Doorman; lfork, rfork: Fork )
  proc live is
  do
  while true do
    thought = think( ... );
    permit = after thought { doorman!enter }
    after permit
        { f1 = lfork! pickup;
          f2 = rfork! pickup;
          eaten = eat(f1,f2)
          after eaten { lfork!putdown; rfork!putdown; doorman!leave }
        }
    od
  od
end

class Doorman()
  { initial occupants = 0;
    initial accept = all
  }

  proc enter: Signal is
  do
  new occupants = occupants + 1;
  new accept = if occupants == 4 then << leave >> else all fi
```

[21]Actually a rather less elegant version written in UFO0

```
  od

  proc leave is
  do
  new occupants = occupants - 1;
  new accept = all
  od
end ** Doorman

class Fork()
{ initial accept = << pickup >> }

  proc pickup: Signal is
  do
  new accept = << putdown >>
  return sig
  od

  proc putdown : Signal is
  do new accept = << pickup >> return sig od
end ** Fork

proc dining_philosophers : Void is
do
bouncer = Doorman;
fork =
  for i in [0 to 4] do f = Fork return f od
philosophers =
  for i in [1 to 5] do
    p = Philosopher( bouncer, fork[(i-1) mod 5], fork[(i+1) mod 5] );
    p! live
  od
od
```

Exclusive access to the forks is provided by CMA. Since the philosophers lead rather serial (as well as tedious) lives, several uses of `after` are required. The block:

```
{ lfork!putdown; rfork!putdown; room!leave }
```

may be evaluated in such a way that the call to `leave` may be executed, and the next philosopher enter the room, before the forks have been put down. However, since the forks will be put down eventually, no deadlock can occur.

The uses of the dummy signal value may seem artificial, but in a realistic application, the forks would be resources required by `eat`, and `eat` and `think` would be computations returning results. In fact, unless thinking and eating are data dependent, or require conflicting resources, a UFO philosopher ought to do both at once: in this respect the example itself exhibits "serial-think".[22]

6.4 A queue

```
** A one-way link with an updatable next field.
stateful class Link[Alpha]( first: Alpha )
{ initial nextlink = null:Link[Alpha] }

next is nextlink;
proc set_next( y: Link[Alpha] ): Void is do new nextlink = y od
end

** The queue class itself.
stateful class Queue[Alpha]( )
{ initial head = null:Link[Alpha];
  initial tail = null:Link[Alpha];
  initial qlength = 0;
  initial accept = << enqueue, length >>
}

proc enqueue( val:Alpha ) is
do
  newlink = Link( val );
  new head, new tail =
    if head.is_null
    then
     ( newlink, newlink )
    else
      { tail!set_next(newlink) ack tailset;
        return after tailset (head, newlink) }
    fi;
  new qlength = qlength + 1;
  new accept = all
od

proc dequeue: Alpha is
do
new head = head.next;
new qlength = qlength - 1;
new tail, new accept =
    if new qlength == 0
    then (null:Link[Alpha], << enqueue, length >> )
```

[22]Of course if the example is regarded as a literal simulation, the UFO program must model any serial constraints in the situation being simulated.

```
    else (tail, all)
    fi
return head
od

length : Int is qlength
end ** Queue
```

This is a rather low-level example, and illustrates how care is necessary when dealing with nested structures; in `enqueue`, the new head and tail pointers should not be released until the message which sets the new tail of the next-to-last element has been safely queued. Once again, the importance of encapsulation and code reuse is stressed: you don't want to have to do this sort of thing too often.

In the long term, it is hoped to develop formal methods for proving that low-level library objects like this are correctly implemented.

6.5 A general "nondeterministic parallel problem solver"

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

1. The queue, as above
2. An "active" processor object which, when activated, repeatedly takes items from a source queue, 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 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 these components, 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 wide range of such "standard parts".

A good UFO implementation should be able to distribute the processor objects sensibly across the physical processors, without the need for explicit placement by the programmer.

