Implicit and Explicit Parallel Programming in Haskell

Mark P. Jones and Paul Hudak

Research Report YALEU/DCS/RR-982
August, 1993

This work is supported by the Darpa grant N00014-91-J-4043

Implicit and Explicit Parallel Programming in Haskell

Research Report YALEU/DCS/RR-982 [*]

Mark P. Jones and Paul Hudak
Department of Computer Science
Yale University
New Haven, CT 06520-2158
{jones-mark,hudak-paul}@cs.yale.edu

August 19, 1993

Abstract

It has often been suggested that functional languages provide an excellent basis for programming parallel computer systems. This is largely a result of the lack of side effects which makes it possible to evaluate the subexpressions of a given term without any risk of interference.

On the other hand, the lack of side-effects has also been seen as a weakness of functional languages since it rules out many features of traditional imperative languages such as state, I/O and exceptions. These ideas can be simulated in a functional language but the resulting programs are sometimes unnatural and inefficient. On the bright side, recent work has shown how many of these features can be naturally incorporated into a functional language without compromising efficiency by expressing computations in terms of *monads* or *continuations*. Unfortunately, the "single-threading" implied by these techniques often destroys many opportunities for parallelism.

In this paper, we describe a simple extension to the Haskell I/O monad that allows a form of *explicit* high-level concurrency. It is a simple matter to incorporate these features in a sequential implementation, and genuine parallelism can be obtained on a parallel machine. In addition, the inclusion of constructs for explicit concurrency enhances the use of Haskell as an executable specification language, since some programs are most naturally described as a composition of parallel processes.

[*] This research was supported by ARPA via a subcontract to Intermetrics, Inc.

1 Introduction

It has long been suggested that functional languages, particularly those with non-strict semantics, provide an excellent tool for parallel computation. The principal motivation for such claims is that, because of the lack of side-effects, there will often be opportunities to evaluate the subexpressions of a given term in parallel without any risk of interference. Furthermore, since there is no explicit mapping of specific tasks to particular processing units, a single functional program may be mapped onto a range of different parallel architectures. This mapping can be performed at compile-time or dynamically at run-time to make best use of the resources available without any need for reprogramming.

On the other hand, the lack of side-effects has also been viewed as one of the biggest disadvantages of functional languages. In particular, many features of traditional imperative languages – state, I/O, and exceptions, for example – are most naturally described in terms of side-effects. While it is possible to simulate these techniques in a functional language, the results may not always be satisfactory:

- Although it may be possible to code the same algorithm in a different way, the encoded programs are sometimes more cumbersome since the implicit state, continuations, etc. in an imperative language must be handled explicitly in the functional program.
- The resulting programs are not always as efficient as the corresponding imperative language, and optimization strategies are often complex, hard to reason about, and implementation dependent.

In recent years, there have been several proposals describing how these problems can be avoided without compromising the use of functional languages. In particular, we mention the use of *monads* [22, 23, 18] and *mutable abstract datatypes* (MADT's) [8]. The most important idea in each case is the use of an *abstract datatype* to control the way the imperative features are used. This goes a long way toward solving both problems mentioned above: "hiding" the imperative features in the ADT results in less cumbersome programs, and limiting the kinds of operations on them can lead to guaranteed efficient performance. Unfortunately, these efforts were aimed primarily at recovering the expressiveness and efficiency of *sequential* imperative languages, and thus the level of control is somewhat more restrictive than hoped, defeating many important opportunities for parallel execution.

In this paper, after a survey of the issues described above in Sections 2 and 3, we identify a class of *commutative* monads for which parallel computation is still possible, as captured by a new *fork* primitive in Haskell's I/O monad (Section 4). The commutativity of these monads is what guarantees *determinacy* in their parallel execution, even in the presence of side-effects. As with sequential computation, however, being able to express non-determinate computation is another degree of freedom and expressiveness, and yields even more opportunities for parallelism. As is well-known, this indeterminacy is often only at a local level, placing proof obligation of determinacy at a more global level on the programmer (for example, parallel updates to an array is determinate if no one element is updated more than once).

Motivated by this, Section 5 outlines an extension to the Haskell I/O monad to support *communicating processes* in the style Hoare's CSP [5] or occam[1] [11, 12]. In particular, we treat channels as *first-class values* and use them to interconnect processes (functions participating in the I/O monad). Several examples are given of this explicit style of parallel functional programming: pipelines (Section 6), parallel sieve of Eratosthenes (Section 7, demonstrating the ability to generate new parallel process dynamically), and a sorting program (Section 8).

We assume the reader to be familiar with Haskell or similar functional language; we have in fact implemented the ideas in this paper in Gofer, an "extended subset" of Haskell. Implementation issues are discussed in Section 9. Finally, Section 10 concludes with pointers to areas for further work and investigation.

2 Parallel Execution of Functional Programs

Functional languages, particularly those with non-strict semantics and an absence of side-effects, have often been suggested as powerful tools for programming parallel computer systems.

As a simple, almost trivial illustration of this, suppose that we wish to find the value of an expression of the form $e_1 + e_2$. This in turn requires that we evaluate both e_1 and e_2. In an imperative language, the evaluation of either of these subexpressions may cause a side-effect which may affect the value of the other subexpression. As a result, if we want a well-defined semantics for the language, then we must arrange for the two expressions to be evaluated sequentially in some predefined order. On the other hand, a pure functional language's semantics guarantees that neither evaluation can interfere with the other so we can evaluate the arguments in parallel and then, when both argument values are known, calculate their sum.

2.1 Implicit Parallelism

The ability to write programs without worrying about how they will be mapped onto particular parallel architectures has obvious benefits. However, if this information is not included explicitly as part of the program then we need to find a good automated strategy for deciding when to start new parallel computations. Much work has been done in this area, both static compile-time techniques and dynamic run-time techniques, and ranging in degree of parallelism from conservative approaches in which parallel computations are started only when

[1] Occam is a trademark of the INMOS group of companies.

it is certain that their results will be needed, to more liberal approaches where evaluations are begun speculatively, before it is known for sure that the results will be needed. For lazy languages, various forms of *strictness analysis* play a key role in uncovering useful forms of parallelism. It is beyond the scope of this paper to discuss these issues in any detail, but in this section we give a few programming examples in which the parallelism is implicit.

2.2 A Simple Form of the N-body Problem

Consider a simple form of the n-body problem, modeling the behaviour of a collection of bodies subject, for example, to mutual gravitational or electrostatic forces. This example is inspired by work on the parallel language *Proteus* [15].

To avoid going into details about any particular physical model, we assume only that we have already been given functions:

$$\begin{array}{lll} \mathit{between} & :: & (\mathit{Body}, \mathit{Body}) \rightarrow \mathit{Force} \\ \mathit{move} & :: & \mathit{Force} \rightarrow \mathit{Body} \rightarrow \mathit{Body} \end{array}$$

that can be used to calculate the force on one body as a result of a second, and the new position of a particular body subject to a given force for some small, fixed unit of time δt. (We assume that the *Body* type includes physical attributes of the body such as its mass, velocity and position.)

The simulation of a collection of bodies can be modeled as a sequence of different configurations, with the transition from one to the next defined as:

$$\begin{array}{lll} \mathit{next} & :: & [\mathit{Body}] \rightarrow [\mathit{Body}] \\ \mathit{next}\ \mathit{bs} & = & [\,\mathit{move}\ (\mathit{sum}\ [\,\mathit{between}\ (b, b')\ \mid\ b' \leftarrow \mathit{bs}\,])\ b\ \mid\ b \leftarrow \mathit{bs}\,] \end{array}$$

In other words, for each body we calculate the sum of the forces acting on it, and use that to calculate its new position. This definition can be used exactly as it is written here in a standard sequential implementation of Haskell. But with an appropriate parallel evaluation strategy, the same program could be executed on a parallel architecture as well, the nested list comprehensions being used to distribute the calculation of inter-body forces over an array of processors.

2.3 Making Parallelism Explicit

While the treatment of parallelism in the example above seems rather attractive, it assumes a parallel implementation good enough to exploit the inherent parallelism. Even if successful, it may be difficult for the programmer to *reason* about the resulting behavior, since doing so requires a good understanding of the particular implementation being used.

One way to try and obtain more reliable performance on parallel architectures is to include "annotations" in the program to indicate when parallel computation might be beneficial. For example, we might choose to write $par\ x\ y$ to indicate that the two values x and y are to be evaluated in parallel, returning the evaluated pair (x, y) only when both components have been evaluated. Semantically, the *par* function can be thought of as the projection:

$$\begin{array}{lcl} par & :: & a \rightarrow b \rightarrow (a, b) \\ par\ x\ y & = & \textbf{if}\ x \neq \bot \land y \neq \bot\ \textbf{then}\ (x, y)\ \textbf{else}\ \bot \end{array}$$

The use of annotations such as these is the basis, for example, of *para-functional programming* [7], in which both explicit mapping and explicit scheduling of expression evaluation is permitted. In this paper we will explore this idea only to the extent implied above: a simple way to provide a "hint" to the compiler that parallelism is intended.

One might argue that providing such a "hint" still does not solve the portability problem: a particular implementation could simply ignore the hint! However, having the hints at least provides a precise handle on which implementations can declare to either execute things in parallel or not, and furthermore one could provide a formal parallel operational semantics that a valid parallel implementation would be obliged to satisfy.

We also note that introducing the *par* function into a program may affect its semantics; for example:

$$fst\ (1, 3/0) = 1 \neq \bot = fst\ (par\ 1\ (3/0)).$$

A sophisticated projection-based strictness analyzer may sometimes be able to determine when the *par* projection can be inserted without changing the semantics of the program, but this causes all the same problems with portability and ease-of-reasoning mentioned earlier. Furthermore, as parallel programmers, we may wish to limit the places where such annotations are inserted in the first place.

