The Uflow Computational Model and Intermediate Format

UMCS Technical Report 94-5-1[1]

John Sargeant, Chris Kirkham, Steve Anderson
Dept. of Computer Science,
University of Manchester,
Manchester M13 9PL,
England
{js,cck,andersos}@cs.man.ac.uk

Abstract

This report motivates and defines a general-purpose, architecture independent, parallel computational model, which captures the intuitions which underlie the design of the United Functions and Objects programming language.

The model has two aspects, which turn out to be a traditional dataflow model and an actor-like model, with a very simple interface between the two. Certain aspects of the model, particularly strictness, maximum parallelism, and lack of suspension are stressed. The implications of introducing stateful objects are carefully spelled out.

The model has several purposes, although we largely describe it as it would be used for visualising the execution of programs.

The model is embodied in a textual intermediate format, and in a set of UFO data structures. This report also serves as a definition of the intermediate format, and gives a brief overview of the data structures.

1 Introduction

This report serves two purposes. Firstly, in sections 1 to 9, the Uflow computational model is motivated, and its key features and properties are described. Secondly, in sections 10 to 12, the textual intermediate format associated with the model is defined, and the corresponding internal data structures are briefly outlined. Readers not interested in this level of detail may skip these sections (or skip 10 and 11) without missing any important concepts.

Recent technical reports issued by the Department of Computer Science, Manchester University, are available by anonymous ftp from ftp.cs.man.ac.uk in the directory /pub/TR. The files are stores as PostScript, incompressed form, with the report number as filename. They can also be obtained on WWW via URL http://www.cs.man.ac.uk/csonly/cstechrep/index.html. Alternatively, all reports are available by post from The Computer Library, Department of Computer Science, The University, Oxford Road, Manchester M13 9PL, U.K.

1.1 Properties required of the model

We seek a computational model for general-purpose parallelism. The purposes of such a model are:

- As a static, architecture-independent, representation for parallel programs, to be used by compilers.
- As a dynamic, architecture-independent representation for parallel programs, to be used by visualisation, debugging, and profiling tools.
- As the basis of a real high-performance parallel implementation, on suitable architectures.

While the above aims are clearly aggressive, we believe they are possible, despite the numerous failed attempts which litter the history of the subject through the 80s. However they impose stringent constraints on the form of the model.

1. It should be intuitively natural, so that people can use tools based on it.
2. It should allow the expression of parallelism at any level of granularity.
3. It should allow expression of locality and modularity in parallel computation. In particular it should allow local synchronisation to be expressed naturally.
4. It should be capable of efficient implementation, at least in principle, either directly on suitable specialised architectures, or by transformation to code appropriate for conventional machines.
5. It should have a clean mathematical description and be amenable, at least in principle, to formal reasoning.

A few points should be made about our order of priority here. We place intuitiveness as a higher priority than mathematical reasoning. This is partly a belief that an intuitively simple and elegant model will have a clean mathematical description, partly an unwillingness to be tied down by the limitations of existing pieces of mathematics which may have been developed historically for other purposes, and partly a recognition of the limitations of the role of mathematics in software development in general. Having said this, we believe that the model described below should be highly amenable to mathematical reasoning.

The stress on intuitiveness also implies that the model cannot be totally language-independent; the programmer must be able to relate what is going on to the source program. The design of such a model therefore reflects a set of choices about what is the "right" sort of programming language for general-purpose parallel computation. Our choices are embodied in the United Functions and Objects programming language, whose main characteristics are summarised in section 2.

1.2 Models considered unsuitable

1.2.1 Communicating sequential threads

Many models can be eliminated if these principles are accepted. We would first propose to eliminate *any* model based on communicating sequential threads(CST). This is clearly a radical and controversial step, as most existing parallel computation is expressed in terms of such threads. In our view, the most fundamental question in this whole field is whether the best way to view parallel computation is in terms of such threads, or in terms of some more radical, more "totally parallel" model.

We would argue against CST on a number of grounds, most which stem from the fact that a CST program contains discontinuities between the insides of threads, which execute serially, and the communication between them, which is where the parallelism is. The placings of these discontinuities is, in practice, often arbitrary, and changing them may change the semantics of the program. Threads may be related to objects in a concurrent OOP style, but this is at best a marriage of convenience: we can see no good reason why the unit of encapsulation should also be the boundary between serial and parallel execution. Indeed, the whole notion of a "thread of control" is an artificial one, an artifact of the way serial computers work rather than a property which is readily identifiable in many real-world objects. Note that we are not ruling out active objects - ones which do autonomous computation rather than merely reacting to incoming messages - but we are questioning the assumption that such autonomous computation should be equated to a thread of control.

The best size for a thread is machine-specific and may vary widely; clearly this makes it undesirable for the programmer to determine the thread boundaries explicitly. On the other hand, if the compiler does it, there is a problem mapping the partitioned code back into the form the programmer knows about. If thread sizes are statically determined, poor performance results for irregular programs; on the other hand the use of techniques such as dynamic granularity and lazy task creation [Mohr90] makes the dynamic behaviour very complex. For instance [SaWa91] shows how the behaviour of a very simple program under such a model can vary dramatically with subtle changes in the exact details of how threads are spawned.

What we attempt to do is to find the best of the alternatives, and to correct the imbalance in the literature, which mostly assumes a CST model as the only one. Our aim, then, is to find the best alternative to CST, although on conventional machines we may well implement our model using CST techniques.

1.2.2 Graph reduction

Graph reduction, and graph rewriting techniques in general suffer from well known efficiency problems, but in our view the lack of intuitiveness (principle 1) is a more fundamental problem. For straightforward strict functional computation, graph reduction is a reasonable representation, but there are simpler ones. For more general computations (lazy or stateful), graph rewriting provides a very general representation, but the graph transformations are low-level and difficult to visualise and, in the case of laziness, the execution order is even more so.

1.2.3 BSP models

Bulk Synchronous Parallel models (e.g. PRAMS [Vali90]), are not really computational models in the sense we are considering. There are clearly many sorts of parallel computation for which such models are not the appropriate way to visualise them. The biggest problem, though, is with principle 3. We consider it inappropriate to take as our fundamental model of parallel computation one which takes no account of locality, information hiding, modularity, object-orientation etc. Of course there are many useful and efficient parallel *algorithms* based on BSP models and other less sophisticated global models, such as SIMD, and so it is important that we should be able to simulate such models efficiently.

1.3 So what is left?

We only know of two existing models which may meet the criteria:

1. A dataflow model
2. An actor model

We do not have a completely new model to propose, and we would be surprised to discover a completely new model which meets the above constraints, so we are left with these two possibilities, or some combination. Before we explore this further, we need to briefly describe the key ideas of the UFO language.

2 United Functions and Objects

Space permits only a listing of the key ideas here. A few self-contained examples appear later, but readers wishing to learn about the UFO language in detail should consult the UFO papers [Sarg93a/b/c]. The key features of UFO are:

- Parallelism by default; sequencing is by data dependence, and by queueing for access to stateful objects, only. This implies that the computational model is *not allowed* to hide parallelism and also that it needs local synchronisation.
- A well-defined subset of UFO is a higher-order functional language, but with the OO notions of classes and inheritance.
- Selection by pattern matching, as used in most functional languages, is restricted to simple cases, for software engineering reasons, and OO-style selection via dynamic binding is used instead.
- UFO has integrated functional arrays and loop structures, in the style of SISAL [McGr85], but significantly generalised.
- UFO has stateful objects, with updateable instance variables, which are accessed by a queuing mechanism like actors [Agha86]. A single-update scheme provides internal coherence; again this is similar to some actor systems.
- Again like some actor languages, UFO provides conditional message acceptance. An object may accept certain messages only when particular conditions (defined in terms of the values of the instance variables) are met. Amongst other things, this allows non-strict or lazy data structures to be programmed if required.
- Unlike most actor languages, e.g. ABCL [Yone90], HAL[HoAg92], UFO has a static type system which provides type safety but with considerable flexibility. Overloading, subtyping via inheritance, and genericity are all supported.
- In addition, the type system provides important static protection from some of the problems associated with the transition from a pure functional environment. In particular, stateful objects can always be identified statically, and there is a static guarantee that a function cannot modify its arguments, even if they are stateful.[1]

3 Dataflow or actors?

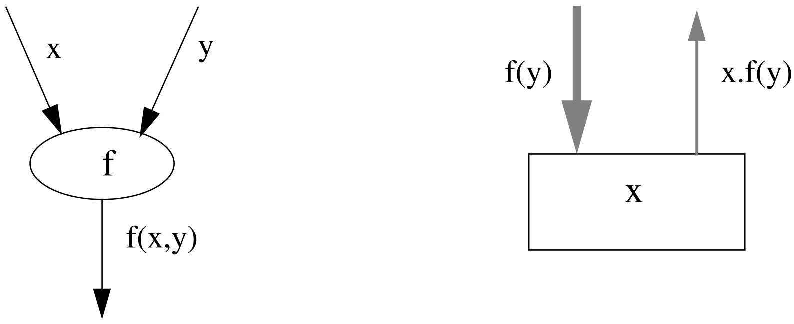
The UFO language design, and the motivation for that design, strongly suggest that aspects of both models are required. Dataflow is the obvious way to represent the (strict) functional computation which is the natural way to write many applications (especially numerical ones) in UFO. On the other hand, UFO provides the facilities of a concurrent object-oriented

1. The guarantee is a structural one, not merely a parameter-passing convention.

language, and the natural way to view such computation is in the actor style. The duality is illustrated by the syntax of a method call, where the functional style f(x,y) and the OO-style x.f(y) (in OO-speak: send message f(y) to the object x) are interchangeable.

