Functional Programming for Concurrent and Distributed Computing*

F. W. BURTON
Department of Computer Science, 3160 MEB, University of Utah, Salt Lake City, Utah 84112

There are at least two approaches to the design of languages for parallel computing. One approach is to use functional or relational languages which are easy to read, write, transform and verify. The more conventional approach is to use procedural languages which give a programmer a high degree of control over the run-time behaviour of a program. There is a need to reconcile these two approaches in a language which permits both simplicity and efficiency.

We propose a small and simple set of annotations (or pragmas) to control the run-time behaviour of a functional program. The annotations allow a programmer to use three forms of parameter passing. The parameter-passing mechanisms correspond to passing by name, value and need in a sequential language. In addition, in a distributed system a programmer can specify that work should be done on the current processor, an arbitrary processor, or a particular processor such as the one containing a specific data item.

The annotations cannot affect the meaning (result) of a functional program, except for causing non-termination in some cases (which we view as an extreme form of inefficiency). This separation of meaning from control allows a program to be both simple and efficient.

Since non-determinism appears to be unavoidable without significant loss of efficiency in a concurrent system, the interaction of the proposed annotations with non-determinism is briefly considered.

The run-time behaviour of an annotated functional program is similar to that of procedural programs using message passing, semaphores or rendezvous to control communication and synchronisation.

Received January 1986, revised July 1986

1. INTRODUCTION

Functional languages have simple semantics.[50] Functional programs are easier than imperative programs to manipulate in program transformations[22] and correctness proofs.[51] In addition, functional programs can be simpler to write, understand and maintain.[7, 41, 57] Functional programs, written in LISP and similar languages, have been used by researchers working in the area of artificial intelligence for many years. As software costs increasingly dominate hardware costs, the case for using functional languages is becoming stronger.

With the advent of VLSI, large-scale concurrency is becoming increasingly attractive. One approach to concurrency is to introduce user-defined tasks and features such as semaphores, messages or rendezvous into a language so that concurrency can be explicitly controlled. Another approach is to let computer systems discover opportunities for parallelism in-high level functional or relational languages.

The advocates of the first approach use the need for efficiency to make the case for explicit programmer control of parallelism. Advocates of the second approach argue that concurrent computation is even more complex than sequential computation, so the difficulty of producing reliable software is greatly increased when explicit control of concurrency is involved.

Both cases are strong. It has been argued that as computing costs drop, the need for efficiency will also drop. (It was also argued in the 1940s that the world would never need more than a few computers, and in the 1950s that nuclear power would make electricity so cheap that it would not be necessary to meter it.) Certainly, efficiency will become less critical in many existing applications. However, as computing costs drop, larger problems can be solved. Applications which are not being considered today will be possible in the future. With large problems, asymptotic efficiency will become more important. With many problems, including NP-hard problems, we can never have as much computing power as we would like. However, efficient use of increasing computational power will make it easier to solve larger problems exactly and to produce better approximations to problems which cannot be solved exactly.

One way to reconcile these two approaches is to provide annotations in a functional language,[10, 40, 48] so that the logic of the program is largely separated from the control of its evaluation. We will examine a set of annotations to control the distributed evaluation of a functional program on a network of processors. These annotations will more or less include those reported in Ref. 10, and additional annotations to control communication, synchronisation and placement of work. With these annotations it is possible to produce a collection of communicating processes. The annotations will not obscure the clarity of an unannotated functional program nor alter its semantics, but may alter the elapsed time, total work, amount of storage and amount of communication required to evaluate the program. (In cases where a program would terminate under normal order reduction, we will regard non-termination as an extreme form of inefficiency.) With reasonable defaults, the annotations will be required in only a few unusual situations. In these cases, annotations can often be packaged in library functions (e.g. to buffer lists for consumer/producer parallelism).

The proposed annotations are suitable for a broad class of parallel machines, but are primarily intended for a system of loosely coupled processors such as the Cosmic

* This material is based upon work supported by the National Science Foundation under grant no. ECS-8312748 and grant no. DMC-8514946.

Cube,[49] viewed as a virtual tree machine.[11] However, annotated functional programs could also run on data-flow[2, 23, 30] and similar machines,[21] and most other parallel machines. With some of these machines not all the annotations would be meaningful.

This work compliments the work of others, which is directed more at the constant factor sources of inefficiency in functional languages. For example, various researchers are looking at the problem of efficiently compiling functional programs on conventional machines.[5, 15, 35, 36, 42, 52]. Other researchers are exploring the use of more appropriate computer architectures for sequential functional language evaluation [17, 47, 53, 58] and parallel functional language evaluation[2, 3, 21, 23, 28, 30, 44, 54]. This work also compliments work by others concerned with other aspects of functional language design, such as data typing and modularity.[7, 59].

In Section 2 we shall consider the sequential evaluation of functional programs. This will be extended to parallel evaluation in Section 3 and distributed evaluation in Section 4. The approach in these sections will be informal, with various examples. Non-determinism will be considered in Section 5. A more formal approach to the material in Sections 2 to 5 can be found in Section 6. Implementation considerations are briefly discussed in Section 7. Section 8 is the conclusion.

2. SEQUENTIAL EVALUATION

2.1. Reduction order

Call by need (or lazy evaluation)[4, 25, 32, 39, 55, 60] has been advocated as an optimal evaluation strategy for functional programs on a sequential processor. Except for some special cases with higher-order functions (which are not a problem, because these cases can be avoided without serious inconvenience) and a constant factor overhead (which can often be eliminated by a compiler optimisation,[46] call by need is optimal with respect to evaluation time. Unfortunately, call by need is not optimal with respect to space.

For example, suppose tail recursion can run in constant space and that storage can be reclaimed as soon as it is no longer referenced. In this case the function *factorial* given in Fig. 1 will run in constant space if parameters are passed by value. (The function definition symbol is ' < = ' and the boolean test for equality is ' = '.) On the other hand, if call by need is used the evaluation of *f* will not result in the first argument, *i*, being evaluated until it is required at the bottom level of recursion. Call by need will require O(*n*) storage.