The *par* function is easily extended to other datatypes. For example, the following definition might be used to describe the parallel evaluation of the elements of a list:

$$\begin{array}{lcl} parl & :: & [a] \rightarrow [a] \\ parl\ [\,] & = & [\,] \\ parl\ (x : xs) & = & (y : ys)\ \textbf{where}\ (y, ys) = par\ x\ (parl\ xs) \end{array}$$

Of course, for the purposes of compiler optimizations, it might be best if functions like this are included in the system as primitives, rather than user-defined functions whose properties must be determined by static analysis. One way to exploit this may be to explore the apparent connection between functions like *par* and *parl* and the use of ***strict data constructors***, as provided in some implementations of Haskell [1]. Alternatively, these functions may be defined directly using the lower-level annotations of para-functional programming.

To illustrate the use of these annotations, the only change to the *next* function described in the treatment of the n-body problem in the previous section would be to insert a call to *parl* at the outermost level:

$$\begin{array}{lcl} \mathit{next} & :: & [\mathit{Body}] \rightarrow [\mathit{Body}] \\ \mathit{next}\ \mathit{bs} & = & \mathit{parl}\ [\,\mathit{move}\ (\mathit{sum}\ [\,\mathit{between}\ (b, b') \mid b' \leftarrow \mathit{bs}\,])\ b \mid b \leftarrow \mathit{bs}\,] \end{array}$$

We might also want to consider adding another call to *parl* for each of the inner lists involved in this expression. However, in a parallel implementation, it would be reasonable to expect that the opportunity to evaluate the elements of these lists in parallel would already be incorporated as part of the definition of *sum*. For example, the *sum* function might well be defined using an equation of the form $sum = sum'\ .\ parl$ for some suitable list-summing function *sum'*.

Of course, even with some annotations, it may still be possible to get better performance on particular architectures by expressing the algorithm in a different form. Following again the work in [15], to obtain good performance on a SIMD/vector machine the definition of *next* above can be rewritten as:

$$\begin{array}{lcllcl} \mathit{next} & :: & \multicolumn{4}{l}{[\mathit{Body}] \rightarrow [\mathit{Body}]} \\ \mathit{next}\ \mathit{bs} & = & \mathbf{let} & \mathit{qs} & = & \mathit{parl}\ [\,(b, b') \mid b \leftarrow \mathit{bs},\ b' \leftarrow \mathit{bs}\,] \\ & & & \mathit{fs} & = & \mathit{parl}\ (\mathit{mapbetweenqs}) \\ & & & \mathit{vs} & = & \mathit{parl}\ (\mathit{segSum}\ (\mathit{lengthbs})\ \mathit{fs}) \\ & & & \mathit{bs'} & = & \mathit{parl}\ (\mathit{zipWith}\ \mathit{move}\ \mathit{vs}\ \mathit{bs}) \\ & & \mathbf{in} & \mathit{bs'} & & \end{array}$$

Although this may appear a bit odd at first sight, the idea is that each line in the definition corresponds to a single vector-level operation, as implied by the explicit *parl* annotations. Of course, further refinement may be necessary in some cases, depending on the definitions of *between* and *move* and on the underlying architecture. (*zipWith* is defined in the standard prelude for Haskell. The *segSum* function is not included but is easily defined: *segsum n fs* is evaluated by splitting the list *fs* into *n* segments of length *n* and calculating the sum of the values in each segment.)

Defining and applying effective transformation rules targeted at particular kinds of processor architecture is difficult. This is an important area of current research and will not be addressed here, other than to point out that transformations such as employed in Proteus [19], for example, are most easily carried out in a functional (as opposed to imperative) framework.

3 Functional Programming and Side-Effects

In the previous sections, we have seen how the lack of side-effects in a functional language leads to a simple treatment of parallel computation. At the same time, the lack of side-effects is often cited as an inherent weakness of functional languages:

- I/O, for example, seems to inherently involve side effects. Many useful forms of I/O can actually be expressed in terms of *lazy streams* without using side-effects, and this is in fact the basis of the I/O design for Haskell [9]. However, for some applications, it has been suggested that programming in this style is cumbersome and unnatural.[2]

- Efficiency is another important issue. To illustrate this, consider a very simple treatment of *arrays* in a functional language, using the functions:

$$\begin{array}{lll} \mathit{mkArray} & :: & \mathit{Int} \rightarrow \mathit{Array}\ a \\ \mathit{lookup} & :: & \mathit{Int} \rightarrow \mathit{Array}\ a \rightarrow a \\ \mathit{update} & :: & \mathit{Int} \rightarrow a \rightarrow \mathit{Array}\ a \rightarrow \mathit{Array}\ a \end{array}$$

  which describe array construction, indexing and updating, respectively. Although "natural" in a functional setting, this design unfortunately may be very inefficient, because in general the *update* operation will require making a complete copy of its array argument before making the update, which is considerably more expensive than a simple in-place update.

  This is the well known "aggregate update problem" in functional languages. Many solutions to the problem have been proposed, but space limitations preclude a detailed discussion here. The solution that most interests us, however, is described in the next section.

3.1 Mutable ADTS, Monads, and Continuations

In recent years, it has become clear that many of the problems described above can be avoided without breaking any of the properties of purely functional programs. In short, we can have our cake and eat it too![3]

The essential idea is to use an *abstract datatype* (ADT) to control the way in which imperative features are used. In particular, the state itself is "hidden" inside the ADT, making it implicit in much the same way that state is implicit in an imperative language. The operations on the ADT are then limited to two kinds: (1) operations that manipulate (read, write, etc.) the state, and (2) combinators that compose the operations in (1). All of these operations can be given purely functional semantics, but they are collectively designed in such a way that the state is always "single-threaded," thus permitting a safe and efficient implementation using side-effects. We will refer to such an ADT as a *mutable* ADT, or MADT for short.

The first example of an MADT was discovered by Wadler for an *array* ADT in his work on *monads* [22, 23]. Hudak later generalized the approach by discovering both *direct* and

[2] it often seems that the term unnatural is used as a synonym for "not the same as in C'. In such cases, it might well be fairer to describe functional I/O as being unfamiliar rather than unnatural.

[3] Or, as expressed in [8], "We can have our state and munge it to!" where "munge" is a highly technical word meaning "to mutate."

continuation-passing versions of the array MADT [8]. Hudak was also able to identify a large class of ADT's for which MADT's in monadic, direct, or continuation-passing style could be derived *automatically*. Wadler and Peyton-Jones, in the meatime, continued with the monadic style to develop a form of *monadic I/O*, which has now been adopted by both the Yale and Glasgow Haskell Projects as the preferred mode of I/O [18]. In our work on parallelism in this paper we will also express things in a monadic style.

The discovery of this general technique for incorporating imperative-like features in a functional language has created somewhat of a new style of functional programming, referred to by Wadler and Peyton-Jones as "imperative functional programming." The idea has received quite a bit of attention because it eliminates many of the criticisms often heard about functional langauges, and opens up interesting new application areas as well.

3.2 Monad Basics

Any monad can be described by a (unary) type constructor m together with a collection of operations. Using the notation of constructor classes [13], the class of monads can be specified as:

$$\begin{array}{lll} \multicolumn{3}{l}{\textbf{class}\ \mathit{Monad}\ m\ \textbf{where}} \\ \quad \mathit{result} & :: & a \rightarrow m\ a \\ \quad \mathit{bind} & :: & m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b \end{array}$$

The constructor class notation is convenient because it allows us to use the same names for generic operations on monads. However, no detailed knowledge of constructor classes is required to be able to understand this paper.

One way to understand the use of monads is to think of values of type $m\ a$ as representing computations which return values of type a. This distinction between computations and values reflects the fact that the use of particular programming language features in a given calculation is a property of the computation itself and not of the result that it produces.

As the definition above indicates, every monad has at least two operations, *result* and *bind* – these are the "combinators" referred to earlier. They are an essential part of the monad, but we will often say simply that a particular type constructor "is a monad" assuming that suitable definitions of these two operations have been specified for that constructor. The intuitive meaning for these operations is as follows:

- An expression of the form *result* e represents the trivial computation which produces the result e with no further action.

- The *bind* operator can be thought of as a way of sequencing two computations. It is usually convenient to write *bind* as an infix operator (just as the semi-colon is the infix sequencer in an imperative language), which can be accomplished in Haskell by enclosing the operator name between backquotes. An expression of the form m ‘*bind*‘ f

denotes a computation which first carries out the computation described by m to obtain a result r. Applying the function f to this result gives a new computation $f\ r$ which is executed to obtain a final result.

3.3 Algebraic Properties of Monads

Technically, to complete the definition of a monad, the *bind* and *result* functions are also required to satisfy a small collection of algebraic laws. These laws are not important to understanding the rest of the paper, but we include them for completeness. It is particularly useful to be able to refer to laws like these during program development and proof.

The laws can be stated directly in terms of *bind* (as is done in [23]), but they are much simpler if we first introduce an auxiliary function (called the *Kleisli composition*) defined by:

$$\begin{array}{lcl} (@@) & :: & Monad\ m \Rightarrow (b \rightarrow m\ c) \rightarrow (a \rightarrow\ m\ b) \rightarrow (a \rightarrow\ m\ c) \\ (f@@g)\ x & = & g\ x\ \text{`}bind\text{`}\ f \end{array}$$

The expression *Monad m* in the type of this function indicates that the (@@) operator can be used for any monad m. The algebraic laws that the monad operators must satisfy can now be stated very simply as:

$$f@@result = f \qquad result@@g = g \qquad (f@@g)@@h = f@@(g@@h)$$

for any f, g and h of suitable types. In other words, the Kleisli composition is associative, with *result* as both a left and right identity.