In the dataflow model, data, as tokens on arcs, flows to methods at nodes. In the actor model, messages containing methods flow to nodes containing objects. So the models are in some sense inverses of each other. The other significant difference is that, in the simplest versions at least, dataflow describes purely functional computation, while actors have state.

.



Dataflow and actor models

There are essentially two possible approaches:

1. Have a single unified view of the computation that allows both aspects to be represented in the same graph structure.
2. Have two separate views, with a clear and simple relationship between them.

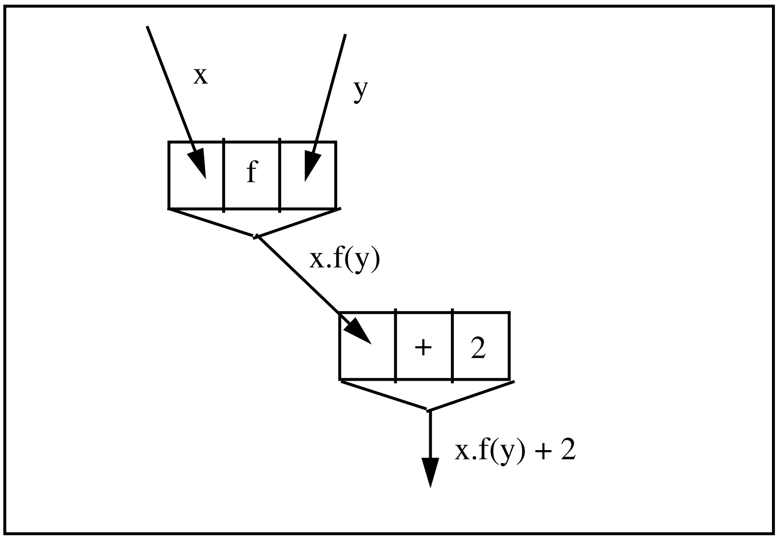
We initially investigated the unified approach. It turns out that this can be done, but the resulting model is rather complex and unsatisfactory in a number of ways. There are several different sorts of nodes and arcs, and the same piece of code can be represented in multiple ways.

We therefore concluded that it was better to have two separate views, the *computational view* and the *object view*. The latter is a straightforward representation of (possibly stateful) objects, and it is mostly the computational view which is of interest here.[1]

The computational view turns out to be a remarkably conventional dataflow model, with strong similarities to IF1 [SkWe85]. Arcs carry values which are primitive data values, references to objects, or references to methods. A node consists of a number of slots, each of which either has an input arc or holds a constant. The first two slots are distinguished. The first slot, when filled, contains (a reference to) the object, i.e. the first (primary) argument of the method. The second slot, when filled, contains the method. The other slots, if any, are for further arguments.

1. Of course a complete UFO toolkit will need other views, e.g. the static inheritance hierarchy, and perhaps more specialised visualisation aids, such as statecharts.

e.g.:



Note that this ordering reflects an object-oriented view of the world: x.f(y) is preferred. In OO terms, the contents of slots 2 .. n form a message to the object referenced by slot 1.

Normally, a node is enabled when all its slots are full. The only exception to this is when the method call blocks on access to a stateful object (see later). A node, once enabled, will be fired, but not necessarily immediately.

The form of the nodes may suggest a similarity to packet based graph reduction, and therefore inefficiency. However, in this case the process is all data driven, and we do not attempt to (directly) implement non-strict semantics.

4 Method call and strictness

In any dataflow model, a key issue is how to model reentrancy. There are two common approaches:

- Tagged tokens
- Graph copying

Tagged tokens are most appropriate when parts of a graph can start executing before other inputs are available. However, this is inefficient in space, which is particularly problematic for visualisation purposes, and can be a problem in a real implementation, exascerbating the problem of parallelism control or ‘‘throttling’’ [RuSa87]. We prefer a copying model where a method call happens at an identifiable moment in time, after all the inputs are available, at which point the graph is expanded, and input tokens are placed on the appropriate arcs. We are therefore rejecting some potential parallelism, but our experience is that this parallelism is usually spurious. The implementor can, of course, use lenient evaluation provided the semantics are preserved. Because the method may be dynamically bound, the graph produced may depend on the first element of the node (the object) as well as the second (the method). However, this is the only way in which dynamic binding affects the model.

Once the graph has been generated, the node has done its job. The node is conceptually replaced by a *container* holding the graph. The container has no computational role, it merely contains the nodes. For visualisation purposes, the container remains in place until all activity within it has finished.

We can now summarise some important aspects of the model:

- Data driven: Data flows to nodes which become enabled when all the necessary inputs are present.
- Strictness: all arguments to a node are evaluated before the node is enabled. This may seem restrictive, but it makes the other desirable properties possible.
- Maximum parallelism: Enabled nodes may be fired in any order, but all must be fired eventually.
- No suspended computation. Nodes become enabled, fire, and always terminate. They never suspend or block.[1]

5 Compound nodes

The next problem is how to represent other control structures such as conditionals and loops. Our approach here owes a great deal to IF1 [SkWe85]. Such structures are represented as special compound nodes, which perform some special-purpose action when fired. The problem with this is that the semantics of compound nodes becomes tied to that of a particular language: SISAL in the case of IF1. Although we are mainly concerned with implementing UFO, we would like the model to be more general; hence we would like to provide a general (object-oriented) method of description for compound nodes. However, there is a tradeoff here between generality, simplicity, and efficiency. Our current definition is something of a compromise, and we expect that things may change in this area in the future.

5.1 Conditionals

Conditionals are straightforward; there is a condition graph and two or more alternative graphs. The strictness principle applies, so the node is not enabled until all the inputs are available. The selected alternative will therefore be enabled as soon as the condition is evaluated. Although it would be possible to allow the condition to be evaluated early if the inputs it needed were available, we believe that the extra parallelism would not justify the extra complexity. Of course an implementor could choose to do this providing the semantics were preserved.

5.2 Loops

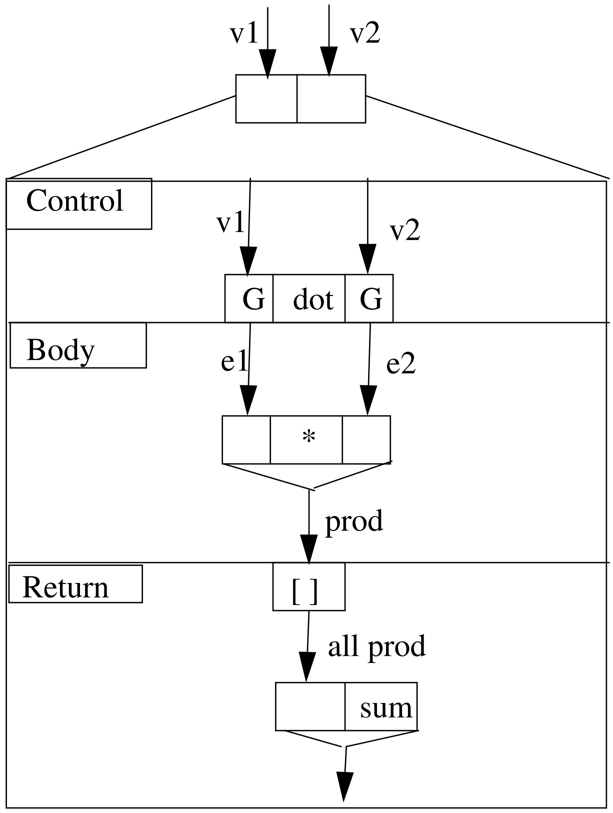
Loops in UFO consist of an initialisation part which gives initial values for single-assignment loop values, loop control, a body, and a return graph, which may take values from the last loop cycle, or collect values from all the cycles into an array, or both. The loop control may consist of a `while` condition or a generator which generates one loop cycle for each element of some array. This leads to the following cases:

1. Deadlock appears as nodes not enabled, and nontermination as never-ending creation of new nodes.

5.2.1 Simple generated loop

```
innerproduct( v1, v2: Vector ): Float is
for e1 in v1 dot e2 in v2 do
   prod = e1 * e2
return
   sum (all prod)
od
```

return all values of prod as an array and sum them.