There are many examples of programs which will run in constant space if call by value parameter passing is used, but will require storage proportional to their execution times if call by need is used. As David Turner has noted, ' In a language with lazy evaluation it is in general rather difficult either to predict or to control the space behaviour of programs'.[59]

There are times when call by value is not the best strategy. For example, consider the list-processing functions in Fig. 2. (The primitive list-processing functions *first*, *rest* and *cons* satisfy the usual list axioms:

first(*cons*(*a*, *b*)) < = *a*
rest(*cons*(*a*, *b*)) < = *b*.)

```
factorial(n) < = f(1, n)
  where f(i, m) < = if m = 0 then i else f(m * i, m − 1)
```

Figure 1. A function to compute *n* factorial.

```
count(i, j) < =
  if i = j then nil
  else cons(i, count(i + 1, j))
map(f, a) < =
  if a = nil then nil
  else cons(f(first(a)), map(f, rest(a)))
sum(a) < = tailsum(0, a)
  where tailsum(n, a) < =
    if a = nil then n
    else tailsum(n + first(a), rest(a))
```

Fig. 2. Useful list-processing functions.

One way to compute

$f(1)+f(2)+\ldots+f(n)$

is by using the expression *sum*(*map*(*f*, *count*(1, *n*))). However, if parameter passing is by value, then lists of O(*n*) length are required to store intermediate results. On the other hand, if *cons* is lazy, so cons(a, b) will return a result without evaluating *a* or *b*, then this computation will run in constant space. For example, the evaluation of *sum*(*map*(*square*, *count*(1, 3))) will proceed as follows (with some steps skipped):

```
sum(map(square, count(1, 3)))
< = sum(map(square(cons(1, count(1 + 1, 3)))))
< = sum(cons(square(1), map(square, count(1 + 1, 3))))
< = tailsum(0, cons(square(1), map(square,
      count(1 + 1, 3))))
< = tailsum(0 + square(1), map(square, count(1 + 1, 3)))
< = tailsum(1, map(square, cons(2, count(2 + 1, 3))))
< = tailsum(1, cons(square(2), map(square,
      count(2 + 1, 3))))
< = tailsum(1 + square(2), map(square, count(2 + 1, 3)))
< = tailsum(5, map(square, cons(3, count(3 + 1, 3))))
< = tailsum(5, cons(square(3), map(square,
      count(3 + 1, 3))))
< = tailsum(5 + square(3), map(square, count(3 + 1, 3)))
< = tailsum(14, map(square, nil))
< = tailsum(14, nil)
< = 14.
```

If call by need were used, instead of call by value with lazy data constructors, then we would have had the same problem as in our previous example.

Hope[7] provides the programmer with both a lazy *cons* and an ordinary (eager) *cons*. We shall take all data constructors to be lazy and assume for the moment that call by value is the standard parameter-passing mechanism. From now on we shall use ':' as a right-associative lazy *cons* operator. Eager pseudo-constructors can easily be written in terms of the lazy constructors. For example, the following function:

eager_cons(*a*, *b*) < = *a*:*b*

acts like an eager list constructor, since its arguments are passed by value.

One further point needs to be clarified here. Each time *first*(a:b) is computed, a is re-valuated. That is, graph reduction is not used. This enables us to discard and recompute a large object, when desired, to save space. For example, consider *f*(a) < = *prod*(a)/*sum*(a), where *prod* is a tail-recursive function to compute the product of the elements in a list. Now *f*(*count*(1, n)) will run in constant space, but will generate the list *count*(1, n) twice (except for the first element which is generated only once). On the other hand, if we wanted to compute *f*(*map*(g, *count*(1, n)) without recomputing the (possibly expensive to compute) list map (g, *count*(1, n)) we can do so by computing

```
f(expand(map(g, count(1, n))))
  where expand(a) < =
    if a = nil then nil
    else eager_cons(first(a), expand(rest(a))).
```

This gives the programmer explicit control over time–space trade-offs.

It is useful to have parameter passing by name as well as value available. (In fact, in Ref. 10 lazy data constructors are considered to be specific cases of higher-order functions having parameters passed by name.) Many of the disadvantages of call by name found in procedural languages such as Algol do not apply to a functional language, where the value of an expression can never change. In a sequential functional language system we have found no need for call by need if both call by name and call by value are provided. If call by need and call by value were provided it would not be possible to discard and recompute expressions such as the one considered above.

We shall take call by value to be the default parameter-passing mechanism for parameters which are not functions, and call by name to be the default for functions. (See Section 6.1.2 for a discussion of the consequences of passing a function by name.) Parameters may be annotated with **name** or **value** to override the default. For example, we can define

```
conditional_or(a, name b) < =
  if a = true then true else b.
```

Similarly we can define partially eager pseudo-constructors such as

```
left_eager_cons(a, name b) < = a:b
```

which will evaluate its first argument but not its second.

(An alternative approach would be to make call by value the default only in cases where a function could be shown to be strict.[38, 43, 46] This would preserve normal order semantics.)

A more detailed discussion of the mixing of call by name and call by value may be found in Ref. 10.

2.2. Calls by opportunity

Schwarz has suggested call by opportunity[48] as a method for evaluating applications of functions such a *f*(a) < = *prod*(a)/*sum*(a) in constant space without re-evaluating list elements. Basically, with call by opportunity, *sum* and *prod* run concurrently. Each time a list element is computed, both consumers consume it before the next element is generated.

This method will fail in the case of two producers and two consumers where the consumers consume in an incompatible manner (For example, one consumer might consume the first-produced list more quickly than the second, while the other consumer might do the opposite.) This forces the consumers out of synchronisation, since they cannot synchronise with both producers at the same time.

Hughes has proposed annotations for a call by need interpreter to solve the same problem in much the same way, but at a lower level.[40] Basically, with Hughes' method an annotated producer is not allowed to produce a value until it has been demanded twice.

In most cases it is easy to transform (as opposed to annotate) a program to achieve the effect of call by opportunity if the evaluation mechanism discussed above is used. For example, we can rewrite f as follows:

```
f(a) < = tail_f(a, 1, 0)
  where tail_f(a, p, s) < =
    if a = nil then p/s
    else tail_f(rest(a), p*x, s+x)
      where x < = first(a)
```

This solution will not work with a call by need evaluator, since the evaluation of p and s will not be done until p/s is computed.

This particular transformation follows a common pattern which can largely be captured in a higher-order function. Let us first consider the simpler function *reduce* shown in Fig. 3. We could redefine *sum* as

sum(a) < = *reduce*(*plus*, 0, a)

and still get the same constant space behaviour as with the definition of *sum* in Fig. 2. (The function *plus* is the prefix form of the infix '+' operator.) Similarly,

prod(a) < = *reduce*(*times*, 1, a)

defines the function *prod* whose existence was assumed above. An efficient function for reversing a list is

reverse(a) < = *reduce*(*eager_cons*, *nil*, a).

Use of functions such as *reduce* to encapsulate common forms of recursion has been widely advocated.[6, 7, 41]

Suppose we wish to compute:

x_1 < = *reduce*(g_1, n_1, a),

x_2 < = *reduce*(g_2, n_2, a),

x_3 < = *reduce*(g_2, n_3, a),

x_4 < = *reduce*(g_4, n_4, a).