3.4 Monads and Arrays

To make some of the descriptions in the previous sections a little more concrete, we will briefly outline a monadic implementation of arrays with efficient array update and lookup. The reader should refer to [22, 23, 8] for further insight and motivation.

We start with the definition of two abstract types, one for the arrays and one for the monad which will be used to control the way that these arrays are used.

```
data RefArr a      -- References to arrays of values of type a, indexed by integers
data A a           -- The array monad
```

Elements of type *RefArr a* should be thought of as references, or pointers, to arrays of values of type a rather than as the arrays themselves.

The three primitives described above for array creation, update and lookup will be represented in this framework by the following operations on the abstract datatype:

$$\begin{array}{lcl} mkArray & :: & Int \rightarrow A\ (RefArr\ a) \\ update & :: & RefArr\ a \rightarrow Int \rightarrow a \rightarrow A\ () \\ lookup & :: & RefArr\ a \rightarrow Int \rightarrow A\ a \end{array}$$

There are two interesting points to notice here. First, the return type of each of these functions involves the type constructor A. This illustrates how the abstract datatype helps to control the way that these values are used; the only operations that can be applied to such values are those which are defined as part of the ADT. The second point is that the return type of ***update*** is () rather than *RefArr a*. There is no need for ***update*** to return a new array since the array update will be "hidden," and performed in-place. (We adopt the convention that functions which are used purely for their effect return a value of type ().)

By themselves, the operations above do not enable us to carry out any useful computations: they provide various ways of constructing values of type of $A\ a$, but there are no operations that can be used to combine or manipulate such values! Recall, however, that this is what the monad combinators are for. Thus we assume that A is a monad:

```
instance Monad A  -- defines monad operations for A
```

In effect, this declaration indicates that we can use the standard monad operations:

```
result  ::  a → A a
bind    ::  A a → (a → A b) → A b
```

to combine values of type $A\ a$ and return results.

The following definition shows how these monadic versions of array operations might be used to implement the *swap* function described earlier:

```
swap        ::  RefArr a → Int → Int → A ()
swap a i j  =   lookup a i      ‘bind‘  \ ai →
                lookup a j      ‘bind‘  \ aj →
                update a i aj   ‘bind‘  \ () →
                update a j ai   ‘bind‘  \ () →
                result ()
```

Finally, we need a mechanism to create a "local imperative scope" where arrays may be allocated, manipulated, and eventually discarded. We use the function ***beginArr*** of type $A\ a \rightarrow a$ for this purpose. As an example, here is a simple program using all of the functions presented so far, including a call to the function *swap*; the reader should verify that the result is the value *True*:

```
beginArr
(mkArray 10           ‘bind‘  \ a →
 update a 1 True      ‘bind‘  \ () →
 update a 2 False     ‘bind‘  \ () →
 swap a 1 2           ‘bind‘  \ () →
 lookup a 2           ‘bind‘  \ x →
 result x)
```

The implementations of these functions, together with the implementation of A, is hidden. However, it may be helpful to think of an expression of type A a as evaluating to a *command* that will produce a value of type a when they are executed. Commands are also sometimes described as *state transformers* because they can be thought of as functions of type $State \to (a, State)$ that map some initial state value to a final state, together with the result value. The following (informal) definition gives an indication of how the semantics of the monad operations might be described in this framework:

```
instance Monad A where
  result x    =  \s → (x, s)
  m `bind` f  =  \s → let (x, s′) = m s in f x s′
```

In practice, since the operations on A a ensure that the state is single-threaded, we can actually avoid passing the state around as a runtime parameter.

3.5 Monads and Sequential Computation

Two of the most fundamental constructs in an imperative language are the *skip* command and the ability to sequence one command after another, often represented by a semicolon. These can be defined in an arbitrary monad as follows:

```
skip          ::  Monad m ⇒ m ()
skip          =   result ()

bind_         ::  Monad m ⇒ m a → m b → m b
p `bind_` q   =   p `bind` \_ → q
```

The underscore character used for the argument of the λ-expression in the definition of *bind_* is a "wildcard" and indicates that any value returned by the computation p will be ignored. This, coupled with the fact that the semi-colon symbol is already used for other purposes in Haskell, motivates our use of the name *bind_*.

*bind_* can be used to simplify somewhat the programs given earlier. It can also be used to define a simple way to sequence a list of commands:

```
seq  ::  Monad m ⇒ [m a] → m ()
seq  =   foldr (bind_) skip
```

A typical application of this might be initializing the elements of an array to zero:

```
initArray      ::  RefArr Int → Int → A ()
initArray a n  =   seq [update a i 0 | i ← [0..n]]
```

4 Parallel Execution in a Monad

Initializing the elements of an array as in the example above is an obvious place where we might hope to use parallel evaluation. Since each *update* command changes a different location in the array, there is no reason we should not update all of the entries in the array at the same time. Unfortunately, the semantics of the monad require that all of the updates are carried out in strict sequence.

The annotations for explicit parallelism discussed in Section 2.3 are not particularly useful here either. For example, we might try to write the definition of *initArray* using *parl* as:

$$\begin{array}{lcl} \mathit{initArray} & :: & \mathit{RefArr\ Int} \rightarrow \mathit{Int} \rightarrow A\ () \\ \mathit{initArray}\ a\ n & = & \mathit{seq}\ (\mathit{parl}\ [\,\mathit{update}\ \ a\ \ i\ \ 0 \mid i \leftarrow [0..n]\,]). \end{array}$$

All this accomplishes is the *evaluation* of the update commands in parallel; they will still be *executed* sequentially. Removing the call to *seq* would not solve this problem; the *initArray* function would just return a list of (unexecuted) commands.

To be able to do any useful form of parallel computation in a monad we need some way of executing commands in parallel. The class of monads which support this kind of parallel execution might be defined as:

$$\begin{array}{l} \textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{ParMonad}\ m\ \textbf{where} \\ \quad \mathit{fork}\ \ ::\ \ m\ a \rightarrow m\ b \rightarrow m\ (a, b) \end{array}$$

The basic idea here is that *fork* executes a command of type $m\ a$ in parallel with a second command of type $m\ b$. When the two processes have both terminated, the *fork* command returns the result of each as a pair. The *fork* function is essentially a monadic version of the *par* function defined in Section 2.3; for example, the types of the two functions are very similar, the only difference being that the type for *fork* wraps each argument (and the result) inside the monad constructor.

4.1 Commutative Monads

There are two simple ways that we might define a *fork* function for an arbitrary monad that are, at the very least, type correct:

$$\begin{array}{lcl} \mathit{fork}_1\ p\ q & = & p\ \text{`}\mathit{bind}\text{`}\ \backslash\, x \rightarrow q\ \text{`}\mathit{bind}\text{`}\ \backslash\, y \rightarrow \mathit{result}\ (x, y) \\ \mathit{fork}_2\ p\ q & = & q\ \text{`}\mathit{bind}\text{`}\ \backslash\, y \rightarrow p\ \text{`}\mathit{bind}\text{`}\ \backslash\, x \rightarrow \mathit{result}\ (x, y) \end{array}$$

However, we may be concerned that these definitions do not give the correct operational behaviour; *fork₁* appears to run the computation *p* first, followed by *q*, while *fork₂* adopts the reverse order.

It is important to realize that, in the context of a non-strict language, these assumptions about the order of evaluation need not be true. In the general case, there is no reason why the monadic *bind* operator should induce a strict, left-to-right, sequential order of evaluation. There are already a number of examples in the literature which illustrate this point; for example:

- Launchbury [14] has investigated the use of a non-strict *bind* operator to give a semantics for state-based computations that is analogous to lazy evaluation.
- Wadler [23] describes the use of a form of state monad in which the state propagates from right-to-left as the computation proceeds from left-to-right.
- Fasel [4] describes an array monad that permits parallel updates on contiguous, but disjoint, portions of an array, as well as *fork* and *wait* primitives for spawning parallel tasks; Fasel's work is most similar to ours.

As an example of a monad in which the *bind* operator does not require any form of sequential evaluation, consider the following specification for a *Gensym* monad, motivated in part by the work reported in [21]:

```
data Gensym a               -- the Gensym monad
instance Monad Gensym

data Name                   -- An abstract type of names with
instance Eq Name            -- equality as the only operation

gensym :: Gensym Name       -- A process to generate new names
```

The *Gensym* monad is useful in applications where it is necessary to be able to obtain "new names" as a computation proceeds. Typical applications include renaming, converting a tree to a DAG, and generating "fresh" type variables in a type checker. Names are represented by the abstract datatype *Name*. Adapting an example presented in [13], and given a datatype of trees defined by:

```
data Tree a = Leaf a | Tree a :^: Tree a
```

the *Gensym* monad might be used to label each node of the tree with a distinct *Name* using the function:

```
label            :: Tree a → Gensym(Tree (a, Name))
label (Leaf x)    = gensym              ‘bind‘ \ n →
                    result (Leaf (x, n))
label (l :^: r)   = label l             ‘bind‘ \ l' →
                    label r             ‘bind‘ \ r' →
                    result (l' :^: r')
```

New names are obtained using *gensym*, and the only other operation on names is an equality test. As a result, it is impossible to distinguish between the two functions $fork_1$ and $fork_2$ when using this monad, and thus $fork_1 = fork_2$. So, using either of these as a definition for *fork*, the last case in the definition of *label* can be rewritten as:

$$\begin{array}{lcl} label\ (l :\hat{}: r) & = & fork\ (label\ l)\ (label\ r)\ \text{`}bind\text{`}\ \backslash(l', r') \rightarrow \\ & & result\ (l' :\hat{}: r') \end{array}$$

The use of *fork* serves to highlight the fact that the left and right subtrees can be labeled *in parallel*; a single name supply will still be required to implement this, but the interleaving of names used by the two subprocesses may be chosen arbitrarily.