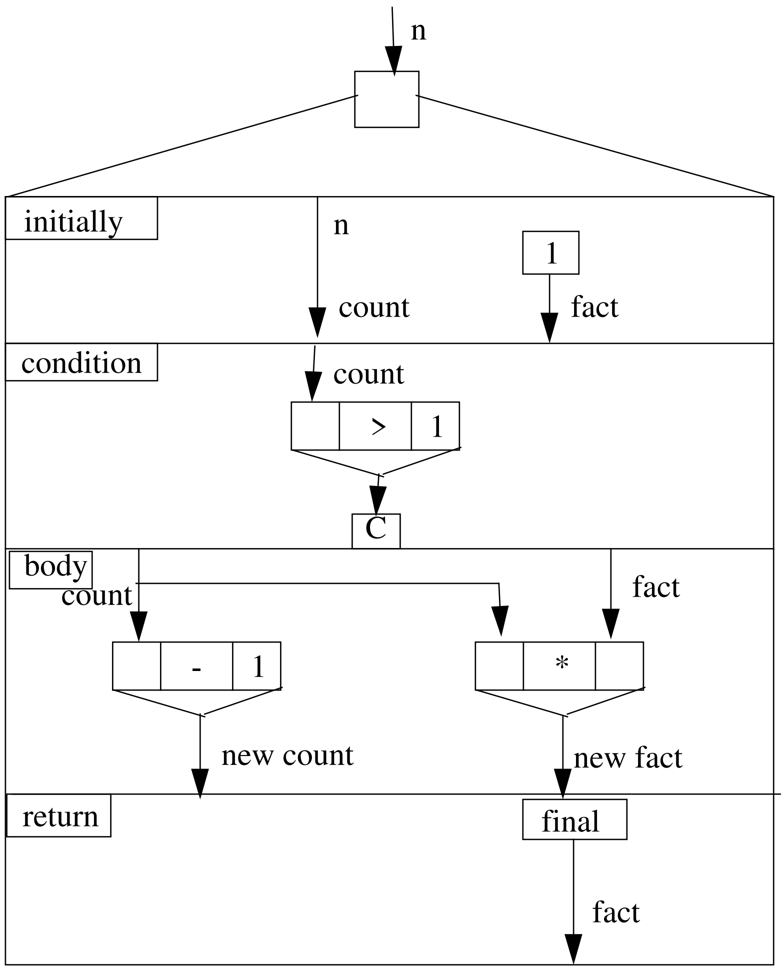


The slots at the top ‘‘recieve’’ the incoming values; when they have all arrived things can start happening. In this case, there are no (explicit) dependencies between the loop cycles. All the loop bodies are enabled as soon as the generator is evaluated, and so all can execute in parallel. Of course if there are a large number of cycles, a scheduling policy is required. This is discussed in section 9.1. The other points of interest are the interfaces between the parts of the loop. Between the control part and the body, the values of e1 and e2 for each instance of the body are generated. Between the body and the return part, the values of prod are converted into an array (all prod), one element for each instance of the body. The result array will have the same size and shape as the generating arrays.

The picture above shows the whole structure of the graph; this is what will be represented inside the compiler. The user of a visualisation tool will see something a little different, because at any particular time, any number of instances of the body (including none) may exist.

5.2.2 While loop

```
factorial( n: Int ): Int is
 initially
   count = n; fact = 1
while count > 1 do
   new count = count - 1;
   new fact = fact * count
return
  fact                    Final value of fact
od
```



In this case, there are data dependencies between the loop cycles, and the number of loop cycles is, of course, not known in advance. Each instance of the loop body is created when the condition for its cycle evaluates to true, and enabled when all its inputs (including the loop values from the previous cycle) are present. This does not necessarily mean that the loop executes sequentially (although the one in the example will). If the condition, and the loop values, can be evaluated quickly, then many instances of the body may be evaluated concurrently. However, the strictness ensures that a loop cycle, once started, has all the values it needs to run to completion.[1] This avoids the problem which dogged early dataflow

1. It may still not do so, because of blocking on stateful objects, discussed later.

implementations of tokens corresponding to many loop cycles clogging up the store, as their partners would not arrive until much later, an aspect of the throttling problem mentioned earlier. The graph also shows another case of body-return interface, where only the final value is returned.

Note that we do not show all the connections between the parts of the graph; there is some ‘‘implicit wiring’’ which is omitted for clarity.

5.2.3 Generated loop with loop values

```
** Replace multiple occurrences of whitespace by single ones.
squash_whitespace( text: Array[Char] ): Array[Char] is
initially
   last_char_white = false
for ch in text do
   this_char_white = ch.is_whitespace;
   ok = not (this_char_white and last_char_white);
   new last_char_white = this_char_white
return
   all ch when ok ** Filter according to the loop value ok.
od
```

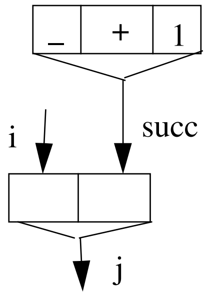
This is a hybrid of the previous two cases, where the number of loop cycles is determined by the generator, but data dependencies restrict their enablement. It also shows the other return option, which is an array but filtered by the value of some other (boolean) loop value (e.g. `all ch when ok`). In this case the result array is 1D. Values which cross the interface between the body and returns part therefore go through one of these three conversions. The return part itself is an arbitrary expression, since the loop values have already been converted. Reduction operators (such as `sum` in the first example) are simply general purpose functions on arrays, not part of the loop infrastructure.[1] The graph of this example is straightforward.

6 Higher-order functions

Function values may be represented simply by having references to functions passed on arcs rather than fixed at nodes. Partial application is represented using a node which has dummy slots which do not need to be filled before it fires. The result is a partial application packet, which can then be passed to other nodes, e.g.

1. A serious implementation will know about these functions, however, and will optimise accordingly.

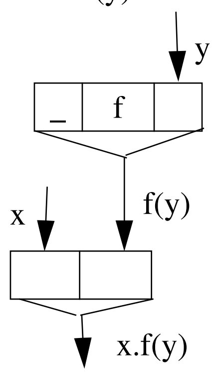
```
succ = Int| + 1
j = i.succ
```



The expression `Int| + 1` is UFO syntax for a partially applied operator; the type name `Int|` is used to determine which + is required (since this is normally inferred from the argument values).

Note that the partial application packet is inserted in a single slot in the node in the usual way; its slots are not “matched” against the slots in the node. Such matching is an attractive idea, but it does not work because a node representing a function application cannot in general “know” whether the incoming function has already been partially applied.

The partial application packet can also be used to provide a more “object-oriented” view of ordinary function application, by constructing the message at one node and then sending it to the node where it meets the object, e.g. for x.f(y)



7 Parallelism measurement and atomicity

The model as defined thus far has two useful properties which we are about to destroy in section 8. They are still worth highlighting, as many applications, or large sections of applications, can be sensibly written in the functional subset of UFO which the simple model describes.

7.1 Parallelism measurement

It is possible to measure the “inherent” parallelism of any program / data set, independently of the target architecture, and independently of the actual evaluation order. This is an old dataflow trick which is well worth reviving. Suppose we assign to each primitive operation (e.g. arithmetic, graph expansion) a cost in terms of arbitrary time units. (For graph expansion the cost may depend on the size of the graph). Primitive operations are assumed to execute sequentially. Then we run the program using an interpreter or instrumented code, and calculate 3 numbers:

- W, the total amount of work done, in terms of the time units (i.e. the time it would take to run the program sequentially).
- L, the length of the longest sequential execution path through the program, i.e. the time taken for an “ideal” parallel execution, where processors are always available and ignoring all overheads.
- Pi = W/L, the average parallelism of the program.

W[1] is obviously easy to determine. L is calculated as follows. Each value is marked with a generation tag, which is the earliest time it could be created. The inputs to the program have generation 0, and L is the largest generation created. At a node executing an atomic operation (such as arithmetic) the generation of the result is the largest generation of any of the inputs, plus the cost of the operation. At a node which causes graph expansion, such as a function call, the inputs have their tags set to that of the latest input plus the graph expansion cost, and the process continues in the generated graph.

Although this is a very abstract measure of parallelism, it is still not “absolute”. For instance if the algorithm could benefit from pipeline parallelism through non-strictness, this will not be reflected in Pi.

7.2 Atomicity

The discussion so far has assumed that graphs are expanded down to atomic nodes. In fact, in any program which terminates and does not crash, we can regard *any* node as atomic in the sense that we can fire it and it will eventually terminate and produce its result, independently of all other nodes. This is important for visualisation purposes, as it means that the graph can be viewed at any level of granularity, and only those nodes which are of interest need be expanded. At the implementation level, it means that any node may be executed sequentially. From the point of view of measuring parallelism, it allows experimentation with the effects of granularity on parallelism, if we assume that nodes deemed atomic are executed sequentially.

8 Stateful objects and non-strictness

As mentioned in section 2, stateful objects have mutable state in the form of instance variables. Internal coherence is ensured by allowing a procedure to define exactly one new value for each variable which it updates; the old and new values are syntactically distinguished, like old and new loop values in single-assignment loops. Interference between method calls is avoided by locking an object while a procedure is updating it. Messages queue for access, by default FIFO, but conditional message acceptance can be used to override this; when the object is unlocked, the first *acceptable* message in the queue is processed next.

Although here we show stateful objects used only in implementing non-strict computation, we expect that in general they will be used in many application areas. In particular, they may often provide a more natural way of modelling real-world objects than a pure functional representation. Real world objects have identity and state. They retain their identity when their state changes[2], a property which a pure functional representation does not address.

1. Historically, W and L have been called S1 and Sinfinity.
2. This is fundamental to Object-Oriented Design, as expounded in e.g. [Booch 94], although the OOD perspective perhaps exaggerates the statefulness of everything.

A simple implementation of non-strictness is enough to explain the model and highlight the issues, however. The following class implements a straightforward I-Structure location. An I structure [ArTh81] is one each of whose elements is written exactly once; an attempt may be made to read an element before it has been written, in which case the read is deferred until the write has been made. A presence bit is associated with each location in order to detect whether or not the write has occurred.