Using *multiple_reduce* which is defined in Fig. 3, we could perform these four reductions with call by opportunity space requirements by computing

x_1:x_2:x_3:x_4 *nil* < = *multiple_reduce*(g_1:g_2:g_3:g_4:*nil*, n_1:n_2:n_3:n_4:*nil*, a).

(We assume that the x's are each given the value of the corresponding element of the structure returned by *multiple_reduce*). We can now redefine f by

```
f(a) < = p/s
  where p:s:nil < =
    multiple_reduce(prod:sum:nil, 1: 0: nil, a).
```

```
reduce(f, n, a) <=
    if a = nil then n
    else reduce(f, f(first(a), n), rest(a))
multiple_reduce(f_list, n_list, a) <=
    if a = nil then n_list
    else multiple_reduce(f_list, funmap(f_list, n_list,
first(a)), rest(a))
        where funmap(f_list, n_list, x) <=
            if f_list = nil then nil
            else eager_cons((first(f_list))(x, first(n_list)),
                funmap(rest(f_list), rest(n_list), x))
```

Figure 3. More list-processing functions.

```
factorial(n) <= prod(1, n)
    where prod(i, j) <=
        if i = j then i
        else prod(i, mid) * prod(mid + 1, j)
            where mid <= (i+j) div 2
```

Fig. 4. A parallel factorial function.

This particular transformation will not work in every case where the use of call by opportunity might be desirable, but it will work with many. In other cases, problem-specific transformations may be required. Annotations cannot replace program transformation. (An annotated bubble sort will never be a quicksort.) However, the use of annotations make transformations such as the one above possible.

3. PARALLEL EVALUATION

3.1. Demand-driven evaluation

One of the primary advantages of functional programs is that they have a high potential for parallelism. The absence of side-effects means that independent sub-expressions may be evaluated in any order, or in parallel.

In Ref. 10 it was proposed that a function should be able to evaluate all its value parameters in parallel. For example, Fig. 4 shows a divide and conquer parallel factorial function. If both arguments of the infix multiply function are evaluated in parallel, then the evaluation of *factorial*(n) will result in a binary tree of tasks of depth O(log n) so, with enough processors, *factorial*(n) can be computed in O(log n) time. This same approach can be used to obtain a high degree of parallelism in many other algorithms.

The annotations given in the previous section can be used to inhibit parallelism when desired. For example, suppose we wish to compute $f(x) * g(y)$ by evaluating $f(x)$ before $g(y)$. (This might be useful if both computations produce large intermediate results.) We can force sequentiality by writing

sequential_times($f(x)$, $g(y)$)

where *sequential_times*(u, **name** v) $<= u * v$.

This forces $f(x)$ to be evaluated before the body of *sequential_times* is evaluated, since u is passed by value, and prohibits $g(y)$ from being evaluated at the same time, since v is passed by name.

3.2. Data-driven evaluation

Additional parallelism is possible if the evaluation of $f(x)$ can start before the evaluation of x has finished. For example, consider the evaluation of the expression

$$f(g(x), g(y), g(z)) \textbf{ where } f(a, b, c) <= h(a, b) + h(b, c) + h(a, c)$$

on two processors. Once any two of the parameters to f have been evaluated, the evaluation of one of the applications of h may be initiated. The data-flow or data-driven approach to parallelism[54] permits any computation to proceed as soon as the necessary data is available.

This second form of parallelism seems to be particularly important in supporting consumer/producer parallelism. For example, in computing *sum*(*map*(*square*, *count*(1, n))), *sum*, *map* and *count* may all run in parallel. On the other hand, increased storage requirements come with the increased parallelism. For example, if *count* gets too far ahead of *sum* then O(n) storage may be required to compute *sum*(*map*(*square*, *count*(1, n))). Concurrent programmers using procedural languages have used bounded buffers for years in order to combine modest storage requirements with consumer/producer parallelism. What is required is some means of controlling synchronisation.

In a procedural programming language, incorrect synchronisation will often lead to an incorrect result. In a functional language, annotations controlling synchronisation may affect time and space requirements, but should not affect the eventual result produced by a program.

Our solution is to have one form of parameter passing that requires the evaluation of an argument to terminate before the function is applied, and another form that supports data-driven evaluation. As in Ref. 10, the default call by value parameter-passing mechanism will not initiate the evaluation of a function application until after the evaluations of all value parameters have terminated. We shall use the key word **speculation** to denote data-driven parameter passing. We shall associate call-by-name semantics with this new mechanism. For example,

funny_or(x, y)

where *funny_or*(a, **speculation** b) <= **if** a **then** *true* **else** b

will always return the result *true* whatever the value of x is *true*, even if the evaluation of y fails to terminate. On the other hand, if there are enough processors, x and y may be evaluated in parallel.

The use of functions such as *funny_or* allows us to achieve the effect of OR parallelism[18, 19] in a functional language. We can speculate that certain results are likely to be required, to take full advantage of available processors. If the results are required we win, and speed the computation. If the results are not required we lose, and may do some unnecessary work. We should expect that speculative work would not be done unless either there are processors that would otherwise be idle or the work is found to be necessary (e.g. after x has been found to be *false*). The use of priorities, as proposed in Refs. 12 and 13, would be useful in controlling which of several speculative computations should be run in preference to others in situations where there are enough processors to run some but not all.

Call by speculation is basically a parallel version of call by need that allows a parameter to be evaluated before it is needed if there are enough processors available. The **speculation** is similar to the *FUTURE* annotation of Multilist[31] and *EAGER* in Ref. 27. However, call by speculation preserves call by name semantics without requiring a 'fair' scheduler which would tend to result in excessive storage requirements[8, 11]

3.3. Buffers

As mentioned earlier, data constructors are lazy. For example, neither a nor b will be evaluated when $a{:}b$ is evaluated. We will allow data structures to contain expressions in the process of being evaluated, as well as unevaluated expressions. We can now define

speculative_cons(**speculation** *a*, **speculation** *b*) < = *a*: *b*.

When *speculative_cons*(*a*, *b*) is evaluated, the evaluations of a and b will be initiated, but a result may be returned before either of these evaluations terminates. However, if *first*(*speculative_cons*(*a*, *b*)) is evaluated, no result may be returned until the evaluation of a has terminated (as would also be the case if *first*(*a*: *b*) were evaluated), and similarly for *rest* and b.