Monads in which the equation $fork_1 = fork_2$ holds are often described as *commutative monads*. This property can be expressed more elegantly using the notation of monad comprehensions [22, 13]:

$$[\,(x, y)\ \mid\ x \leftarrow xs,\ y \leftarrow ys\,] = [\,(x, y)\ \mid\ y \leftarrow ys,\ x \leftarrow xs\,]$$

In addition, this formulation probably makes it a little easier to see why such monads might be described as being commutative.

Clearly, computations in commutative monads are well-suited for parallel execution. Unfortunately, very few of the monads that are useful in functional programming are commutative. Of the nine or so different monads described in [22], for example, only two – the identity and strictness monads[4] – are commutative. The strictness monad is particularly interesting for our purposes because it can be used to define the *par* and *parl* functions described in Section 2.3:

$$\begin{array}{lcl} par\ p\ q & = & [\,(x, y)\ \mid\ x \leftarrow p,\ y \leftarrow q\,]^{Str} \\ parl\ [\,] & = & [\,[\,]\,]^{Str} \\ parl\ (x : xs) & = & [\,z : zs\ \mid\ z \leftarrow x,\ zs \leftarrow xs\,]^{Str} \end{array}$$

4.2 Algebraic Properties of *fork*

With the discussion of the previous section in mind, we consider briefly the kinds of algebraic properties that we would expect an implementation of *fork* to satisfy. Intuitively, we expect the order of the two parallel processes not to be significant, but we have to be careful to get the types correct: If $p :: IO\ a$ and $q :: IO\ b$, then $fork\ p\ q :: IO\ (a, b)$ while

[4] The *result* function for each of these monads is just the identity function. The *bind* operator for the identity monad is just function application, $(\backslash x\ f \rightarrow f\ x)$. In the strictness monad, $bind = \backslash x\ f \rightarrow strict\ f\ x$ where *strict* is the (non λ-definable) function described by:

$$strict\ f\ x \;=\; \textbf{if}\ x = \bot\ \textbf{then}\ \bot\ \textbf{else}\ f\ x$$

fork q p :: *IO* (b, a). So the closest we can come to a commutative law is that, for all p and q of the appropriate types:

$$fork\ p\ q = fork\ q\ p\ \text{`}bind\text{`}\ exch$$

where $exch = \backslash(x, y) \rightarrow result\ (y, x)$. In a similar way, we can express a kind of associativity law for *fork*, as follows. For all p, q and r of the appropriate types:

$$\begin{aligned} fork\ (fork\ p\ q)\ r &= fork\ p\ (fork\ q\ r)\ \text{`}bind\text{`}\ \backslash((x, y), z) \rightarrow result\ (x, (y, z)) \\ fork\ p\ (fork\ q\ r) &= fork\ (fork\ p\ q)\ r\ \text{`}bind\text{`}\ \backslash(x, (y, z)) \rightarrow result\ ((x, y), z) \end{aligned}$$

Once again, the monad comprehension notation provides a more concise and elegant way to express both commutativity and associativity:

$$\begin{aligned} [(x, y) \mid (x, y) \leftarrow fork\ p\ q] &= [(x, y) \mid (y, x) \leftarrow fork\ q\ p] \\ [(x, y, z) \mid ((x, y), z) \leftarrow fork\ (fork\ p\ q)\ r] &= [(x, y, z) \mid (x, (y, z)) \leftarrow fork\ p\ (fork\ q\ r)] \end{aligned}$$

4.3 Forks and Side-Effects

Now let us return to the array initialization problem discussed at the beginning of this section. We cannot hope to be able to define a safe *fork* function for the array monad A. To understand why, suppose that we wrote a program fragment of the form:

$$fork\ (update\ a\ i\ x)\ (update\ a\ j\ y)$$

The basic idea here is to update the array locations i and j with values x and y in parallel. But what does this command actually mean? If the values of i and j are distinct, then the result is equivalent to both of the following expressions:

$$\begin{aligned} &fork_1\ (update\ a\ i\ x)\ (update\ a\ j\ y) \\ &fork_2\ (update\ a\ i\ x)\ (update\ a\ j\ y) \end{aligned}$$

But what if the values of i and j coincide? In that case, there is no obvious way to define a deterministic semantics for the original expression.

The problem here is as before: the array monad is simply not cummutative. Some side-effecting monads are commutative, however. As an example, consider a "histogram" monad in which all one can do is *increment* positions in an array – i.e. general update is disallowed. It is easy to see that the increment operation is commutative (in the same way that addition is commutative), even though the array is being updated destructively. For this monad high degrees of parallelism are possible with a deterministic semantics.

The frustrating aspect of the array monad, however, is that, if used with caution, it may in fact be commutative (for example if it's never the case that i and j coincide in the above example). However, in the general case, we cannot expect a compiler to detect this situation;

instead we must rely on the programmer. Another possibility is to install dynamic run-time checks, as is done in [4], but this is an added computational overhead.

We are faced with two conflicting alternatives. On the one hand, a purist would argue that we should not include *fork* in any non-commutative monad since it would compromise safety. In reply, a pragmatist might suggest that this loses valuable opportunities for parallelism. For the remaining sections of this paper, we will take the second position, accepting that, wherever a *fork* is used, there is a proof obligation on the programmer.

To simplify the development of the programs in the rest of this paper we will introduce some additional utility functions. We will often be interested in processes that are executed for their effect rather than for their final result. The (<||>) operator defined below can be used to run two such computations, discarding the final result from each:

$$\begin{array}{lcl} (<\|>) & :: & \mathit{ParMonad}\ m \Rightarrow m\ a \to m\ b \to m\ () \\ p <\|> q & = & \mathit{fork}\ p\ q\ \text{`}\mathit{bind\_}\text{`}\ \mathit{skip} \end{array}$$

It follows immediately from the algebraic properties of *fork* in the previous section that (<||>) is both associative and commutative.

Using the definition of *seq* in Section 3.5 as a guide, the parallel execution of a list of commands can be described using the following function:

$$\begin{array}{lcl} \mathit{parCmds} & :: & \mathit{ParMonad}\ m \Rightarrow [m\ a] \to m\ () \\ \mathit{parCmds} & = & \mathit{foldr}\ (<\|>)\ \mathit{skip} \end{array}$$

Assuming now that a *fork* function has been defined for the array monad, we can write a parallel version of the *initArray* function:

$$\begin{array}{lcl} \mathit{initArray} & :: & \mathit{RefArr}\ \mathit{Int} \to \mathit{Int} \to A\ () \\ \mathit{initArray}\ a\ n & = & \mathit{parCmds}\ [\,\mathit{update}\ a\ i\ 0 \mid i \leftarrow [0..n]\,] \end{array}$$

It is easy to see that the use of *parCmds* (and hence indirectly of *fork*) is safe in this example since the elements of [*0..n*] are distinct. With a change of notation, this definition of *initArray* coincides with the array initialization operation described in [6].

5 Communicating Sequential Processes

In this and subsequent sections, we restrict our attention to a single monad that supports parallel execution using *fork* together with interprocess communication channels. The design has been strongly influenced by some of the ideas used in the programming language occam and in Hoare's study of communicating sequential processes, CSP.

We have constructed a prototype implementation (described in further detail in Section 9) of these ideas using an extension of the I/O monad in the current version of Gofer (a pared down version of the ideas described in [18]). The type constructor part of this monad is written *IO* and we will assume that suitable implementations of:

$$\begin{array}{lcl} \mathit{result} & :: & a \to IO\ a \\ \mathit{bind} & :: & IO\ a \to (a \to IO\ b) \to IO\ b \\ \mathit{fork} & :: & IO\ a \to IO\ b \to IO\ (a, b) \end{array}$$

have also been provided.

We should mention that the decision to implement these ideas as an extension of the *IO* monad was motivated largely by the desire to speed up the process of developing the prototype. It may well be sensible to reconsider this decision in the future to provide a more coherent framework for programmers (for example, to reduce the overlap between channel I/O and conventional operating system I/O).

5.1 Communication Channels

To allow otherwise independent parallel processes to interact with one another, we introduce a simple form of *channel* for point-to-point communication. Channels carrying values of type a will be represented by the type $\mathit{Chan}\ a$. Notice that all of the values transmitted on a single channel are required to have the same type. The original definition of occam [11] only allows single machine words to be transmitted on a channel. In contrast, we allow arbitrary types of value to be passed down a channel, including lists, functions, arbitrary data structures, IO processes, and even other channels! The sequential and variant protocols of occam 2 [12] are easily dealt with in this framework by defining a suitable datatype of values to be sent over the channel using product and sum types, respectively.

New channels are created using the *newChan* primitive function:

$$\mathit{newChan} \ :: \ IO\ (\mathit{Chan}\ a)$$

while the following primitives are used to deal with channel input and output:

$$\begin{array}{lcl} \mathit{input} & :: & \mathit{Chan}\ a \to IO\ a \\ \mathit{output} & :: & \mathit{Chan}\ a \to a \to IO\ () \end{array}$$

Obviously, the execution of an input command may require the input process to be suspended until a value has been output on that channel. We will make the relationship between input and output more symmetric by making a similar requirement for output commands. Specifically, any output command will be suspended until another process is ready to receive the output value. It is an error for two processes to make use of a single channel at the same time unless one is using the channel for input and the other is using it for output. In short, channels provide exclusive, synchronous, unbuffered communication between parallel processes.