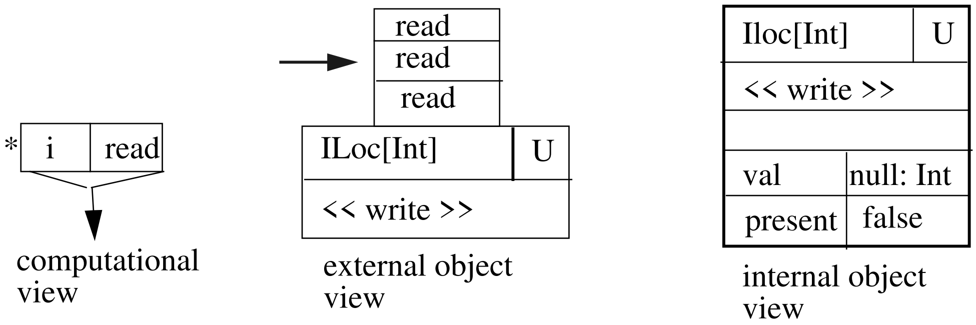
```
stateful class ILoc[T]
** Give initial values for the instance variables
{ initial val = null:T;
  initial present = false }

accept read when present

** read is a function
read: T is val

** write is a procedure - note similarity to loops.
proc write( v: T ): Void is
 pre
  not present
do
  new val = v;
  new present = true
od

end ** ILoc
```

The use of an array of such objects to implement a wavefront algorithm, and generalisation to more powerful non-strict structures such as futures, are described in [Sarg93a].

The implementation relies on conditional message acceptance: reads are only accepted after the value is written. In a high performance implementation, this would be a built-in library class, implemented as efficiently as possible, with hardware support if available. We note in passing that this example neatly illustrates the difference between acceptance conditions and preconditions. If an acceptance condition is not met (`read` when not `present`), the method simply is not accepted yet. In the computational view, this appears as a node which, although all the inputs are present, is not yet enabled. If a precondition is not met, (`write` when `present`) an error occurs. Also, it seems to us that the acceptance condition is a property of the *object*: it decides what messages it is currently prepared to accept. The precondition, on the other hand, seems more naturally a property of the *method*: it decides whether things are in the right state when it gets there. We would therefore argue against merging acceptance conditions and preconditions, as proposed in [Meye93].

Representation of the methods is straightforward. In particular, the body of procedure `write` can look *exactly* like the body of a loop with loop values `val` and `present`. The single assignment updates mean that parallelism can be exploited within the body of a procedure just like anywhere else.

For visualising stateful objects, the object view becomes more important. More precisely, encapsulation requires two versions of this view: one of the outside of an object (the state of its message queue, and any public instance values) and one of the inside (the instance variables and private instance values). For instance, an I-Structure with a blocked read might appear as follows:



In the computational view, we see a node with all inputs present, but not enabled (indicated by *). In the outside object view, we see our read request in the queue, along with two others (the user could follow these back into the computational view). We also see that the object is of type ILoc[Int], is unlocked (U), but is only accepting writes. In the inside computational view, we see the values of the instance variables, and in particular that the presence bit is not set. The empty slot would refer to the method currently updating the object if there were one (and hence would be another link back to the computational view).Typically, a user of the class would not need to see (perhaps even not be allowed to see) the inside view, but the designer of the class would. The object views would also allow the user to follow references between objects.

What are the consequences of introducing state? The maximum parallelism principle still applies: it is still ok to fire enabled nodes in any order provided all are fired eventually. Blocking on stateful objects (either because of mutual exclusion or because of CMA) shows up in the computational view as nodes which are not enabled even though they have all their inputs. There is still no notion of suspended computation. Deadlock therefore appears as a situation where no nodes are enabled, and can be traced by investigating which nodes are blocked and why (by flipping between the views).

However, there are several difficulties. The parallelism in the program is no longer independent of evaluation order: indeed some orders may lead to nontermination or deadlock while others do not. The atomicity property no longer holds in general, as there can be implicit dependencies between nodes which may require one node to fire before another can terminate. In general, programs are much harder to visualise and reason about. Implementation is also more difficult, as described in the next section.

Nevertheless, we believe that stateful objects are necessary for some applications, and that these problems are inherent, and not an artifact of our model. The UFO type system allows us to distinguish statically which objects are stateful. The model provides a natural representation both of the overall computation and of the behaviour of the individual objects.

9 Implementation issues

9.1 Scheduling

The scheduling mechanism must guarantee that all enabled nodes get fired eventually, but otherwise it is not constrained. For visualisation purposes, a user might well decide by clicking a mouse which node to fire next. However, when not user-guided, a scheduling strategy is necessary. The main consideration is limiting the number of nodes in existence at any one time, to limit store usage (and screen usage in the visualisation case). The basic strategy therefore is:

1. Go breadth-first until enough parallelism is generated, depth-first thereafter.
2. For loops, do earlier cycles first.

This is straightforward to implement in the pure functional case, but blocking on stateful objects may result in there being no enabled nodes on the "ideal" path.

9.2 Efficiency

Obviously the model maps very directly onto modern dataflow architectures. For instance, the graph expansion on firing a node is equivalent to allocating an operand segment in the EM4 machine [Saka89].[1]

For more conventional architectures, the code must be transformed to a threads-based model. A suitable model, incorporating variable granularity, lazy task creation, and load balancing via active messages is described in [Wats94]. The main problem with such models is making the threads large enough, as conventional machines tend to have very long (relative to our requirements, at least) context switching times. In the pure functional case, the implementation can use whatever thread granularities it likes (even a single serial thread), and the semantics are preserved. In the presence of stateful objects, this is no longer the case, as there are implicit dependencies between nodes, and using large threads may well cause spurious deadlock.

A naive correct implementation is to generate a separate thread for each node representing a method call on a stateful object.[2] The game then is to optimise this by detecting as many cases as possible where the separate thread is unnecessary, and thus merging threads. The type system provides a certain amount of useful information in this regard. An M.Sc. project is currently investigating this [Hoot94].

Of course there are many other issues to address, for instance load balancing and data distribution. For simple applications involving arrays and loops, we expect to be able to apply existing technology such as that using in parallelising Fortran compilers, and particularly that used in the SISAL project [Cann 92,HaBo92]. For more irregular computations, we will draw on our own experience with parallel functional implementations [WaSa88,SaWa91,Sarg91] and that from the concurrent OOP community.

10 Textual format definition

10.1 Introduction

There are in fact three formats, which have the same lexical conventions, and the same representations for everything except expressions (i.e. class definitions, method headers, typespecs etc. are the same for all three). The variations in the way expressions are represented reflect the purposes of the formats:

1. In practice, of course, it's not quite as simple as this. Things like operand segments often have fixed sizes, requiring the compiler to partition large graphs.
2. Note that we associate threads with nodes in the computational graph (method calls), not with nodes in the object graph (objects).

1. UFO Intermediate Format (.uif). This is the output from the parser, the input to the main part of the current compiler. Expressions are represented in the simplest possible way, i.e. in syntax tree form. This format may become obsolete once we have an integrated parser.
2. UFO Intermediate Format Interface (.uifif). This is used to represent interface information only, for separate compilation, and therefore expressions are not represented at all. uifif is essentially the subset of uif where all method bodies are deferred.
3. Uflow textual format (.uflow). This is the representation which will be used as the input to the visualisation tool, and for optimisation and parallel code generation. An expression is represented as lists of nodes forming an acyclic dataflow graph.

The use of a textual intermediate format allows different tools and optimisations to be worked on independently. It is likely that, at some point in the future, the uflow data structures and many of the optimisations will be added into the compiler, to eliminate the overhead of generating and reading in the intermediate files. However, the textual format will still have a role to play in allowing optimisations to be added incrementally.

10.2 Common formats

10.2.1 Lexical conventions

Consecutive tokens must always be separated by whitespace. This is because the tokeniser is dumb, apart from handling whitespace in character and string constants as special cases.

Constants are lexically distinguished:

`constant ::=`	`digit+`	Integer
	`│ digit+ '.' digit+ [ 'E' ['+' │ '-'] digit+ ]`	Float
	`│ ''' char '''`	Char
	`│ '"' char* '"'`	String

Characters may be escaped with \. Single and double quotes in characters and strings are represented by \' and \" respectively.

Identifiers are the usual thing:

```
identifier ::= letter (letter | digit)*
```

where ''letter'' includes underscore. Identifiers and operators may be qualified by typenames. Disambiguated global method names are used in the uflow format

```
qualified_id ::= identifier '|' identifier     Note no whitespace
qualified_op ::= identifier '|' op '|' identifier
global_method_id ::= qualified_id | qualified_op
general_id ::= identifier | global_method_id
```

The operators are listed later.

Names may come with or without line numbers attached:

```
linenum ::= digit+
name ::= identifier [whitespace linenum]
```

The current convention is that the uif contains line numbers (because the compiler uses them to generate error messages) but the uifif and uflow do not.