Let us take a minute to review the three types of list (pseudo-)constructors defined so far. The lazy constructor, ':', will return a result immediately. The result will contain unevaluated expressions as components unless its arguments have been previously evaluated. The eager list pseudo-constructor, *eager_cons*, will have evaluated its arguments and will return a structure with evaluated components only after their evaluations terminate. Finally, *speculative_cons* will initiate the evaluation of its components but will return a result without waiting for the evaluations to terminate. This final behaviour will form the basis of the implementation of buffers.

To implement buffers, we really need

buffer_cons(**speculation** *a*, **name** *b*) < = *a*: *b*

which initiates the evaluation of its first argument but not its second (rather like the *left_eager_cons* considered earlier). Fig. 5 gives the implementation of a function, *buffer*, which will transform a lazy list into a buffered list. If x is a lazy list, then *buffer*(x) will initiate the evaluation of the first item in the list, but will not do anything to the remainder of the list. However, computing *rest*(*buffer*(x)) will cause *buffer*(*rest*(x)) to be computed. This will initiate the evaluation of the second item in the list. We note that when buffers are used, we speculate that the next item in the buffer will eventually be required.

```
buffer(a) < =
  if a = nil then nil
  else buffer_cons(first(a), buffer(rest(a)))
    where buffer_cons(speculation x, name b) < = x:b
```

Figure 5. A list-buffering function.

To get buffering, and hence consumer/producer parallelism, in an expression such as *reduce*(*f*, *i*, *map*(*g*, *a*)) it is necessary only to change it to *reduce*(*f*, *i*, *buffer* (*map*(*g*, *a*))). We may regard *buffer*(...) as a user-defined annotation.

The approach can easily be generalised to buffers of size greater than one.

Data structures that can be partially used while they are still being constructed have been advocated by others.[1, 24] In data-flow computing, lists (which are often called streams) are often implemented in this way. Streams may be the only data structure permitting this type of pipelined, or consumer/producer, parallelism. We are pleased to have a single mechanism which will degenerate to call by need on a single processor and will permit buffering on a parallel system. If data constructors are viewed as higher-order functions as in Ref. 10, we do not even need to specify that a list constructor can return a result containing values still in the process of being evaluated. This follows directly from the higher-order function defining the list constructor. Of course, we would expect list constructors to be implemented as primitives, as an optimisation for efficiency. We would also expect that a standard library would contain functions such as *buffer*, so that a programmer would rarely need to use the **speculation** annotation directly.

3.4. Arbitrary process communication

In procedural programming languages, arbitrary communication topologies may be constructed with processes communicating via buffered channels. In a functional language, arbitrary communication topologies may be constructed by using mutual recursion, with processes replaced by functions and channels replaced by buffered lists.

Let us suppose that process P sends messages to Q and R using channels *P_to_Q* and *P_to_R* and returns a result for the overall computation on the channel *RESULT*. Similarly, processes Q and R send messages on channels *Q_to_P*, *Q_to_R*, *R_to_P* and *R_to_Q*.

The function, p, corresponding to process P, may return a list of three lists as a result. These will correspond to the three output channels used by P. Let us suppose that the first list in p's result is the overall result, the second list corresponds to the channel to Q, and the third list corresponds to the channel to R. To make things more readable, we can define the following functions to select the appropriate components:

```
result(a) < = first(a),
p_to_q(a) < = first(rest(a)),
p_to_r(a) < = first(rest(rest(a))).
```

Functions q and r will correspond to processes Q and R respectively. These will also return lists of lists, like p. We can now describe this computation functionally by the expression:

```
result(a)
    where a < = p(q_to_p(b), r_to_p(c))
    and b < = q(p_to_q(a), r_to_q(c))
    and c < = r(p_to_r(a), q_to_r(b)).
```

4. DISTRIBUTED EVALUATION

When evaluating a functional program on a distributed system, both data and work must be sensibly located if the time spent on communication is not to dominate the time spent on computation. An example is given in Ref. 10 of an $O(n)$ time-sequential algorithm which requires $O(n^2)$ communication when inappropriately evaluated on a network of processors. In general, it is not possible to determine automatically when work may be sent advantageously to another processor.

There are basically two types of distributed systems that should be considered. The first type of system consists of a number of identical nodes (e.g. the cosmic cube).[49] Some nodes may be closer to a given node than others, or all nodes may be equally accessible from any given node. With this type of system, multiple processors are used to increase the raw computing power of the system. The programmer does not consider which processor does what, but may require that certain sets of computations be performed on the same processor in order to reduce communication costs. The second type of system consists of a number of nodes serving different purposes. This type of system might be embedded in a physical system requiring real-time control; the different nodes might be attached to different external sources of input, for example. The programmer may require that certain computations be performed on particular processors. Both types of systems can be handed within a common framework. Two annotations are required.

One annotation, **anywhere**, indicates that a sub-expression may be evaluated on an arbitrary processor to be chosen by the language implementation. In effect, whenever a programmer indicates that an expression may be evaluated on an arbitrary processor, a new virtual processor is created. The system is responsible for mapping virtual processors on to physical processors. This permits work to be distributed among the available processors without the programmer knowing how many processors are available or knowing how they are configured. Of course, in the degenerate case of a single processor, all work is performed on that processor.

In a shared-memory system, the **anywhere** annotation would have no significance. Parallel tasks can result from evaluating **value** parameters in parallel. However, a system should not be expected to decide when it is desirable to ship a task to another processor. It is easy to construct examples where communication costs will dominate computation costs if the wrong decision is made.[10] We have found it convenient to annotate expressions. However, the **anywhere** annotation could be given another location in the syntax without changing the power of the notation.

Fig. 6 shows the factorial program from Fig. 4 annotated to run on a network of processors. (It may be a bit silly to use more than one processor to compute *factorial*(n) for any n that will not cause overflow. However, this approach will work with many divide and conquer algorithms. We prefer to illustrate the approach with a simple example.)

The other annotation, **at**, specifies that a sub-expression must be evaluated at the node where a particular data item is located. This makes it possible to move work to where large data structures reside. In the case of a system of processors of the second type

```
factorial(n) < = prod(1, n)
   where prod(i, j) < =
     if i = j then i
     else (prod(i, mid)anywhere) * (prod
(mid + 1, j)anywhere)
        where mid < = (i + j) div 2
```

Figure 6. A distributed factorial function.

described above, a dummy data item, of type **void**, may be associated with each processor. By requiring that work be performed where a particular data item is located, it is possible to control directly which processor does what. For example, if *processor_8b* is a data item of type **void** that is located on a particular processor (and may be passed to the program as a parameter) then *preprocess* (*input_stream_8b*) **at** *processor_8b* will compute *preprocess*(*input_stream_8b*) on the desired processor.