```
stateful class Processor( source: Queue[T]; dest: Queue[U]; f: TtoU );
{ initial  accept = << activate >> }

proc activate is
do new accept = << continue, deactivate >>; self!continue od
```

```
proc continue is
do
dest!put( f(source !get) );
self!continue ** Queue behind any waiting deactivate.
od

** Effectively kills the object and allows it, and any incoming messages,
** to be garbage-collected.
proc deactivate is do new accept = << >> od
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;

findasolution( candidates: Array[Candidate]; numprocs: Int;
         null_solution: Solution; try: Candidate -> Solution): Solution
is
{
  ** Create input and output Queues.
  input_queue  = Queue[Candidate];
  output_queue  = Queue[Solution];
  ** 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 if it's prime, and Bignum(0) otherwise.
```

Notice that the deactivation of a processor object requires its cooperation. The alternative is to put pre-emption into the language, as is done in ABCL. This would be very difficult to implement in the everything-parallel context of UFO (it works in ABCL because that language has a notion of sequential threads), and would make programs more difficult to reason about.

6.6 Wavefronts and non-strict evaluation

Some problems can only be formulated in a functional style if non-strict evaluation is assumed. An example is the "wavefront" problem, where a computation proceeds from the top left corner of a 2D array in a wavefront, where the value of each element is some function of its north, west, and northwest neighbours. Such a problem has O(n) parallelism for an n by n array. The I-store location shown above can be used to simulate such non-strict evaluation.

```
type Nonstrict_array = Array[ILoc[Real]];
type func = ( Real, Real, Real ) -> Real;

wavefront( m,n: Int; initial: Real; f: func ): Array[Real]
is
{
** Initialise the array
a =
  for i in [1 to m] cross j in [1 to n] do
  emptyij = ILoc[Real]
  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 = a[i,j].read
  return aij
  od
}
```

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

6.7 Futures

A final example illustrating inheritance, higher-order functions, and CMA, is a class which represents an arbitrary future (or thunk or closure).

```
class Future[T]( f : Trigger -> T ) inherits ILoc
  ** val, read and write as for ILoc, although direct writes will never
  ** be accepted.
  { initial accept = << force >> }

  proc force is do new val = f(trig); new accept = << read >> od
end
```

`Trigger` is an enumerated type with one element called `trig`. A suitable f can be generated by partial parameterisation, e.g.

```
trig_rands( init_seed: Int; n: Int; t : Trigger ) is rands( init_seed, n )
...
future_rands = Future( trig_rands( 12345, 100, _ ) )
```

This can be used to build lazy lists and so on. There is therefore no need to include a special future construct in the language.

7 Other features

This report has not attempted to provide comprehensive coverage of the language. Some features which do not appear in the examples above are briefly mentioned here.

7.1 The type system

No attempt is made to give a full description of the type system here; a separate report is in preparation to do this. Apart from the aspects mentioned above, the following are worthy of note:

- Inheritance does not necessarily imply subtyping: a class only conforms to its ancestor if certain conditions hold. The case of stateful descendents of stateless classes was mentioned above. Another occurs where a class changes the interface it inherited (by making instance values or methods private). It is also possible to declare a class `closed`, which switches off conformance to ancestors and descendents, thereby avoiding any possible overhead due to dynamic binding.
- There are a set of rules which define what forms of redefinition are allowed on inheritance. These have currently been made quite restrictive, to avoid complex global type checks. Experience will show whether some of these restrictions should be relaxed.
- There are a set of rules which govern when explicit typespecs may be omitted. These are designed to allow omission of typespecs which are "obviously" redundant.

7.2 Other array features

There is a fair amount of syntax for creating arrays, and there are various standard functions on arrays. For instance, there is a function `shaped_like` whose output is the result of coercing one array to be the same shape as another.

There is one other way of combining generators, apart from `dot` and `cross`. A `join` generator is like `cross` in that the loop cycles correspond to the cartesian product of the generators. However, the result is one-dimensional, and the generators need not be independent: a name defined by the first can be used in the second.

A loop creating a 1-D array may also have a `when` clause which filters out values other than those satisfying some predicate. For instance, the following finds all integers `a, b, c` less than `n` such that $a^2 + b^2 = c^2$:

```
for a in [1 to n] join b in [a+1 to n] join c in [b+1 to n]
do
when a*a + b*b == c * c
return (a,b,c)
od
```

In languages with a "list comprehension" syntax, the above could be written:

```
[ (a,b,c) | a <- [1 to n], b <- [a+1 to n], c <- [b+1 to n],
            a*a + b*b == c * c ]
```

Provision of this more concise notation, implemented by the obvious syntactic transformation to the loop form, is being considered.

One other comment about arrays. It is possible to use arrays simply as collections with fast indexing, and not worry about their shapes. This is done simply by avoiding the use `cross`, so that all arrays are 1-D.

7.3 I/O

I/O is defined by standard library classes and methods (`File, read, write`, etc.) rather than in the language. Simple I/O (producing a result, reading from files etc.) can be done purely functionally. Since UFO has strict semantics, more complex I/O and interaction has to be done procedurally.

For UFO1, I/O will be limited to text strings (with the obvious standard functions for numbers etc.). The issue of persistence in a parallel environment will be tackled in the design of UFO2.

7.4 Enumerations and case expressions

Despite the comments on explicit selection in section 2.5, UFO does in fact have enumerated types, and a case expression, e.g.