The following two expressions illustrate the use of these primitives in a simple example:

```
prog1  =  newChan                          `bind`  \ c →
          fork (input c) (output c 42)     `bind`  \ (v,_) →
          result v

prog2  =  newChan                          `bind`  \ c →
          fork (output c 42) (input c)     `bind`  \ (_,v) →
          result v
```

Each of these programs generates a new channel and then runs two parallel processes, one to output a value on that channel, and another to input it. It is a simple exercise using the algebraic laws of Sections 3.1 and 4.2 to show that these two programs are equivalent.

A program which cannot make any progress because all of its subprocesses are waiting, either for an input or an output, is said to be *deadlocked*. It is relatively easy to detect deadlock at run-time and to terminate the program with a suitable error message. Nevertheless, it will often be preferable to try to prove in advance that a given program cannot become deadlocked. A simple example of a program that is guaranteed to reach deadlock is:

```
newChan `bind` \ c → output c "Is anybody there?"
```

The *newChan*, *input* and *output* primitives all work at a fairly low level but can easily be used to build higher level operators. For example, the following two functions can be used to broadcast a copy of a particular message to a list of channels, or to send a list of messages down a single channel:

```
broadcast        ::  a → [Chan a] → IO ()
broadcast m cs   =   parCmds [ output m c | c ← cs ]

outputs          ::  Chan a → [a] → IO ()
outputs c ms     =   seq [ output c m | m ← ms ]
```

Notice that the *broadcast* function outputs to each channel in parallel. On the other hand, the individual output commands used in *outputs* must be executed sequentially because they all use the same channel.

Two more useful commands – this time for allocating a number of new channels and gathering the inputs from a collection of channels (a form of inverse to *broadcast*) – can be defined as follows:

```
newChans     ::  Int → IO [Chan a]
newChans n   =   parList (copy n newChan)

gather       ::  [Chan a] → IO [a]
gather       =   parList . map input
```

where $copy\ n\ x = [\,x \mid i \leftarrow [1..n]\,]$. Both of these functions return a list of values which can be collected in parallel; this is reflected by the use of the *parList* function, defined by:

```
parList :: ParMonad m ⇒ [m a] → m [a]
parList  =  foldr parCons (result [])
  where parCons p ps  =  fork p ps       `bind` \ (x, xs) →
                         result (x : xs)
```

This function is an analogue of the *parl* function in Section 2.3, in much the same way as *fork* corresponds to *par*.

5.2 To Input, or to Output: That is the Question!

Note that, in addition to allowing arbitrary values to be transmitted along a channel, there is no distinction between channels used for input and channels used for output. This is particularly important for the purposes of the type system because it means that we are free to connect an output channel from one process to an input channel of another without any type incompatibility.

In fact, it is quite possible for two processes to communicate with one another using a single process, so long as a suitable protocol is used to ensure that the two processes will not simultaneously attempt to read or write from the same channel. For example, the following code might be used to model two competing bidders at an auction, neither of whom is prepared to let the other win. All of the bids between the two parties are transmitted in both directions using the same channel:

```
auction  =  newChan `bind` \ c →
            let opener  =  output c 100          `bind_`
                           bidder
                bidder  =  input c               `bind` \ yourBid →
                           output c (yourBid + 1) `bind_`
                           bidder
            in  opener <||> bidder
```

The result, of course, is non-termination.[5] While it is safe to use two way communication on a single channel in this particular example, there are many other examples where it is not. For example:

```
bad  =  newChan `bind` \ c →
        input c <||> input c <||> output c 0 <||> output c 1
```

This program violates the condition that a single channel can only accept at most one output request and one input request at any given time. In other words, the subprocesses involved

[5] Or, perhaps, bankruptcy for one of the participants.

in this program may interfere with one another and, as described in Section 4.3, it is not safe to run these programs in parallel.

One way for a programmer to avoid this problem is to adopt the convention that, within a given logical process, each channel is always used either for input or output, but not for both. If this rule is followed, no run-time error can ever occur as a result of two simultaneous input or output requests on a single channel. The *auction* program above does not satisfy this rule but it is very easy to rewrite it so that it does, using two one-way communication channels rather than one two-way channel.

Ideally, we would hope that programs could be checked automatically at compile-time to ensure that this condition is satisfied. However, since channels are first class values, determining whether a program satisfies these conditions is not decidable. Once again, the burden of proof lies with the programmer, as it does in CSP or occam.

(Another solution is to base the communication primitives on something other than a CSP/occam paradigm. In particular, a paradigm which supported unbounded communication requests on individual channels could avoid these problems.)

5.3 Why Not Use Lazy Streams?

Most, if not all, of the examples of parallelism and channel I/O in this paper can be coded safely in terms of lazy streams. For example, something resembling the *auction* program might be written as follows using lazy streams:

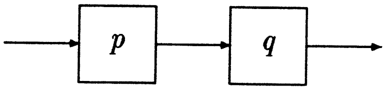
```
let opener               =  100 : beat bidder
    bidder               =  beat opener
    beat (yourBid : bids) =  (yourBid + 1) : beat bids
in  ...
```

Indeed, there are some programs – a buffer with unbounded storage capacity, for example – that can be expressed using lazy streams, but not using the form of channel I/O presented here.

Why then should we be interested in the use of channel I/O? There are three reasons, all of which have been mentioned before, but it is certainly worth summarizing them again here. The first is the interaction with the use of monads. For the "toy" examples considered here, the use of a monad is not essential. In realistic programming examples, it may be considerably more important. The second is an issue of programming style; for some programmers or applications, explicit parallelism and channel I/O may provide a more natural way to describe an algorithm than stream processing. The third reason is that the use of explicit parallelism can provide a compiler with valuable hints (or precise operational semantics in the case that it has been defined for a particular language) about how a given program should be mapped onto a particular parallel architecture.

6 Pipelines

One of the most effective ways to map a program onto a parallel computer system is to split it into a number of separate passes (each of which can be executed on a different processor), with the output from each pass connected to the input of the next. The following diagram illustrates two processes p and q with the output of p connected to the input of q to form a *pipeline*:



Neglecting communication costs, the time taken to produce an output result from a given input value will be the same for both parallel and sequential implementations. However, given a steady stream of input values and assuming that p and q take approximately the same time to map inputs to outputs, a two processor version of the pipeline can calculate twice as many output values as a single processor version in any fixed time period.

Pipelines with exactly one input channel and one output channel can be represented as elements of type:

type *Pipe a b* = *Chan a* → *Chan b* → *IO* ()

This allows the type of values produced on the output channel to be different from those received on the input channel.

The following program is a simple pipeline with only one component that outputs the square of each integer value received on its input channel:

```
squarer        :: Pipe Int Int
squarer ic oc  =  input ic              `bind` \ v →
                  output oc (v * v)     `bind_`
                  squarer ic oc
```

To illustrate the close correspondence between our notation and that of occam, the same program written in the syntax of [11] is:

```
PROC squarer (CHAN ic, CHAN oc) =
  WHILE TRUE
    VAR v :
    SEQ
      ic ? v
      oc ! v * v :
```

Pipelines can be combined using the (≫) operator defined by:

```
(≫)              :: Pipe a b → Pipe b c → Pipe a c
(p ≫ q) ic oc    =  newChan                      `bind` \ mid →
                    p ic mid <||> q mid oc
```

(Of course, the type of values output by *p* must coincide with the type of input values expected by *q*.) This is essentially the same notation as used in CSP in which the λ-abstraction for *mid* corresponds to *hiding* in CSP.

One simple application of (≫) is to implement a process for calculating the fourth power of each input value using two *squarer* processes:

squarer ≫ *squarer*

We can think of the components in a pipeline as functions mapping a sequence of input values to a sequence of output values. Many useful pipelines correspond directly to standard list processing idioms:

```
mapChan             :: (a → b) → Pipe a b
mapChan f ic oc      = input ic               ‘bind‘ \ x →
                       output oc (f x)        ‘bind_‘
                       mapChan f ic oc

filterChan          :: (a → Bool) → Pipe a a
filterChan p ic oc   = input ic               ‘bind‘ \ x →
                       if (p x) then
                           output oc x        ‘bind_‘
                           filterChan p ic oc
                       else
                           filterChan p ic oc
```

With these definitions, the *squarer* function defined above could have been defined as simply *mapChan* ($\backslash x \rightarrow x * x$).

7 The sieve of Eratosthenes

The sieve of Eratosthenes is a standard algorithm for enumerating prime numbers. The basic idea is to count through the list of integers, starting with *2*, the smallest prime, and filter out any values which are multiples of previously discovered primes. Any number which is not a multiple of a previous prime must itself be prime. Many parallel implementations of this algorithm have been provided in the literature; we will provide yet another.

We will split the task of enumerating the list of primes into two separate parallel tasks. The first just counts through the integers starting with *2*, outputting these values on a channel *ints*. The second process inputs values from *ints* and filters out all but the prime numbers which it outputs on a channel *out*:

```
sieve      :: Chan Int → IO ()
sieve out   = newChan ‘bind‘ \ ints →
              outputs ints [2..] <||> pfilter ints out
```

The next problem is to work out how to define the *pfilter* process. One way to do this is to consider a slightly more general case. Consider a process of the form *pfilter c out* and suppose that the values input on channel c are in ascending order with first element p and such that none of the following values are multiples of any smaller prime. (It is easy to verify that these conditions hold for the initial *pfilter* process.) Obviously, the first thing that *pfilter c out* should do is to input the prime p from c and output this value on the *out* channel. Any subsequent values received on c that are multiples of p can be discarded since they are certainly not prime.