We shall call the computation involved in evaluating an argument passed by value or speculation an *evaluation*. Clearly an evaluation may include sub-evaluation. For example, the evaluation of $b*b-4*a*c$ will include sub-evaluations of $b*b$, $4*a*c$ and $4*a$. An evaluation excluding all sub-evaluations will be called a *task*. (We do not mean to imply anything about the granularity of parallelism in an implementation. We are using the term 'task' to describe a conceptual unit of work. A physical task in an implementation may correspond to many conceptual tasks.) For example, the task evaluating $b*b-4*a*c$ would conceptually produce two new tasks to evaluate $b*b$ and $4*a*c$. When both of these tasks had terminated, the original task would compute the difference between their results, report its result to its parent, and then terminate.

Each task will run on a particular processor. Usually this will be the processor where the task's parent is running. When a task terminates, it will leave its result on the processor where it ran, so each data item will be associated with a particular processor. The only tasks which will not run on the same processor as their parents are tasks evaluating expressions annotated with **anywhere** or **at**. The expression

e **anywhere**,

where *e* is any expression, may run on any processor. This gives the system the freedom to distribute work to other processors. The expression

e **at** e_1,

where *e* and e_1 are any expressions, will evaluate *e* on the processor on which e_1 resides. When a parameter is passed by name it is not moved. Both *e* **anywhere** and *e* **at** e_1 have the value *e*, so '**anywhere**' and '**at** e_1' may both be considered annotations which do not change the meaning of a program.

In Section 3 we saw how consumer/producer parallelism could be performed using the expression

reduce(*f*, *i*, *buffer*(*map*(*g*, *a*))).

We will generalise this example to provide for distributed consumer/producer parallelism. To perform all applications of *f* on one processor while allowing all applications of *g* to be performed on another, we write

pipe_reduce(*f*, *i*, *buffer*(*map*(*g*, *a*)))

where

```
pipe_reduce(f, n, a) <=
    if a = nil then n
    else pipe_reduce(f, f(first(a), n), (rest(a) at a)).
```

One of the advantages to *buffer*, introduced in Section 3.3, was that it encapsulated all the details of buffering in a function that can be viewed as a non-primitive annotation (since *buffer*(*x*) differs from *x* only in terms of evaluation strategy). We can do the same with distributed consumer/producer parallelism. The function

```
pipe(a) <= fix(buffer(a) anywhere)
    where fix(a) <= first(a):fix(rest(a) at a)
```

will do the job. Notice that the parameter to *fix* at the highest level of recursion is placed on an arbitrary processor, and that the **at** keeps parameters to *fix* on the same processor at lower levels of recursion. When we compute *rest*(*pipe*(*a*)) the computation proceeds as follows:

```
<= rest(fix(buffer(a)))
<= rest(fix(buffer_cons(first(a), buffer(rest(a)))))
<= rest(...:fix(buffer(rest(a))))
<= fix(buffer(rest(a))).
```

since the argument to *fix* is always evaluated on the virtual processor created by the **anywhere**, and since *buffer* always initiates the speculative evaluation of the next list element, all elements of a piped list are evaluated on the new virtual processor. We can now get the desired distributed consumer/producer parallelism by writing

reduce(*f*, *i*, *pipe*(*map*(*g*, *a*))).

A distributed network of processes communicating via message passing can be established using *pipe* and the approach outlined in Section 3.4. The **at** and **anywhere** annotations may also be used to implement a rendezvous like communication and synchronisation structure. We will think of *b* as the *state* of a process with which we want to rendezvous. Let us start by considering the expression *g*(*a*, *b*) **at** *b*. This will send a copy of *a* to the processor on which *b* is located, and compute the value of *g*(*a*, *b*) there. Let us now assume that *g* returns a pair of values. One value is the 'result' of the rendezvous and the other is a 'new state' of the process. (To avoid introducing new notation, we will use ':' as a pairing operation as well as a list operation). The function *g* could have the form

```
g(data, oldstate) <=
    g_cons(result(data, oldstate), new_state(data, oldstate))
        where g_cons(a, speculation b) <= a:b.
```

Notice that the 'result' of the rendezvous must be computed before the rendezvous (computation of *g*) may terminate. However, the 'new state' is speculatively initiated, so need not be computed before the rendezvous ends. The 'new state' may be computed on one processor while in parallel the result of the rendezvous is used on another processor. Of course, on the next rendezvous the 'new state' should be used rather than *b*.

An annotation similar to the proposed **anywhere** annotation was introduced in Ref. 10. However, without an **at** annotation only simple divide and conquer type distributed evaluation was possible. Hudak and Smith[37] have proposed an annotation for placing work on specific processors in a network of processors.

5. NON-DETERMINISM

In conventional languages for parallel programming, non-determinism has been found to be unavoidable. If functional languages are to be used for parallel programming, it seems that some form of non-determinism will be required.

Several non-deterministic constructs for use in functional languages have been proposed.[26, 33, 34, 45] For example, *amb*[45] may non-deterministically return either of its two arguments subject to the restriction that its result will not be undefined unless both of its arguments are undefined.

Before proceeding, we should note that *amb* and the other non-deterministic constructs cited above (*frons*,[26] *or*[33] and *interleave*)[34] do not preserve referential transparency (the property that an expression always has the same value in the same environment). This means that our annotations become more than annotations when one of these constructs is added to a functional language. For example, consider the function

f(*x*) < = *x* = *x*.

This function will aways return *true* unless its argument is undefined (or is of a type for which the equality test is not defined). On the other hand

f(**name** *x*) < = *x* = *x*.

may return either *true* or *false* when applied to *amb*(*a*, *b*), since the argument will be evaluated twice and need not produce the same result both times.

One solution to this problem is given in Ref. 14. However, we shall use the *amb* construct in this paper, since it is more likely to be familiar to the reader.

There is an obvious problem with using a function such as *amb* in the context of a lazy interpreter. If *a* and *b* are two lists, what is the value of *amb*(*a*, *b*)? It can be argued that the list constructor is not strict, so once either list is known to be not completely undefined it may be returned (e.g. the list $\perp:\perp$ could be returned). With our annotations, we can control when decisions are made. For example, if we want to return a list only if it is fully defined, then we could write *amb*(*expand*(*a*), *expand*(*b*)), where *expand* is as given in Section 2.1.