```
enum Colour = { red, green, blue, black }

...

case colour of
  red, green : ...
  otherwise: ...
esac
```

The case expression is simply syntactic sugar for nested `if-then-else` expressions. It is expected that these features will be used relatively rarely, but there are occasions where distinguishing between a set of simple values by creating a class for each of them is unnecessarily tedious.

8 Current status

A subset of UFO (UFO0) has been implemented, and used to write its own compiler. A full implementation of UFO1, the language described above, will be completed within the next few months.

UFO is not a minimalist language, and a good parallel implementation, together with the tools, libraries etc., necessary to make the language really useable, will involve a substantial effort. Such an effort should also include the definition of a formal semantics for the language, and a formal development methodology to allow proofs of correctness. Such an effort seems to us highly worthwhile given the limitations of current implicitly parallel languages.

A number of issues have deliberately not been addressed in the design of UFO1. These include full exception handling and persistence. It is hoped that after a couple of years' experience with UFO1, a review will take place with the aim of defining a UFO2 which does address these matters, and any others which arise in the meantime.

Acknowledgements

Manchester provides a rich, diverse, and therefore healthy environment for the birth of a new programming language. Many colleagues have made helpful comments, from many different perspectives. Mario Wolczko reviewed this report and several previous ones, and has commented on many aspects of the language design. Chris Kirkham did much of the work on the array facilities, and has been closely involved throughout the development of UFO. Cliff Jones, Ian Watson and Dave Lester have contributed to stimulating discussions. Outside of Manchester, Wim Böhm of Colorado State University and Aki Yonezawa and the ABCL group at the University of Tokyo, have provided valuable input.

Financial support for the author has been provided directly by the Department of Computer Science at the University of Manchester

References

[1] J. R. McGraw, S. K. Skedzielewski, S. J. Allan, R. R. Oldehoeft, J. Glauert, C. C. Kirkham, W. Noyce, and R. Thomas: **SISAL: Streams and Iteration in a Single Assignment Language**, Reference Manual 1.2, Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, 1985.

[2] R.S. Nikhil: **ID Reference Manual**, CSG memo 284, MIT Laboratory for Computer Science, 1988.

[3] G. Agha: **Actors: A Model of Concurrent Computation in Distributed Systems** MIT Press Series in Artificial Intelligence, 1986.

[4] A. Yonezawa (ed.): **ABCL, an Object-Oriented Concurrent System**, MIT Press Computer Systems Series, 1990.

[5] 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.

[6] K. Ueda, T. Chikayama: **Design of the Kernel Language for the Parallel Inference Machine**, Computer Journal 33(6), 1990, pp494-500.

[7] D. C. Cann **Retire Fortran? A Debate Rekindled**, Communications of the ACM 35(8), August 1992, pp 81-89.

[8] P. Hudak et al: **Report on the Programming Language Haskell, Version 1.1**, Univerity of Yale tech. report, August 1991.

[9] A. P. W. Böhm, D. C. Cann, J. T. Feo, R. R. Oldehoeft: **SISAL 2.0 Reference Manual**, tech. report CS-91-118, Computer Science Dept. Colorado State University, 1991.

[10] B. Meyer: **Object-Oriented Software Construction**, Prentice Hall 1988.

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

[12] Arvind, R.E. Thomas: **I-structures: An Efficient Data Type for Functional Languages** MIT/LCS/TM-178, Computer Science Laboratory, MIT., Cambridge, MA, 1981.

[13] J. Sargeant, C.C. Kirkham: **Stored Data Structures on the Manchester Dataflow Machine** Proc. 13th Annual Symposium on Computer Architecture, 1986, pp 235-242.

[14] A.P.W. Böhm, J. Sargeant: **Code Optimisation for Tagged-Token Dataflow Machines**, IEEE Transaction on Computers, 38(1), pp. 4-14, Jan. 1989.

[15] R. Bird, P. Wadler: **Introduction to Functional Programming**, Prentice Hall, 1988.

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

[17] M. R. Sleep, J. R. Kennaway, **The Zero Assignment Parallel Processor (ZAPP) project**, in D.A. Duce (ed.) Distributed Computing Systems Program, Peter Peregrinus, London, 1984, pp 250-267.

[18] V. J. Bush, I. D. Cottam, J. T. Latham **The Programming Process: an introduction using Pascal and VDM**, Addison Wesley 1989.

[19] S.L. Peyton Jones, P. Wadler **Imperative functional programming**, to appear in POPL '93