And the operators are:

```
op ::=      `++' | `and' | `or' | `#' | `::'
     |  `==' | `!=' | `<` | `>' | `<=' | `>='
     | `+' | `-' | `*' | `/' | `div' | `mod' | `not' | `.' | `!'
```

10.2.2 Overall structure

Lists are always delimited by parentheses. To reflect this, in the BNF below, `thing*`, `thing+` and `thing++` are taken to mean

`` `(` thing zero or more times `)' ``
`` `(` thing one  or more times `)' `` and
`` `(` thing two  or more times `)' ``
respectively. A program or a separately compiled piece of program is a list of top level definitions:

```
program ::= topleveldef+
topleveldef ::= classdef | methoddef
                | typedef | typevardef | enumdef
```

10.2.3 Class definitions

```
classdef ::= `C' status [stateflag] classname genname*
              classformal* supername initval* methoddef*
```

`` status ::=  `o' ``	Open - the default
``	`d' ``	Deferred
``	`c' ``	Closed
``	`b' ``	Built-in

`` stateflag ::= `!' ``	Indicates class is stateful if present
`classname ::= name`	The name of the class
`genname ::= name`	Generic parameter name
`supername ::= name`	Superclass
`initval ::= valdef`	Initialised value or variable

Valdefs differ in the different formats. Class formal parameters have a similar format to formal parameters to methods (below), but may include a visibility:

```
classformal ::= [visibility] name+ typespec
```

`` visibility ::=   `~` ``	Public
``	`?' ``	Private

The default visibility for classformals is public. Note that there is a list of names for each typespec.

10.2.4 Method definitions

```
methoddef ::= [visibility] methodsort methoddec methodbody

methodsort ::=  'F'                                  Function
              | 'O'                                  Operator
              | 'P'                                  Proc

methoddec ::= name formal* typespec                  method name, formals, result type
formal ::= name+ typespec
```

Again, a parameter list has a list of names for each typespec. Method bodies differ in the different formats.

10.3 Other top level definitions

```
typedef ::= 'T' name name+ typespec                  Type name, params, named type.
typevardef ::= 'V' name
enumdef ::= 'E' name name+                           Enumeration name, value names.
```

Note: Enumerations will probably disappear in a future version of the language.

10.3.1 TypeSpecs

```
typespec ::=
      'I'                                  Int
    | 'S'                                  String
    | 'B'                                  Bool
    | 'R'                                  Float
    | 'C'                                  Char
    | 'X'                                  Bits
    | 'N' name typespec*                   Named type, actual generics.
    | 'A' typespec                         Array, element type
    | 'F' typespec* typespec               Function, arg types, result type
    | 'T' typespec++                       Tuple, element types
    | 'V'                                  Void
    | 's' name typespec                    Synonym, target type
    | 'c' name typespec*                   Class type, actual genererics
    | 'e' name                             Enumeration
    | 'g' name                             Generic variable
```

Types s, c, e and g do not appear in the uif, as the parser cannot distinguish them; they all appear as named types. The compiler constructs a type namelist and resolves them. They are distinguished in the uflow.

10.4 Syntax tree format for expressions

This format is used for expressions in the uif , not in the uflow. Value definitions (valdefs) may have multiple left hand sides if the RHS expression yields a tuple.

```
valdef ::= name+ exp
```

A ‘‘statement’’ which is not a valdef (e.g. a Void proc call) appears as a valdef with a dummy lhs name --.

Currently, tuples are transformed out on input (to instance of the standard classes Tuple1, Tuple2...), so they do not appear in the uflow format below, and valdefs in the uflow have single LHSs.

In general, ‘‘leaf’’ expressions such as identifiers and constants have line numbers attached, so that the compiler can give a reasonable line number for any expression by finding the nearest leaf.

```
exp ::=    'I' name                          Identifier
      | 'O' op exp exp                       Binary operator
      | 'M' op exp                           Unary operator
      | 'C' typespec constant linenum        Constant, with line num.
      | 'B' blockexp
      | 'i' ifexp
      | 'e' caseexp
      | 'F' callexp                          Function call f(x,y)
      | 'D' callexp                          Dotcall x.f(y)
      | 'P' callexp                          Proc call x!p(y)
      | 'c' typespec exp                     "cast", e.g. Float(42)
      | 'S' linenum                          self
      | 'V' linenum                          void
      | 'E' typespec linenum                 Error: T
      | 'N' typespec linenum                 null: T
      | 'A' arrayexp
      | 'L' loopexp
      | 'S' exp slice                        a[i], or more general slice
      | 'T' exp++                            Tuple, 1-tuples not allowed.
      | 'G' name                             Gather i.e. all x
      | '_'                                  Missing expression for
                                             partial application.
```

A block is simply a sequence of valdefs and a return expression (which is void if the block returns no result).

```
blockexp ::= valdef* exp
```

An ifexp simply consists of its three subexpressions. A caseexp consists of an expression (the of part), the arms, and the otherwise expression.

```
ifexp ::= exp exp exp
caseexp ::= exp casearm+ exp
casearm ::= exp+ exp                         LHSs, RHS.
```

Calls are the obvious thing:

```
callexp ::= name exp*                        Method name, actual args
```

Method names are not disambiguated in the uif - this is the job of the compiler.

Arrays can be created by listing their elements, or as arithmetic progressions. Currently, the only form of slice allowed is simple indexing.

```
arrayexp ::=      'L' exp exp+               lowerbound, elements.
       | 'A' exp exp exp exp    Lowerbound, from, to, step
       | 'E' typespec linenum   Empty array, typespec is compulsory.

slice ::= indexexp
indexexp ::= exp
```

Loops consist of an (optional) set of initial values, a loop control part, a body and a return expression. while loops and for loops have the same general form and differ only in the loop control part. Currently, only single generators are implemented.

```
loopexp ::=   ['I' valdef*] loopcontrol body expression
loopcontrol ::=   'W' expression   | 'F' generator
body ::= valdef*
generator ::= 'O' name expression             for x in e
```

10.5 Uflow node format

In ufllow, a (non-deferred) method body is represented as a list of nodes, as are expressions in general.

```
 body ::=   'd'    Deferred
       | node+
```

At the moment, nullcase/otherwise bodies are not explicitly represented: they are transformed to the equivalent if expressions. This may be reviewed later.

Each node within a graph has a distinct number; the output of the (textually) last node in the graph is the output of the graph. Currently, the nodes are in fact numbered consecutively, but to avoid over-specification, we do not require this. A graph normally contains at least one node, so that its output can always be identified.

```
node ::= node_number node_body node_result
node_body ::= simple_node | compound_node
node_result ::= typespec [ name ]
```

The result type of the node is given, and if the node result corresponds to a name in the UFO source, this name is appended.

10.5.1 Simple nodes

A simple node is simply a sequence of (at least 2) slots

```
simple_node ::= 'S' slot++
```

The first slot corresponds to the first argument of the method (‘‘the object’’), and the second to the method. There are four types of slot:

```
slot ::= empty_slot | input_slot | param_slot | const_slot
empty_slot ::= '_'
input_slot ::= 'N' node_number
```

```
param_slot ::= 'P' identifier
const_slot ::= 'C' uflow_constant
uflow_constant ::= constant | global_method_id
```

An empty slot corresponds to a missing argument in a partial application, i.e. it does not need to be filled for the node to be enabled. An input slot represents input from another node in the same graph. A parameter slot represents input from a parameter to the graph (e.g. a parameter to the enclosing function). The parameter is identified merely by its name. A constant in a constant slot is either a constant value, as above, or a global method id.

The method name may be provided as a constant (the method is a global one: the most common case), as a parameter (the method is provided as a parameter to the enclosing graph) or an input (the method is constructed locally as a partial application). Method names are provided in disambiguated form, e.g.

```
Complex|square
Int|+|Int
```

In execution terms, a node is enabled when values are provided in all its input slots, and the object can accept the method. Other types of slots do not delay enabling, since constant values are already present, parameter values are available when the graph containing the node is expanded (strictness principle), and empty slots are not supposed to be filled.

10.5.2 Compound nodes

Currently, only two compound nodes are specified, representing `if` expressions and loops. It is intended that the format should be extensible in this area. A likely extension is to cover `case` and/or `typecase` expressions. The current simple `case` expression is implemented by transforming to a nested `if` expression. `elseif` clauses are also transformed to nesting. Obviously for efficiency reasons we will want a proper multi-way branch at some stage.

```
compound_node ::= if_node | loop_node
```

Note that there is no compound node corresponding to a block: a simple sequence of nodes is sufficient. The inside and outside of a compound node are ''wired'' together via names, which are accessed as parameter slots inside the compound. This results from the observation that the only way that an outside value can be accessed from within a compound is via a name. A compound node therefore has a ''header'' consisting of a sequence of slots as usual except that every slot is associated with a name:

```
named_slot ::= param_slot | input_slot_with_name
input_slot_with_name ::= input_slot identifier
```

As usual, the node is enabled when all the input slots are filled. Within the compound, all the input values are referred to by name, as parameter slots.

```
if_node ::= 'I' named_slot* node+ node+ node+
```

Apart from the header, an `if` node simply consists of graphs for the condition and the then and else expressions. The output of the last node of the condition graph is the value of the condition. The output of the whole compound comes from the last node of whichever alternative is selected.

```
loop_node ::= 'L' named_slot* initials control body
               return_interface loopreturn
initials ::= node*
control ::= node+
body ::= node*
return_interface ::= node+
loopreturn ::= node*
```

Loops are quite complex, and the definition here is provisional. The main ideas are as follows:

As many components as possible are treated as ordinary graphs. This is true for the initial, body, and loopreturn parts. In the latter case this is achived by separating out the return interface, which handles the collecting of values from the body instantiations. Although there are several separate graphs, they share a ''node number space'', i.e. the node numbering is contiguous through all the graphs. This means that when appropriate a loop can be treated as a single graph, whose output is the output of the last node overall. The may be either in the loopreturn graph or the returninterface graph. The ''wiring'' between the different graphs is largely implicit in the names of the loop variables etc. New loop variables are distinguished from old ones by having `new` prepended to their names. There are no ''magic numbers'' defining the inputs and outputs of each subgraph. The application using the format is responsible for constructing the wiring.