Now let us consider the problem of non-deterministically merging two lists. The question is, when do we decide from which list we shall select the next element? Usually, we shall want to select the next element from a list as soon as the value of the first element of that list is available. We can write a function, *interleave*, to do this type of non-deterministic merging as follows:

```
interleave(speculation a, speculation b) <=
    amb(
if $a = nil$ then $b$
else $left\_eager\_cons(first(a), interleave(rest(a), b))$,
if $b = nil$ then $a$
else $left\_eager\_cons(first(b), interleave(rest(b), a)))$
```

(The function *left_eager_cons* is defined in Section 2.1.) Notice that *interleave* uses call by speculation parameter passing, so it can return a value before anything is known about one of the parameters provided enough is known about the other. The use of *left_eager_cons* prevents either of the two alternative computations from terminating before the value of the next element is actually available. (If we used the ordinary lazy cons, ':', we would be selecting the next element from a particular list as soon as we knew that the list was not *nil*.)

There is one minor problem concerning the efficiency of *interleave* under certain circumstances. Suppose a large amount of work must be done in computing each list element (which would argue in favour of our approach of not selecting an element until its value is available). We might often end up computing *first*(a) and *first*(b) in parallel. The work on the element which was not selected would be discarded. We can solve this problem by using

$$merge(a, b) <= interleave(buffer(a), buffer(b)).$$

The first element of a buffered list is always computed speculatively. Hence if *first*(b) is not selected, at least the work that has gone into computing *first*(b) will be work toward computing the first element of the list b, which is passed on to the next lower level of recursion.

6. FORMAL DETAILS

6.1. Speculative computations

In this subsection we shall consider an abstract interpreter for annotated λ-expressions. We shall start by considering how our annotated functional programs relate to annotated λ-expressions.

The translation of a functional program without annotations to the λ-calculus is straightforward. We shall use an annotated λ-calculus having three forms of application corresponding to parameter passing by name, value and speculation. We shall use an explicit apply operator, @, so that we have something on which to hang our annotations. The abstract syntax of our expressions is as follows:

$$\langle exp\rangle ::= \langle var\rangle \mid \lambda\langle var\rangle . \langle exp\rangle \mid @_N\langle exp\rangle\langle exp\rangle \mid @_V\langle exp\rangle\langle exp\rangle \mid @_S\langle exp\rangle\langle exp\rangle.$$

Since our functional notation annotates formal parameters and our λ-notation annotates applications, an extra level of abstraction is introduced in the translation. The functional expressions

λ **name** $v.e$,

λ **value** $v.e$, and

λ **speculation** $v.e$

translate to

$\lambda v.e$,

$\gamma v. @_V (\gamma v.e)v$, and

$\lambda v. @_S (\lambda v.e)v$

respectively. All other applications introduced in translating from the functional notation to the λ-notation should be $@_N$'s.

For the moment, let us forget about $@_S$. Fig. 7 shows an abstract interpreter from Ref. 10 (with some minor notational changes) for evaluating functional programs with the annotations considered in Section 2. (Note: in Ref. 10 $@_L$ and $@_E$ were used for $@_N$ and $@_V$ respectively). In the interpreter, **I** is the identity function, v stands for an arbitrary variable, and e, e_1 and e_2 stand for arbitrary expressions. The function *beta* does beta substitution. The result of *beta*($\lambda v.e_1, e_2$) is e_1 with e_2 substituted for every free occurrence of v in e_1. It is clear that the functions *reduce*, *value* and *make_ab* always convert a λ-expression into an equivalent one in the syntactic λ-calculus sense, with the different @s considered to be equivalent. (This is easy to prove by induction on the size of an expression, given that *beta*(e_1, e_2) is always equivalent to $@e_1 e_2$).

The value of a functional program, p, is *reduce*(p). In our meta notation, we assume that all parameters are passed by value and that the λ-expressions are data objects. In a sequential implementation, parameters are

```
$reduce(v) <= v$
$reduce(\lambda v.e) <= \lambda v.reduce(e)$
$reduce(@_N e_1 e_2) <= combine(make\_ab(e_1), e_2, reduce,$
                              $reduce, @_N)$
$reduce(@_V e_1 e_2) <= combine(make\_ab(e_1), value(e_2),$
                              $reduce, reduce, @_V)$

$value(v) <= v$
$value(\lambda v.e) <= \lambda v.value(e)$
$value(@_N e_1 e_2) <= combine(make\_ab(e_1), e_2, value, I,$
                              $@_N)$
$value(@_V e_1 e_2) <= combine(make\_ab(e_1), value(e_2),$
                              $value, I, @_V)$

$make\_ab(v) <= v$
$make\_ab(\lambda v.e) <= \lambda v.e$
$make\_ab(@_N e_1 e_2) <= combine(make\_ab(e_1), e_2,$
                              $make\_ab, I, @_N)$
$make\_ab(@_V e_1 e_2) <= combine(make\_ab(e_1, value(e_2),$
                              $make\_ab, I, @_V)$

$combine(e_1, e_2, f_1, f_2, apply) <=$
    if $is\_ab(e_1)$ then $f_1(beta(e_1, e_2))$
    else $apply\ f_2(e_1) f_2(e_2)$

$is\_ab(v) <= false$
$is\_ab(\lambda v.e) <= true$
$is\_ab(@_N e_1 e_2) <= false$
$is\_ab(@_V e_1 e_2) <= false$
```

Figure 7. Mixed-order reduction of annotated λ-expressions.

$reduce(@_S e_1 e_2) <= combine(make\_ab(e_1),\ \# e_2,\ reduce,\ reduce,\ @_S)$

$reduce(\# e) <= reduce(value(e))$

$value(@_S e_1 e_2) <= combine(make\_ab(e_1),\ \# e_2,\ value,\ \mathbf{I},\ @_S)$

$value(\# e) <= value(e)$

$make\_ab(@_S e_1 e_2) <= combine(make\_ab(e_1),\ \# e_2,\ make\_ab,\ \mathbf{I},\ @_S)$

$make\_ab(\# e) <= value(e)$

$ie\_ab(@_S e_1 e_1) <= false$

$is\_ab(\# e)$ Can not occur.

Figure 8. Reduction rules for speculative computations.

evaluated from left to right. In a parallel implementation, parallelism is produced by evaluating $make\_ab(e_1)$ and $value\ (e_2)$ in parallel when applying *reduce*, *value* or *make_ab* to an expression of the form $@_V e_1 e_2$.

The reader is referred to Ref. 10 for further information about this abstract interpreter and its evaluation. We note that many practical implementation details are not included in our abstract interpreter. For example $value(value(e))$ is always equal to $value(e)$. We would expect each expression to include a bit indicating whether it had already been evaluated by *value*. If e has been evaluated, $value(e)$ can return e immediately.