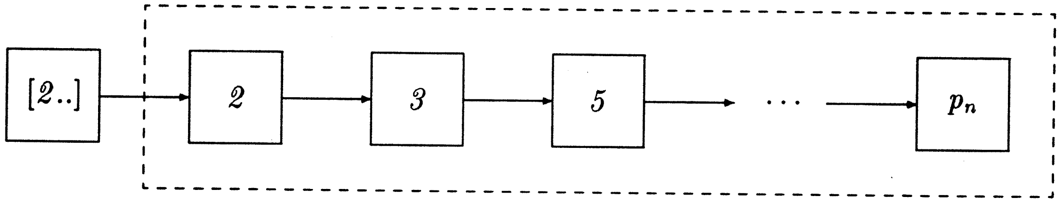
What should we do with the remaining values? The easiest thing is to pass them on to a new *pfilter* process. This leads to the following definition for *pfilter*:

```
pfilter        :: Pipe Int Int
pfilter c out   = input c                       `bind` \p →
                  output out p                  `bind_`
                  newChan                       `bind` \c' →
                    (filterChan (divis p) c c')
                  <||>
                    (pfilter c' out)

divis          :: Int → Int → Bool
divis n m       = m `mod` n ≠ 0
```

Note the use of the *filterChan* function, introduced in the previous section, to describe the task of filtering out the multiples of p from the input channel c.

We can picture the state of the *sieve* process immediately after the nth prime number p_n has been output using the following diagram:



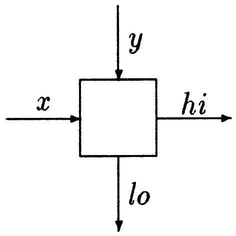
The dashed box on the right represents the original *pfilter* process which has expanded into a pipeline of n processes each filtering out all the multiples of a particular prime number from the stream of integers produced by the process on the left.

This particular example has limited practical applications. But it demonstrates a very powerful technique – the ability to generate new processes as a program executes – which has many practical uses.

8 Parallel Sorting

Sorting algorithms have many practical applications. This section, describes a simple sorting algorithm expressed as a network of parallel processes and suitable for implementation on a parallel computer system.

The sorting algorithm described here uses a network of simple processes called ***comparators***, each of which can be used to sort a pair of input values into the correct order. For the purposes of this section, individual comparators will be illustrated using diagrams of the form:

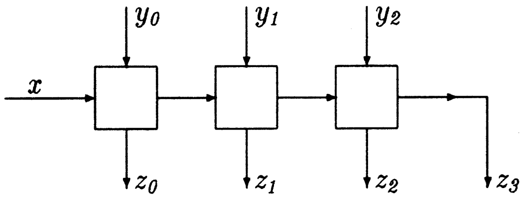


The comparator process reads the values supplied on its two input channels (labeled x and y in the diagram above) and outputs the largest value on the output channel hi and the smaller of the two values on the output channel lo. Formally, a comparator can be defined as:

```
comparator x y lo hi  =  loop
  where loop  =  fork (input x) (input y)  ‘bind‘ \ (u, v) →
                 (output lo (min u v))
                 ‘fork‘
                 (output hi (max u v))   ‘bind_‘
                 loop
```

Notice that the inputs from the channels x and y are performed in parallel; we cannot be certain in which order these inputs will be received so we must be prepared for either possibility to avoid deadlock. A similar argument motivates the decision to output the result values on channels lo and hi in parallel.

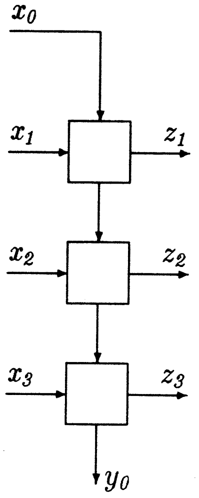
Comparators can be connected in various configurations to build larger components. For example, the following diagram illustrates a way of inserting a value into a sorted list:



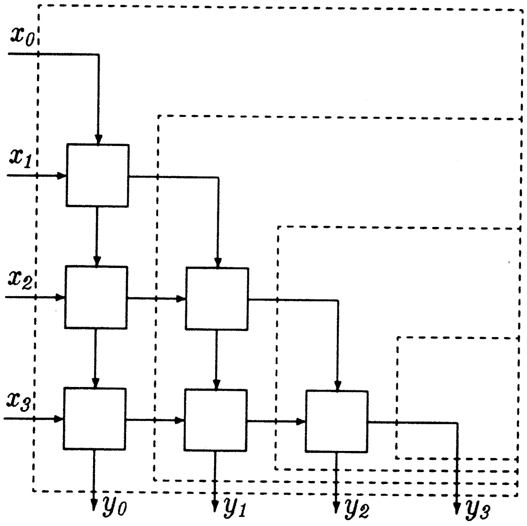
If the values input on channels y_0, y_1 and y_2 are already arranged in ascending order, then the same values will be output on the z_0, z_1, z_2 and z_3, together with the value input on channel

x, inserted at the appropriate point to maintain the correct ordering. Obviously, different numbers of comparators can be used to deal with different numbers of input channels.

Comparators can also be connected in columns as illustrated in the following diagram. The overall effect in this case is to take input values (not necessarily in any particular order) from the channels x_0, x_1, x_2 and x_3, "bubbling" the smallest value out on the channel y_0 with the remaining values output on channels z_1, z_2 and z_3:



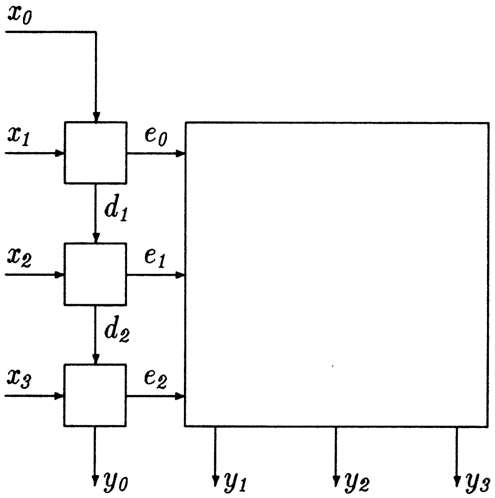
The second smallest value can be produced by passing the values output on channels z_1, z_2 and z_3 into a second column with one less comparator. Continuing in this manner, the complete set of input values can be sorted into ascending order using a network of the form:



(Aside: We have chosen to break up the diagram above into columns. As such, the network of comparators can be thought of as an implementation of the traditional "bubble sort" algorithm. It is worth pointing out that the same diagram can be broken up in a different way as a sequence of rows, each of which implements a simple insertion operation of the kind described above. As such, the network of comparators can also be thought of as an implementation of the "insertion sort'. These two sorting methods have traditionally be considered as distinct algorithms. In truth, the only difference between them is the order in which certain comparisons are made. Expressed in a parallel language where data dependencies alone determine the order of evaluation, we see that they are just different views (i.e. sequentialized versions) of the same process.)

Each of the nested boxes, outlined by lines of dashes, in the diagram above contains a network of comparators for sorting a particular number of input values. And each of these boxes, except in the simplest case where there is only one input, can be broken down into a column of comparators combined with a sorting network for one fewer input values. This pattern suggests one way of describing the construction of a sorting network as a recursive procedure (there are several different ways to do the same thing).

We will describe the construction of the sorting network by a function which takes a list of input channels as its argument and returns both a process to implement the sorting network and the list of channels that the output values are sent to. The most interesting case, illustrated by the diagram below, is when there are several input channels:



Given a list of input channels $(x : xs)$ (with xs non-empty), the first step is to allocate two lists of channels ds and es to be used as the output channels for the *lo* and *hi* outputs respectively of the column of comparators on the left. One d channel and one e channel is needed for each channel in cs. The smallest value input on channels cs will be output on the last channel in ds, written *last ds*. The remaining values can be sorted by constructing a sorting network (represented by the box on the right) using the channels es as input. This description corresponds directly to the following Haskell implementation:

```
sorter        :: Ord a ⇒ [Chan a] → IO (IO (), [Chan a])
sorter [x]    =  result (skip, [c])
sorter (x : xs) = dupChans xs                   `bind` \ ds →
                  dupChans xs                   `bind` \ es →
                  sorter es                     `bind` \ (p, ys) →
                  result (parCmds (zipWith4 comparator xs (x : ds) ds es)
                                <||> p,
                          last ds : ys)
```

The *zipWith4* function is defined in the Haskell standard prelude [9]. It is used here to build a list of comparators with the inputs and outputs taken from corresponding elements in each of the four lists *xs*, $(x : ds)$, *ds* and *es*.

The *dupChans* function allocates a new channel for each value in its argument list:

```
dupChans      :: [a] → IO [Chan b]
dupChans cs   =  parList [ newChan | c ← cs ]
```

An alternative (less efficient) definition is *dupChans cs* = *newChans* (*length cs*).

9 Implementation

In this section we describe an implementation of the *IO* monad on a sequential machine that includes the *fork* function and primitives for channel I/O. This implementation is almost completely written in Haskell and has been used to experiment with the examples in the previous sections. Given the observation that any implementation of *fork* for this monad must be unsafe, at least to some degree, it follows that some parts of the implementation cannot be written in a purely functional language and so must be supplied as primitives. We have tried to keep this set of primitives to a bare minimum; only five primitives are involved. Three of these are used to support reference cells (allocation, update and dereference). The remaining two primitives are used to control process scheduling and have almost trivial implementations in the underlying run-time system (written in C).