Loop values occupy an iteration space, whose size (and shape in the multi-D case) is determined by the nodes in the control part, which form either a `while` condition or a `for` loop generator. For a `while` condition the iteration space is 1D, with a lowerbound of 0 for loop values initialised in the initials part, and 1 for the others. The upperbound corrsponds to the iteration where the condition becomes false.

In the case of a `for` loop the iteration space is determined by special nodes which generate loop cycles from the input arrays. Currently, only a single generator (the ascatter node below) is implemented. This creates a 1D iteration space with the same bounds of the generating array, except for initialised loop values which have a lowerbound 1 less than that of the array.

The return interface consists entirely of special nodes which collect iteration spaces of loop values, returning all of them as an array, or just the last value.

10.5.3 Special functions

Note that most builtin functions and operators are simply referred to by their (disambiguated) names, e.g.

```
Int|*|Float
Int|print
```

Special functions are limited to those cases which correspond to syntactic constructs in the language, connected with arrays and loops, and a few special cases such as no-ops.

In the table below, T is an arbitrary type variable, and IS[T] is the type of an iteration space of Ts.

Table 1:

name	input types	output types	description
index	Array[T], Int	T	Array indexing A[I]
abuild	Int, T, T, ... T	Array[T]	Build an array from a collection of values. First input is lwb.
aparray	Int,Int,Int,Int	Array[Int]	Build an array as an arithmetic progression inputs are lwb, from, to, step
ascatter	Array[T]	IS[T]	Generate the iteration space corresponding to the array
agather	IS[T]	Array[T]	Yield the array corresponding to the iteration space
final	IS[T]	T	Yield the final value in the iteration space
noop	T	T	Pass input to output
const	T	T	Trigger a constant (synonym for noop).

Note that the abuild node is unique in having a variable number of inputs. When only a lowerbound is present, the array is empty. It is expected that further special functions will be added to deal with multiple generators.

11 Examples

11.1 Hello world

This example shows the basic form of all formats, and also how strings are represented.

hello.ufo

```
main: String is "Hello world"
```

hello.uif

```
( F main 1 ( ) S C S "Hello world" 2 )
```

hello.uifif

```
( F main ( ) S d  )
```

hello.uflow

```
F |main( ) S
( 1 S (  C "Hello world" C const ) S )
```

11.2 Nfib

Another simple example of all formats, showing an example of a conditional.

nfib.ufo

```
nfib( n: Int) : Int is
  if n < 2 then 1 else 1 + nfib(n-1) + nfib(n-2) fi

main: Int is nfib(20)
```

nfib.uif

```
( F nfib 1 ( ( n 1 ) I ) I i O <  F  n 2 ( ) C I 2 2 C I 1 2 O
+  O +  C I 1 2 F  nfib 2 ( O -  F  n 2 ( ) C I 1 2 ) F  nfib 2
( O -  F  n 2 ( ) C I 2 2 )
F main 4 ( ) I F  nfib 4 ( C I 20 4 ) )
```

(The parameter n is represented as `F n` rather than `I n` because the parser cannot distinguish between local names and parameterless function calls).

nfib.uifif

```
( F nfib ( ( n ) I  ) I d  F main ( ) I d  )
```

nfib.uflow

```
F Int|nfib ( ( n ) I  ) I
(
1 I (  P n  )
     (
     1 S (  P n  C Int|<|Int C 2 ) B           Condition
     )
     (
     1 S (  C 1 C const ) I                    Then part
     )
     (
     1 S (  P n  C Int|-|Int C 1 ) I           Else part
     2 S (  N 1 C Int|nfib ) I
     3 S (  C 1 C Int|+|Int N 2 ) I
     4 S (  P n  C Int|-|Int C 2 ) I
     5 S (  N 4 C Int|nfib ) I
     6 S (  N 3 C Int|+|Int N 5 ) I
     ) I
)
```

```
F |main ( ) I
(
1 S (  C 20 C Int|nfib ) I
)
```

Note how the nfib function and the operators are disambiguated in the uflow.

The rest of the examples omit the uif and uifif, as they are fairly uninteresting.

11.3 Area

A larger example featuring a block of value definitions, and a higher-order function call.

area.ufo

```
type RtoR =  Float -> Float;

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

area(l,r,est,tol: Float; f: RtoR ): Float is
{
mid = (l + r)/2.0;
a1 = trap(l, mid, f); a2 = trap(mid, r, f);
newest = a1 + a2 r
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
}

g( x: Float): Float is 3.0*x*x + 2.0*x + 1.0

main : Float is area( 0.0, 10.0, 0.0, 0.001, Float|g )
```

area.uflow

```
T RtoR ( ) F ( R  ) R

F Float|trap ( ( l ) R  ( r ) R  ( f ) N RtoR ( )  ) R
(
1 S (  P r  C Float|-|Float P l  ) R
2 S (  P l  P f  ) R
3 S (  P r  P f  ) R
4 S (  N 2 C Float|+|Float N 3 ) R
5 S (  N 1 C Float|*|Float N 4 ) R
6 S (  N 5 C Float|/|Float C 2.0 ) R
)
```

```
F Float|area ( ( l ) R  ( r ) R  ( est ) R  ( tol ) R  ( f ) N
RtoR ( )  ) R
(
1 S (  P l  C Float|+|Float P r  ) R
2 S (  N 1 C Float|/|Float C 2.0 ) R  mid
3 S (  P l  C Float|trap N 2 P f  ) R  a1
4 S (  N 2 C Float|trap P r  P f  ) R  a2
5 S (  N 3 C Float|+|Float N 4 ) R  newest
6 I (  N 3 a1  N 4 a2  P est   P f   P l   N 2 mid  N 5 newest
P r   P tol  )
     (
     1 S (  P est  C Float|-|Float P newest  ) R
     2 S (  N 1 C Float|abs ) R
     3 S (  N 2 C Float|<|Float P tol  ) B
     )
     (
     1 S (  P newest  C noop ) R
     )
     (
     1 S (  P tol  C Float|/|Float C 2.0 ) R
     2 S (  P l  C Float|area P mid  P a1  N 1 P f  ) R
     3 S (  P tol  C Float|/|Float C 2.0 ) R
     4 S (  P mid  C Float|area P r  P a2  N 3 P f  ) R
     5 S (  N 2 C Float|+|Float N 4 ) R
     ) R
)

F Float|g ( ( x ) R  ) R
(
1 S (  C 3.0 C Float|*|Float P x  ) R
2 S (  N 1 C Float|*|Float P x  ) R
3 S (  C 2.0 C Float|*|Float P x  ) R
4 S (  N 2 C Float|+|Float N 3 ) R
5 S (  N 4 C Float|+|Float C 1.0 ) R
)

F |main ( ) R
(
1 S (  _  C Float|g ) F  ( R  ) R
2 S (  C 0.0 C Float|area C 10.0 C 0.0 C 0.001 N 1 ) R
)
```

Note how nodes which give value names are followed by the defined name, e.g.

```
2 S (  N 1 C Float|/|Float C 2.0 ) R  mid
```

11.4 Complex

This example is mainly to show a simple class definition. It also features operator definitions.

complex.ufo

```
class Complex( public re, im : Float )

square: Complex is self * self

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

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

end ** Complex

main: Float is Complex( 1.5, 2.5 ).square.im
```

complex.uflow

```
F |main ( ) R
 (
1 S (  C 1.5 C Float|Complex C 2.5 ) N Complex ( )
2 S (  N 1 C Complex|square ) N Complex ( )
3 S (  N 2 C Complex|im ) R
)

C o Complex ( ) ( ( re ) R  ( im ) R  ) ANY  ( )

F |square ( ) N Complex ( )
(
1 S (  P self  C Complex|*|Complex P self  ) N Complex ( )
)

O Complex|* ( ( c ) N Complex ( )  ) N Complex ( )
(
1 S (  P self  C Complex|re ) R
2 S (  P c  C Complex|re ) R
3 S (  N 1 C Float|*|Float N 2 ) R
4 S (  P self  C Complex|im ) R
5 S (  P c  C Complex|im ) R
6 S (  N 4 C Float|*|Float N 5 ) R
7 S (  N 3 C Float|-|Float N 6 ) R
8 S (  P self  C Complex|re ) R
9 S (  P c  C Complex|im ) R
10 S (  N 8 C Float|*|Float N 9 ) R
11 S (  P self  C Complex|im ) R
```

```
12 S (  P c  C Complex|re ) R
13 S (  N 11 C Float|*|Float N 12 ) R
14 S (  N 10 C Float|+|Float N 13 ) R
15 S (  N 7 C Float|Complex N 14 ) N Complex ( )
)

O Float|* ( ( r ) R  ) N Complex ( )
(
1 S (  P self  C Complex|re ) R
2 S (  N 1 C Float|*|Float P r  ) R
3 S (  P self  C Complex|im ) R
4 S (  N 3 C Float|*|Float P r  ) R
5 S (  N 2 C Float|Complex N 4 ) N Complex ( )
)
```