In order to describe how speculative computations are performed, we need to extend the abstract syntax for λ-expressions and to add some new equations to the abstract interpreter. We start by extending our syntax to

$\langle exp \rangle ::= \# \langle exp \rangle \mid$ as before.

An expression of the form $\# e$ denotes an expression whose value has been speculatively initiate but not yet required. Expressions of this form may be generated only as intermediate results in the evaluation process. (We would expect a process to compute and save $value(e)$ to be produced whenever an expression of the form $\# e$ is created. Any use of $\# e$ must start with the computation of $value(\# e)$, which is equal to $value(e)$. This value should be computed only once. Our equations do not show these details.) The equations defining the behaviour of the abstract interpreter for λ-expressions of the form $@_S e_1 e_2$ and $\# e$ are given in Fig. 8. The rule for $make\_ab(\# e)$ makes use of the fact that $make\_ab(value(e)) = value(e)$, which can easily be proved by induction on the size of e.

Let us consider how this abstract interpreter can be used to analyse the behaviour of a program.

6.1.1. Lists

List constructors and selectors can be defined in a number of ways in the λ-calculus. While an efficient implementation is almost certain to implement list constructors and selectors as primitives, it is useful to consider how the λ-calculus version would behave. (We would expect any correct implementation to behave in the same way).

The λ-calculus definitions of *cons*, *first* and *rest* given in Ref. 10 are:

$cons <= \lambda a.\lambda b.\lambda z.@_N @_N z\ a\ b$

$first <= \lambda x.@_N x \lambda a.\lambda b.a$

$rest <= \lambda x.@_N x \lambda a.\lambda b.b$

Lemma 1

$value(@_N @_N cons \# a \# b) <= \lambda z.@_N @_N z \# a \# b.$

Proof

The proof is straightforward, using the questions in Figs 7 and 8 as rewrite rules. Details are left to the reader. □

Lemma 2

$value(@_N @_N\ speculative\_cons\ \ a\ b)$ will evaluate via $value(@_N @_N cons \# a \# b)$ to $\lambda z.@_N @_N z \# a \# b$. given that

$speculative\_cons <= \lambda a.\lambda b.@_S @_S(\lambda a.\lambda b.@_N @_N\ cons\ a\ b)\ a\ b,$

Proof

The proof is straightforward. Details are left to the reader. □

Theorem 1

$value(first(x)$ **where** $x <= speculative\_cons(a, b)) <= value(a)$ even if b is $\perp$.

Proof

The proof is straightforward. Details are left to the reader. □

Corollary

$value(rest(x)$ **where** $x <= speculative\_cons(a, b)) <= value(b)$.

Other types of list constructors have a different behaviour. For example, $first(eager\_cons(a, \perp))$ will always be $\perp$, since $combine(\ldots, value(\perp), \ldots)$ must have the value $\perp$. In Ref. 16 it is observed that there are various types of lazy lists. By using annotations, it is possible to use several types of lazy lists within a single program. For example, *leaf_eager_cons* is a list constructor that is strict in its first argument but not its second. By introducing the **speculation** annotation, and the list constructors that can be constructed using it, we are able to define lists with lazy list semantics that also facilitate parallelism, as seen in the buffer example in Section 3.3.

6.1.2. Differences between speculation and name

Due to interactions with call by value, the semantics of **speculation** and **name** are slightly different.

If we compare the equations for the two mechanisms, we see that call by speculation generates the expression $\# e_2$ where call by name uses e_2. The difference lies in how these two expressions are treated later.

There are two places where $\# e$ and e are treated differently. Since $reduce(value(e)) \equiv reduce(e)$, where

$\equiv$ is used to mean that two λ-calculus expressions are identical, the only interesting difference is in the treatment of *make_ab*. It is not the case that *make_ab*(e) is always the same as *value*(e). (Although *value*(e) $\equiv$ *value*(*make_ab*(e))).

Consider the function

$$f \equiv \lambda a.\lambda b.@_V\, \lambda c.a\ b$$

Clearly *make_ab*(f) $<=f$, while

$$\mathit{value}(f) <= \mathit{value}(\lambda a.\lambda b.@_V\, \lambda c.a\ b)$$

$$<= \lambda a.\lambda b.a.$$

Hence

$$\mathit{value}(@_S\, \lambda g.@_N\, @_N\, g\ x \perp f)$$

$$<= \mathit{value}(x).$$

On the other hand

$$\mathit{value}(@_N\, \lambda g.@_N\, @_N\, g\ x \perp f)$$

$$<= \perp,$$

since *value*($\perp$) is $\perp$.

In general, it is better not to pass a function by value or speculation, since this will eliminate much of the built-in evaluation control. The problem is that often we do not want functions reduced before they are applied to their arguments. There are exceptions. For example, if we pass the function $\lambda a.\lambda b.\mathit{first}(a\!:\!b)$ by value, it will simplify to $\lambda a.\lambda b.a$, which may save some work if the corresponding formal parameter is applied many times. Hughes has argued that one of the primary advantages to functional programming is the modularity which results when programs are constructed by combining functions using programmer-defined higher-order functions.[41] Passing functions by value can be used to simplify functional expressions, if care is used.

6.2. Distributed computation

We shall assume that we have an unbounded number of virtual processors, and that each use of **anywhere** places work on a new virtual processor. Of course, an implementation must be responsible for mapping virtual processors on to physical processors.

Each expression will be associated with a particular virtual processor. (We would expect the root of the representation of that expression to reside on the corresponding actual processor, and would expect any application of *reduce*, *value* or *make_ab* to that expression to be performed on the same processor.) In order to describe which virtual processor is associated with a particular expression we shall insert a superscript before the first symbol of the expression giving the name of the virtual processor. If i is a virtual processor name then ${}^ie$ is (a copy of) expression e on processor i.

Fig. 9 shows how the evaluation of a distributed annotated λ-expression without **ats** or **anywheres** should be performed. For example, the rule for $@_N$ and *value* says that the root of the operator should be moved to the processor associated with the application before *make_ab* is applied to it. The operand (which is probably represented by an off-processor pointer) should not be moved.