9.1 Implementation of the *IO* Monad

One of the most important requirements for a sequential implementation of a parallel language is the ability to switch from one process to another with a minimum of overhead. This is often referred to as *context switching*, and plays a significant role in the choice of a suitable implementation for the *IO* monad.

In their paper [18], Peyton Jones and Wadler concentrate on an implementation of *IO* using state transformers. In concrete terms, their implementation is based on the monad:

$$\begin{array}{lcl}\textbf{type}\ \mathit{IO}\ a & = & \mathit{World} \rightarrow (a, \mathit{World}) \\ \\ \mathit{result}\ x & = & \backslash w \rightarrow (x, w) \\ m\ `\mathit{bind}`\ f & = & \backslash w \rightarrow \textbf{let}\ (x, w') = m\ w\ \textbf{in}\ f\ x\ w'\end{array}$$

The *World* datatype used here represents the complete state of the world at a particular point in time. (These definitions are used to describe the semantics of the monad operations. As we indicated in Section 3.4, there is no need to provide a concrete representation for the *World* type in the actual implementation; it is sufficient to use a dummy token in its place and to update the world in-place. The discipline of passing a dummy token simply ensures that the updates are carried out in the correct order.)

Unfortunately, it is difficult to implement context switching with this "world-passing" implementation of *IO*. Consider, for example, the execution of an *input* command in a context of the form:

$$(\ldots((\mathit{input}\ c\ `\mathit{bind}`\ f_1)\ `\mathit{bind}`\ f_2)\ \ldots\ `\mathit{bind}`\ f_n)$$

The context in this case is the sequence of pending calls to be executed once an input value has been received. In a concrete implementation, this will typically correspond to a sequence of stack frames which must be saved and later restored to implement a context switch.

What we really need is a simple way of capturing the remaining part of the computation to be performed once the input value has been received. This is precisely the role of a *continuation*! With this in mind, we adopt an implementation for *IO* which makes direct use of continuations:

$$\begin{array}{lcl}\textbf{type}\ \mathit{IO}\ a & = & (a \rightarrow \mathit{Ans}) \rightarrow \mathit{Ans} \\ \\ \mathit{bind} & :: & \mathit{IO}\ a \rightarrow (a \rightarrow \mathit{IO}\ b) \rightarrow \mathit{IO}\ b \\ m\ `\mathit{bind}`\ n & = & \backslash k \rightarrow m\ (\backslash a \rightarrow n\ a\ k) \\ \\ \mathit{result} & :: & a \rightarrow \mathit{IO}\ a \\ \mathit{result}\ x & = & \backslash k \rightarrow k\ x\end{array}$$

The answer type, *Ans*, is the type that we assign to a computation and will be discussed in more detail in Section 9.4. Values of type $a \rightarrow \mathit{Ans}$ correspond to continuations. For example, in this framework, a call to the *input* command on a channel $c :: \mathit{Chan\ Int}$ takes the form:

$$\mathit{input}\ c\ k$$

where $k :: \mathit{Int} \rightarrow \mathit{Ans}$ is a continuation that describes the computation to be performed once the input value of type *Int* has been received. In other words, the context for this command is captured directly as one of the arguments of the ***input*** function. It is equally straightforward to restore a captured context once the continuation argument is known. All that we need to do is apply the continuation to its argument.

From a low-level perspective, the main difference between these two implementations of *IO* is that the continuation-based version uses an explicit representation for the continuation of a command as a heap allocated closure, while the world-passing version represents this continuation implicitly as a sequence of stack frames. The continuation-based implementation of *IO* is discussed by Peyton Jones and Wadler in [18]. However, for their particular implementation, and without the need for context switching, the world-passing version seems preferable because it reduces the need for heap allocation.

9.2 Implementation of *fork*

Both the implementation of *fork* in this section and of the channel I/O primitives in the next will be described in terms of some simple primitives for dealing with references. This requires an abstract datatype *Ref a*, the type of references to values of type *a*, and operations for allocation, update and dereferencing:

$$\begin{array}{lll}
\textbf{data}\ \mathit{Ref}\ a & & \\
 & & \\
\mathit{newvar} & :: & \mathit{IO}\ (\mathit{Ref}\ a) \\
\mathit{assign} & :: & \mathit{Ref}\ a \to a \to \mathit{IO}\ () \\
\mathit{deref} & :: & \mathit{Ref}\ a \to \mathit{IO}\ a
\end{array}$$

References have many applications other than those described here and might well have been included in the *IO* monad anyway as general purpose utilities (see [18, 16] for sample applications).

To model parallelism in a sequential implementation, we will assume that the runtime system maintains a set of values (each of type *Ans*) corresponding to suspended (but ready to execute) computations in addition to the current process. We will refer to this as the *task set*. Two further primitives are necessary to control switching between different tasks:

$$\begin{array}{lll}
\mathit{schedule} & :: & \mathit{Ans} \to \mathit{Ans} \to \mathit{Ans} \\
\mathit{resched} & :: & () \to \mathit{Ans}
\end{array}$$

The *schedule* function is used to add processes to the task set suspended computations. For example, *schedule p q* might be implemented by adding *p* to the set and then executing *q*. The *resched* primitive terminates the current process and removes and resumes a previously suspended computation from the task set. A call to *resched* will fail if the task set is empty. However, with the implementations presented below, this is only possible when a program reaches deadlock. This makes it very easy to detect deadlock at run-time and to abort the current program with an appropriate diagnostic message. Both of these functions have simple implementations that can be described in just a few lines of C code.

The dummy parameter which makes *resched* a function with domain type () rather than just a value of type *Ans* arose naturally from a need to use *resched* as a continuation. As it happens, this extra parameter would have been needed anyway to guarantee the correct sequencing to calls of *resched*.

In an attempt to provide a reasonable degree of fairness, it may be appropriate to implement the task set as a queue, or to use a random number generator in the implementation of *resched* to decide which process should be executed next. We will not concern ourselves further with such issues here.

At last! We are finally in a position to describe the implementation of *fork*. First, we introduce a datatype whose values can be used to record the status of an executing process of the form *fork p q*:

```
data Fork a b  =  Running  |  LDone a  |  RDone b.
```

Running represents the situation where neither branch of the *fork* command has terminated. A value of the form *LDone x* indicates that the left process, *p*, has terminated with result *x*. In a similar way, a value of the form *RDone y* is used to indicate that the right process *q* has terminated with result *y*. There is no need to provide a representation for the case where both component processes have terminated because, as soon as this happens, the *fork* process will also terminate.

The first step in the execution of a *fork* command is to allocate a new reference cell, initialized to *Running*. The two subprocesses are then scheduled for execution using special continuations *lDone k v* and *rDone k v* that capture both the reference cell and the original continuation for the *fork* command:

```
fork        ::  IO a -> IO b -> IO (a,b)
fork p q k   =  newvar (\ v ->
                assign v Running (\() ->
                schedule (p (lDone k v))
                         (q (rDone k v))))
```

The continuations involving *lDone* and *rDone* are invoked when the left and right processes (respectively) terminate but it is not possible to predict which of these will be called first. Suppose, for the sake of argument, that the left process terminates first with result *x*. In this case, the continuation simply updates the status reference cell from *Running* to ***LDone x*** and then uses *resched* to continue with the execution of another process. Sometime later, the right process terminates with value *y* and the *rDone* continuation is invoked. Checking the value in the status reference cell reveals that the left process has already terminated and hence execution continues by passing the pair (x, y) to the original continuation for the *fork* process.

Taking the other possibility – that the right process may sometimes terminate before the left – into account, we obtain the following definitions for *lDone* and *rDone*:

```
lDone        :: ((a, b) → Ans) → Ref (Fork a b) → a → Ans
lDone k v a  =  deref v (\f →
                case f of
                  Running   →  assign v (LDone a) resched
                  RDone b   →  k (a, b))

rDone        :: ((a, b) → Ans) → Ref (Fork a b) → b → Ans
rDone k v b  =  deref v (\f →
                case f of
                  Running   →  assign v (RDone b) resched
                  LDone a   →  k (a, b))
```

9.3 Implementation of Channels

One simple way to implement channel input would be for the input process to check the status of a channel and loop until an output value has been sent. Obviously, the input routine would need to be suspended each time before looping so that other processes have an opportunity to make some progress before the channel is examined again. This approach has the disadvantage that we may have to poll the input channel many times before a value arrives. It will also be necessary to keep track of which processes in the task set are suspended waiting for I/O so that we can detect deadlock and not enter an infinite loop.

A much better approach is to store the suspended input process (or rather, the continuation for the input process) in the channel itself. Sometime later, when an output command has been executed, we can move the input process (formed by applying the input continuation to the output value) into the task set, ready for further execution. This avoids the need for repeated polling of the input channel and does not clutter up the task set with processes waiting for input, making it easier to detect deadlock.

It is also possible for an output command to be executed before the corresponding input command. We can use the same basic approach to deal with this situation except that this time we have both an output value and an output continuation to be saved in the channel.