11.5 A loop example

The following artificial example shows the format of loops.

looptest.ufo

```
typevar T, U
type TtoU = T -> U

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

factorial( n: Int ): Int is
 initially     fact = 1;  i = 1;
while i <= n do
    new fact = fact*i;
    new i = i + 1
return
    fact * 2
od

square( x: Int ): Int is x * x

main : Int is
{
a = [1,2,3,4,5];
b = a.map( Int|square )
return
    b[5] + factorial(5)
}
```

looptest.uflow

```
V T

V U
```

```
T TtoU ( ) F ( N T ( )  ) N U ( )

F Array|map ( ( a ) A N T  ( f ) N TtoU ( )  ) A N U
(
1 L (  P a   P f  )
     ( )                                       (Empty) initial part
     (
     1 S (  P a  C ascatter ) N T  x           Generator
     )
     (
     2 S (  P x  P f  ) N U  y                 Body
     )
     (
     3 S ( p y C agather ) A N U all_y         Return interface
     )
     ( )                                       (Empty) return graph
)

F Int|factorial ( ( n ) I  ) I
(
1 L (  P n  )
     (
     1 S (  C 1 C const ) I  fact              Initial part
     2 S (  C 1 C const ) I  i
     )
     (
     3 S (  N 2 C Int|<=|Int P n  ) B          Test
     )
     (
     4 S (  N 1 C Int|*|Int N 2 ) I new fact   Body
     5 S (  N 2 C Int|+|Int C 1 ) I  new I
     )
     (
     6 S ( p fact C final ) I fact             Return interface
     )
     (
     7 S (  N 6 C Int|*|Int C 2 ) I            Return graph
     )
)

F Int|square ( ( x ) I  ) I
(
1 S (  P x  C Int|*|Int P x  ) I
)

F |main ( ) I
(
1 S (  C 1 C abuild C 1 C 2 C 3 C 4 C 5 ) A I  a
```

```
2 S (   _  C Int|square ) F ( I  ) I
3 S (  N 1 C Array|map N 2 ) A I  b
4 S (  N 3 C index C 5 ) I
5 S (  C 5 C Int|factorial ) I
6 S (  N 4 C Int|+|Int N 5 ) I
)
```

In the case of map, as frequently occurs, the return graph is empty, and the output is taken from the agather node in the return interface graph. This would normally be the case for the factorial example too, so a multiply node has been artificially introduced to illustrate the case where the result does come from the return graph.

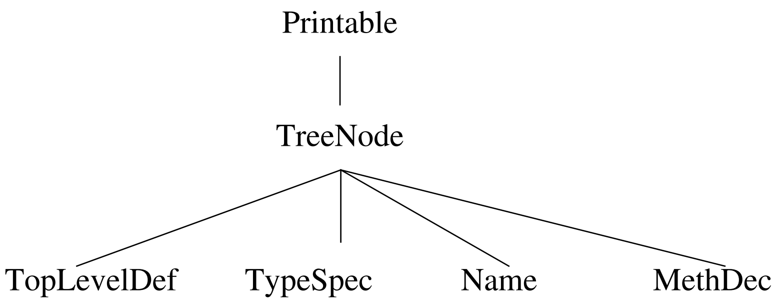
12 Internal representation

The standard uflow classes mirror the structure of the intermediate format closely, and provide little functionality beyond reading in the format and printing it out again. It is intended that users should either inherit from, or extend, these classes as they require and that over time any general-purpose functionality thus added should filter back into the standard classes.

The classes are stateless. In line with UFO philosophy, we avoid using stateful classes where there is no need. It is expected, however, the some applications (especially visualisation and debugging) may need to extend some of them to make them stateful.

12.1Top level definitions

The top of the inheritance hierarchy is as follows:



Printable is the abstract class of anything which can be represented as a string (via a print function). The string used in the case of the uflow classes is simply the uflow textual representation. A treenode is any element of the uflow graph (much of which is a tree).

TopLevelDefs are the definitions which can appear at the top level, i.e. ClassDefs, MethodDefs, TypeDefs TypeVarDefs, and EnumDefs, which define classes, methods, type synonyms, type variables, and enumerations respectively. The Name hierarchy is discussed below. MethDecs are method declarations.

Each class is built up from its components in a straightforward way. For instance, the header of class ClassDef is:

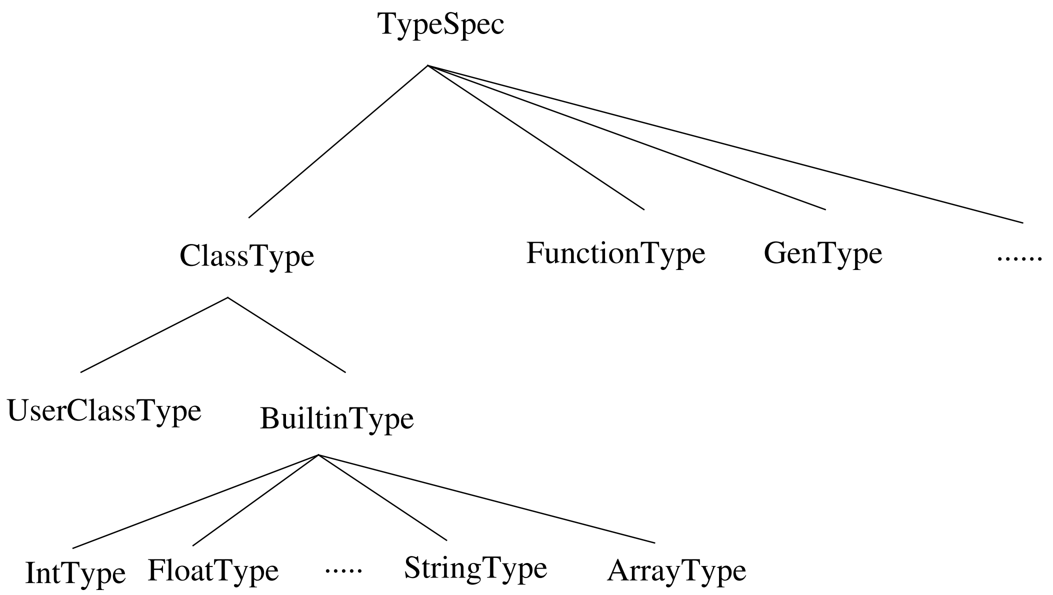
```
Classdef(
     status: Status;          deferred, closed, or “open” (the default).
     stateflag: Bool;         Whether the class is stateful
     name: String;            The name of the class
```

```
gens: List[String];              Class generic parameters
formals: List[ClassFormal];      Class formal parameters
super: String;                   Superclass name
initvals: InitiallyContainer;    Initialisation block.
methods: List[MethodDef]         The class methods
)
```

ClassFormal is a descendent of Name (see below). Containers are also discussed below.

12.2 Types

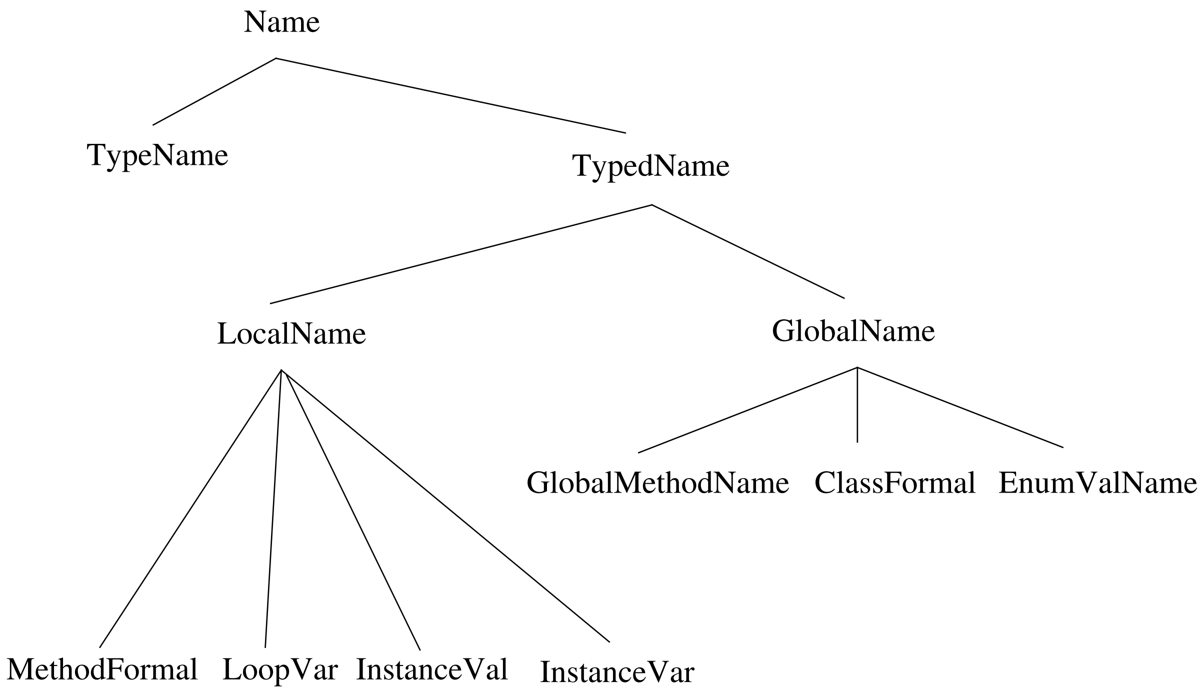
Types are represented by class TypeSpec and its descendents:



Most types correspond to classes, which are either user-defined classes or builtin ones. We choose to regard builtin types such as Ints as classes, since the number of such types is expected to increase in future versions of the language. Apart from scalars (Int, Float, Char, Bool, Bits) there are currently two other builtin classes, String and Array. We expect to add a hierarchy of Array classes, to represent multi-dimensional arrays, in the near future. Note that Strings are regarded as different from either List[Char] (List is just a user-defined class) or Array[Char]. The actual internal representation of strings is different from either of these; in particular it is optimised for append.