It follows that there are three possible states for a channel: inactive, waiting for an output process or waiting for an input process. The channel status may change between these three alternatives as the program executes. These observations lead us to the following implementation of channels, described using reference cells:

```
type Chan a          =  Ref (ChanStatus a)
data ChanStatus a    =  Inactive
                     |  InReady (a → Ans)
                     |  OutReady a (() → Ans)
```

With this representation, constructing a new channel simply requires allocating a new reference cell and setting the initial channel status to *Inactive*:

```
newChan  :: IO (Chan a)
newChan  =  newvar              `bind` \ c ->
            assign c Inactive   `bind_`
            result c
```

The implementation of *input* is also straightforward. First we examine the current status of the channel. If the channel is inactive then we store the request for input in the channel and use *resched* to switch to a different parallel task. If the channel already contains an output value then we reschedule the execution of both the input and output processes using the appropriate arguments for each continuation and set the channel status back to *Inactive*. Finally, if the channel already contains another input request, then a run-time error occurs.

```
input      :: Chan a -> IO a
input c k  =  deref c (\cs ->
              case cs of
                Inactive      ->  assign c (InReady k) resched
                OutReady v k' ->  schedule (k' ())
                                   (schedule (k v)
                                    (assign c Inactive resched))
                InReady k'    ->  error "simultaneous inputs")
```

The definition of *output* is very similar:

```
output c e k  :: Chan a -> a -> IO ()
output c e k  =  deref c (\cs ->
                 case cs of
                   Inactive      ->  assign c (OutReady e k) resched
                   InReady k'    ->  schedule (k' e)
                                      (schedule (k ())
                                       (assign c Inactive resched))
                   OutReady v k' ->  error "simultaneous outputs")
```

9.4 The Answer Type, *Ans*

The answer type *Ans* was used in the definition of the *IO* monad in Section 9.1 and we have informally described values of this type as representing executable processes. It is actually rather surprising that we have not had to go in to more detail than this to produce the implementation described in the previous sections!

In the interests of completeness, we will end with a description of one possible implementation for the *Ans* type and for the primitives *schedule* and *resched*. The first step is to think of an answer as a function of type:

$$\textbf{type}\ \mathit{Ans} \quad = \quad \mathit{World} \to \mathit{World}.$$

Starting in a world w, the execution of program $a :: \mathit{Ans}$ results in a new world $a\ w$. If the program a has side-effects then the new world will not be the same as the old world. However, because of the sequencing of side-effecting operations enforced by the use of the monad ADT, these side-effects can be implemented directly by updating the world in-place.

What then does the *World* type represent? In theory, it may have many different components including, for example, a mapping from reference cells to values and a representation of the task set. However, as has already been observed, in practice, these individual components can be implemented within the runtime system and it suffices to use a single token as a representation of the *World*.

To describe the implementation of *schedule* and *resched* we will instead use a representation of the world that suppresses all of these different components except the task set. For simplicity, we will represent this using a list:

$$\textbf{data}\ \mathit{World} \quad = \quad \mathit{World}\ [\mathit{Ans}]$$

Given this definition, a process $p :: \mathit{IO}\ a$ can be executed by evaluating the expression:

$$p\ (\backslash a\ w \to w)\ (\mathit{World}\ [\,]).$$

In other words, the process is executed with a continuation, $(\backslash a\ w \to w)$ that ignores the result obtained by the program p and leaves the world unchanged, and starting with an empty task set.

The definitions for *schedule* and *resched* follow directly from the informal descriptions given in Section 9.2:

$$\begin{array}{lcl}
\mathit{resched} & :: & () \to \mathit{Ans} \\
\mathit{resched}\ ()\ (\mathit{World}\ [\,]) & = & \mathit{error}\ \text{``}\mathit{deadlock!}\text{''} \\
\mathit{resched}\ ()\ (\mathit{World}\ (q : qs)) & = & q\ (\mathit{World}\ qs) \\
 & & \\
\mathit{schedule} & :: & \mathit{Ans} \to \mathit{Ans} \to \mathit{Ans} \\
\mathit{schedule}\ p\ q\ (\mathit{World}\ ps) & = & q\ (\mathit{World}\ (ps{+}{+}[p]))
\end{array}$$

10 Conclusions and Future Work

Many of the ideas presented here are still in a preliminary stage and we anticipate that some refinements will be suggested by further work. The main contributions of this report are as follows:

- First, motivated by the observation that programs written using monads or similar techniques often destroy important opportunities for parallel execution, we have proposed the use of a class of monads in which parallelism can be captured explicitly using the *fork* function.

- Second, we have investigated the use of unsafe implementations of *fork* in monads that also support side effects. This places an unavoidable burden of proof on the programmer. On the other hand, it allows parallel algorithms to be expressed more directly and may be useful in the implementation of compilers for parallel machines, guiding the mapping from source programs to particular parallel architectures.

The conflict between safety and parallel execution is very unfortunate. One of the greatest advantages of "pure" functional languages is the ability to reason about programs using simple algebraic laws. If these laws are invalidated by the introduction of unsafe primitives, we may need to reassess our motivations for using such "pure" languages in the first place. On a more positive note, there are still many opportunities for different approaches that may, for example, help to limit the impact of unsafe primitives or even to eliminate them altogether without sacrificing the use of parallelism.

There are a number of areas that would be interesting topics for investigation in further work:

- **Non-determinism**: The reason that *fork* is unsafe in a monad with side-effects is that it introduces an element of non-determinism into the language. For example, we can come fairly close to defining a (monadic) version of McCarthy's *amb* function:

  ```
  amb        ::  IO a → IO a → IO a
  a ‘amb‘ b   =  newvar                          ‘bind‘ \ v →
                 fork (assign v a) (assign v b)  ‘bind_‘
                 deref v
  ```

  Examples like this are well-known sources of difficulty in pure functional languages. Further studies of the use of non-determinism in such languages (such as the work described in [10]) may help us to deal with some of the most significant problems described in this report.

- **Local parallel computation**: With the primitives described in this paper, there is no way for a computation, described in terms of the *IO* monad to be encapsulated as part of a purely functional subprogram. This kind of problem is dealt with in [18] by introducing an unsafe primitive function:

  ```
  delayIO :: IO a → a.
  ```

  The *pure* construct in [16] is closely related to this but again, it is not clear whether *pure* can be implemented safely without imposing significant restrictions on its use (based perhaps on the type system described in [3]). Riecke [20] addresses the same issues using the concept of an *effects delimiter*. Finding a satisfactory way to deal with these problems in practical work may be possible, but it is unlikely to be easy – the task of determining whether the use of one of these constructs in a particular situation is safe is not decidable.

- **Formal semantics**: Despite a fairly precise presentation of the implementation of *fork*, we have often relied rather heavily on our intuitions about parallelism and less so on any formal semantics. Such a semantics would be useful in giving a proper description of the proof obligations needed to justify the use of unsafe primitives. It would also be useful to validate some of the algebraic laws for the monad operators, as described in a number of places in this report. Related work in this area includes [2, 17].

References

[1] L. Augustsson. Implementing Haskell overloading. In *Proceedings of the 6th ACM conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, ACM Press, June 1993.

[2] D. Berry, R. Milner and D.N. Turner. A semantics for ML concurrency primitives. In *19th Annual Symposium on Principles of Programming Languages*, Santa Fe, New Mexico, January 1992.

[3] K. Chen and M. Odersky. A type system for a lambda calculus with assignments. Yale University, Department of Computer Science, Research Report YALEU/DCS/RR-963, May 1993.

[4] J.H. Fasel. Some examples of monadic concurrency (preliminary summary). Draft, Los Alamos National Laboratory, March 1993.

[5] C.A.R. Hoare. Communicating Sequential Processes. Prentice-Hall, 1985.

[6] P. Hudak. Arrays, non-determinism, side-effects, and parallelism: a functional perspective. In *Proceedings of the Santa Fe Graph Reduction Workshop*, Los Alamos/MCC, Springer-Verlag LNCS 279, October 1986.

[7] P. Hudak. Para-functional programming in Haskell. In *Parallel functional languages and compilers*, ACM Press (New York) and Addison-Wesley (Reading), 1991.

[8] P. Hudak. Mutable abstract datatypes – or – How to have your state and munge it too. Yale University, Department of Computer Science, Research Report YALEU/DCS/RR-914, December 1992, revised May 1993.

[9] P. Hudak, S.L. Peyton Jones and P. Wadler (eds.). Report on the programming language Haskell, version 1.2. *ACM SIGPLAN notices*, 27, 5, May 1992.

[10] J. Hughes and A. Moran. A semantics for locally bottom-avoiding choice. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, Ayr, Scotland, July 1992. Springer Verlag Workshops in computing series.

[11] INMOS Limited. *occam Programming Manual.* Prentice Hall, 1984.

[12] INMOS Limited. *occam 2 Reference Manual.* Prentice Hall, 1988.

[13] M.P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proceedings of the 6th ACM conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, ACM Press, June 1993.

[14] J. Launchbury. Lazy imperative programming. In *Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages*, Copenhagen, Denmark, June 12, 1993. Yale University Research Report YALEU/DCS/RR-968.

[15] P.H. Mills, L.S. Nyland, J.F. Prins and J.H. Reif. Prototyping N-body simulation in Proteus. In *Proceedings of the Sixth International Parallel Processing Symposium*, Beverley Hills, California, IEEE, March 1992.

[16] M. Odersky, D. Rabin and P. Hudak. Call by name, assignment and the lambda calculus. In *20th Annual Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993.

[17] A. Ohori and K. Kato. Semantics for communication primitives in a polymorphic language. In *20th Annual Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993.

[18] S.L. Peyton Jones and P. Wadler. Imperative functional programming. In *20th Annual Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993.

[19] J.F. Prins and D.W. Palmer. Transforming high-level data-parallel programs into vector operations. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, California, May 1993. ACM SIGPLAN Notices, volume 28, number 7.

[20] J.G. Riecke. Delimiting the scope of effects. In *Proceedings of the 6th ACM conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, ACM Press, June 1993.

[21] M. Rittri. Private communication.

[22] P. Wadler. Comprehending Monads. *ACM conference on LISP and Functional Programming*, Nice, France, June 1990.

[23] P. Wadler. The essence of functional programming. In *19th Annual Symposium on Principles of Programming Languages*, Santa Fe, New Mexico, January 1992.

[24] P. Wadler. Monads and composable continuations. Manuscript, University of Glasgow, November 1992.