The other types available are function types, generic types (i.e. type variables), enumerated types, synonyms, and Void, the result type of a procedure which does not return a value.

12.3 Names



Names are divided into those which have types and those which represent types. The latter, which include class names, generic names, and enumerated type names, are not further distinguished as they are always associated with the relevant TypeSpec. TypedNames are either local or global. Local ones are formal parameters to methods, loop variables, class instance values or intance variables. Straightforward local names defined in blocks are represented as LocalNames.[1] GlobalNames are names of global methods, class formal parameters (considered as global because, if public, they can be accessed like global methods), or enumeration values.

12.4 Nodes, Slots and Containers

The structures above allow a syntax tree representation of the program, and are very similar to those in the compiler, which continues this process with a recursively defined Expression class. In the uflow, expressions have to be represented as a dataflow graph. This is done with the following classes:

- Class Node represents graph nodes. SimpleNodes represent operators or method application. CompoundNodes represent compounds: currently there are only two of these, IfNodes and LoopNodes.
- A Node has a number of Slots. A Slot may be a ConstantSlot (holding a constant value), an InputSlot, which expects an input from another node or from an input to the surrounding Container, or an EmptySlot, which does not expect an input, for partial application.

1. A more purist OO design would be to keep LocalName abstract and introduce an extra class, perhaps SimpleLocalName, for these.

- Nodes are held within Containers. A NormalContainer has a reference to the result Node of the graph, and the Slots of that Node points to its inputs, and so on, so the graph is built from the bottom up. For cases where there is no single result node (such as loop bodies), a MultiContainer holds a list of ‘‘bottom’’ nodes.

The simple, functional bottom-up representation of the graph is not intended to be adequate for all applications. Its purpose is merely to ensure that all the information is there, in an easy to scan form. Routines have been written to transform the graph into ‘‘top down’’ form, and some applications (particularly execution) will need to map the graph onto a stateful structure.[1]

12.5 Stubs and Skeletons

Use of these classes, by several different people for different purposes, poses some practical problems. Should they be regarded simply as a library, extendable only by inheritance, or should users take their own copies and extend them directly as required, integrating the results later? The problem with the former approach is that to organise such a hierarchy properly would require multiple inheritance: a BuiltinTypeWithWidget would need to inherit both from BuiltinType and WidgetfulThing. The classes are also not stable enough to be considered a proper library, but then coping with changes is even more of a problem for the second approach.

The solution we adopted is rather novel and it remains to be seen how effective it is. For each (abstract or concrete) class, Widget, say, there is a skeleton class WidgetSkel. All the functionality is contained in WidgetSkel, and Widget inherits from it, so the definition of Widget is merely a ‘‘stub’’:

```
class Widget(...) inherits WidgetSkel end
```

The idea is that the skeletons are held centrally, and may have new functionality added, bugs fixed etc., while the stubs never change (in fact the stubs are all held in one file). To extend a class, a user merely replaces the stub with a class containing the required routines. Almost any extension will work (modulo name clashes), the exception being that the user cannot change the constructor of a class. The reason for this is that some routines of WidgetSkel may need to call the constructor for Widget.

13 Conclusion

We have outlined a computational model for UFO which emphasises strictness and maximum parallelism. The computational view of the model is a traditional dataflow model with very simple properties. Nodes become enabled, can be fired in any order, and terminate. There is no notion of suspended execution in this view. Blocking on a stateful object corresponds to a node which has all its inputs but is not enabled yet. It becomes enabled when the object accepts the corresponding message, by whatever acceptance rules the object has. A complementary view shows the behaviour of individual objects, either on the outside only, or their internal states.

1. It is tricky to do execution in a purely functional manner, because sharing has to be preserved as values arrive at a node. It is possible to do by inserting an indirection - an association list - but the stateful way looks cleaner. In fact, our original assumption that a purely functional representation of the graph would be best now looks very questionable. This part of the structure is very much open to future review.

The textual intermediate format and the associated UFO data structures outlined above are currently being used in 3 projects:

1. To produce a visualisation and debugging tool which will animate computations as suggested above.
2. For sharing and array build/update in place analysis, similar to that employed in the SISAL project using IF1.
3. For analysis to optimise the implementation of stateful objects[Hoot94], as described earlier.

The uflow format will also be used in the compiler for the initial parallel implementation of UFO, on the KSR machine at Manchester, and hopefully for implementations on other parallel machines also.

14 References

[Agha86] G. Agha: **Actors: A Model of Concurrent Computation in Distributed Systems**, MIT Press series in artificial intelligence, 1986.

[ArTh81] Arvind, R.E. Thomas: **I-structures: An Efficient Data Type for Functional Languages**, M.I.T./LCS/TM-178, Computer Science Laboratory, M.I.T., Cambridge, MA, Oct 1981.

[Booc94] G. Booch: **Object-Oriented Analysis and Design with Applications**, second edition, Benjamin/Cummings 1994.

[Cann92] D. C. Cann: **Retire Fortran? A Debate Rekindled**, Communications of the ACM 35(8), August 1992, pp 81-89.

[HaBo92] M. Haines and A.P.W. Böhm: **Towards a Distributed Memory Implementation of SISAL**, Scalable High Performance Computing Conference, April 1992.

[HoAg92] C. Houck, G. Agha: **Hal: A High-level Actor Language and its Distributed Implementation**, proc. 21st International Conference on Parallel Processing, August 1992.

[Hoot94] S. Hooton: **Efficient Implementation of Stateful Objects**, M.Sc. thesis, Department of Computer Science, University of Manchester 1994. In preparation.

[Meye93] B. Meyer: **Systematic Concurrent Object-oriented Programming**, Communications of the ACM 36:9, September 1993, pp 56-79.

[McGr85] 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.

[Mohr90] E. Mohr, D.A. Kranz & R.H. Halsted: **Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs**, Proceedings of the ACM Conference on Lisp and Functional Programming. 1990.

[RuSa87] C.A. Ruggiero & J. Sargeant: **Control of Parallelism in the Manchester Dataflow Machine**, in Proc. 1987 Conference on Functional Programming Languages and Computer Architecture, Portland, Oregon, 1987.

[Saka89] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodoma, T. Yuba: **Architecture of a Dataflow Single Chip Processor**, proc 16th Annual Symposium on Computer Architecture, pp46-53, 1989

[Sarg91] J. Sargeant: **Improving Compilation of Implicit Parallel Programs by Using Runtime Information**, Proc. Workshop on Compilation of Symbolic Languages for Parallel Computers, San Diego, 1991, Argonne National Laboratory tech. report ANL-91/34, pp129-148.

[Sarg93a] J. Sargeant: **United Functions and Objects: An Overview** Technical report UMCS-93-1-4, Department of Computer Science, University of Manchester, 1993. (Available by anonymous ftp from ftp.cs.man.ac.uk. /pub/TR/UMCS-93-1-4.ps.Z)

[Sarg93b] J. Sargeant: **Implicit Parallelism: The United Functions and Objects Approac**h PARLE 93, Munich, June 1993. LNCS 694, pp120-132.

[Sarg93c] J. Sargeant: **Uniting Functional and Object-Oriented Programming** Invited Paper, International Sympoisum on Object Technologies for Advanced Software, Kanazawa, Japan, November 1993 LNCS 742, pp 1-26.

[SaWa91] J. Sargeant, I. Watson: **Some experiments in controlling the dynamic behaviour of parallel functional program**s, Proc. Workshop on the Parallel Implementation of Functional Languages, Southampton, June 1991, Southampton University tech. report CSTR 91-07, pp 103-121.

[SkWe85] S. K. Skedzielewski and M. L. Welcome: **Dataflow Graph Optimization in IF1** in J-P. Jouannaud (editor), Functional Programming Languages and Computer Architecture, pp17-34, Springer-Verlag, NY, 1985.

[Vali90] L. G. Valiant. **A Bridging Model for Parallel Computation** Communications of the ACM 33(8), pp 103-111, 1990.

[WaSa88] I. Watson, J. Sargeant, P. Watson, J.V. Woods, **The Flagship Parallel Machine**, Proc.CONPAR 88, Manchester, England, Cambridge University Press 1988.

[Wats94] I. Watson: **SLAM: Lazy Task Creation and Dynamic Load Balancing Using Active Messages**, Internal report, Department of Computer Science University of Manchester 1994. In preparation.

[Yone90] A. Yonezawa (ed.): **ABCL, an Object-oriented Concurrent System**, MIT press Computer Systems Series, 1990.