In computing $\mathit{beta}({}^1\lambda v.{}^2e_1, {}^3e_2)$, ${}^3e_2$ is substituted for

$$\mathit{reduce}({}^1v) <= {}^1v$$

$$\mathit{reduce}({}^1\lambda v.{}^2e) <= {}^1\lambda v.\mathit{reduce}({}^1e)$$

$$\mathit{reduce}({}^1@_N{}^2e_1{}^3e_2) <= \mathit{combine}(\mathit{make\_ab}({}^1e_1), {}^3e_2, \mathit{reduce}, \mathit{reduce}, @_N)$$

$$\mathit{reduce}({}^1@_V{}^2e_1{}^3e_2) <= \mathit{combine}(\mathit{make\_ab}({}^1e_1), \mathit{value}({}^1e_2), \mathit{reduce}, \mathit{reduce}, @_V)$$

$$\mathit{reduce}({}^1@_S{}^2e_1{}^3e_2) <= \mathit{combine}(\mathit{make\_ab}({}^1e_1), {}^1\#{}^1e_2, \mathit{reduce}, \mathit{reduce}, @_S)$$

$$\mathit{reduce}({}^1\#{}^2e) <= \mathit{reduce}({}^1e_1) \textbf{ where } {}^2e_1 \equiv \mathit{value}({}^2e)$$

$$\mathit{value}({}^1v) <= {}^1v$$

$$\mathit{value}({}^1\lambda v.{}^2e) <= {}^1\lambda v.\mathit{value}({}^1e)$$

$$\mathit{value}({}^1@_N{}^2e_1{}^3e_2) <= \mathit{combine}(\mathit{make\_ab}({}^1e_1), {}^3e_2, \mathit{value}, \mathbf{I}, @_N)$$

$$\mathit{value}({}^1@_V{}^2e_1{}^3e_2) <= \mathit{combine}(\mathit{make\_ab}({}^1e_1), \mathit{value}({}^1e_2), \mathit{value}, \mathbf{I}, @_V)$$

$$\mathit{value}({}^1@_S{}^2e_1{}^3e_2) <= \mathit{combine}(\mathit{make\_ab}({}^1e_1), {}^1\#{}^1e_2, \mathit{value}, \mathbf{I}, @_S)$$

$$\mathit{value}({}^1\#{}^2e) <= {}^1e_1 \textbf{ where } {}^2e_1 \equiv \mathit{value}({}^2e)$$

$$\mathit{make\_ab}({}^1v) <= {}^1v$$

$$\mathit{make\_ab}({}^1\lambda v.{}^2e) <= {}^1\lambda v.{}^2e$$

$$\mathit{make\_ab}({}^1@_N{}^2e_1{}^3e_2) <= \mathit{combine}(\mathit{make\_ab}({}^1e_1), {}^3e_2, \mathit{make\text{–}ab}, \mathbf{I}, @_N)$$

$$\mathit{make\_ab}({}^1@_V{}^2e_1{}^3e_2) <= \mathit{combine}(\mathit{make\_ab}({}^1e_1), \mathit{value}({}^1e_2), \mathit{make\_ab}, \mathbf{I}, @_V)$$

$$\mathit{make\_ab}({}^1@_S{}^2e_1{}^3e_2) <= \mathit{combine}(\mathit{make\_ab}({}^1e_1), {}^1\#{}^1e_2, \mathit{make\_ab}, \mathbf{I}, @_S)$$

$$\mathit{make\_ab}({}^1\#{}^2e) <= {}^1e_1 \textbf{ where } {}^2e_1 \equiv \mathit{value}({}^2e)$$

```
$combine({}^1e_1, {}^2e_2, f_1, f_2, apply) <=$
    **if** $is\_ab({}^1e_1)$ **then** $f_1(beta({}^1e_1, {}^2e_2))$
    **else** ${}^1apply\ f_2({}^1e_1) f_2({}^2e_2)$
```

is_ab ignores location.

Figure 9. Reduction rules annotated to show placement of work and results.

all free occurrences of v in ${}^2e_1$ (with suitable name changes to avoid name conflicts, as is usual with beta substitution) and then the root of the result is moved to virtual processor 1. That is, the result of beta substitution is always associated with the processor that initiates the beta substitution. For example.

$$\mathit{beta}({}^1\lambda v.{}^2v, {}^3e) \equiv {}^1e$$

and

$$\mathit{beta}({}^1\lambda v.\ {}^2@_N\ {}^3v{}^4v, {}^5e) \equiv {}^1@_N{}^5e{}^5e.$$

We can now consider how expressions of the form e **anywhere** and e_1 **at** e_2 should be represented in our λ-notation.

We shall again extend our abstract syntax to include two additional forms.

$\langle exp \rangle ::=$ as before | **anywhere**$\langle exp \rangle$ | **at**$\langle exp \rangle \langle exp \rangle$.

The forms e **anywhere** and e_1 **at** e_2 translate to **anywhere** e and **at** $e_1 e_2$. The new reduction rules for these new syntactic forms are given in Fig. 10. The processor name n is used to indicate a new virtual processor. In interpreting these equations, n should be assigned the name of a not previously used virtual processor.

$$\mathit{reduce}({}^{1}\mathbf{anywhere}\ {}^{2}e) <= \mathit{reduce}({}^{n}e)$$
$$\mathit{reduce}({}^{1}\mathbf{at}\ {}^{2}e_1{}^{3}e_2) <= \mathit{reduce}({}^{3}e_1)$$
$$\mathit{value}({}^{1}\mathbf{anywhere}\ {}^{2}e) <= \mathit{value}({}^{n}e)$$
$$\mathit{value}({}^{1}\mathbf{at}\ {}^{2}e_1{}^{3}e_2) <= \mathit{value}({}^{3}e_1)$$
$$\mathit{make\_ab}({}^{1}\mathbf{anywhere}\ {}^{2}e) <= \mathit{make\_ab}({}^{n}e)$$
$$\mathit{make\_ab}({}^{1}\mathbf{at}\ {}^{2}e_1{}^{2}e_2) <= \mathit{make\_ab}({}^{3}e_1)$$

Figure 10. Reduction rules for moving work.

We can use these equations to study where work and data values are located in a distributed computation.

Theorem 2

If on processor 1 the value of *first*(u:v) is computed, the result will be the value of u computed on processor 1.

Proof

Let us rewrite the above expression in terms of the annotated λ-calculus. We will use the superscript ? to indicate that a particular computation is on an arbitrary processor. (I.e. the ? means we do not care where the expression is located. All ?s need not stand for the same processor).

$$\mathit{value}(\mathit{first}(u\!:\!v))$$
$$<= \mathit{value}({}^{1}@_N{}^{?}\lambda x.{}^{?}@_N{}^{?}x{}^{?}\lambda a.{}^{?}\lambda b.{}^{?}a{}^{?}\lambda z .{}^{?}@_N@_N{}^{?}z{}^{?}u{}^{?}v)$$
$$<= \mathit{combine}(\mathit{make\_ab}({}^{1}\lambda x.{}^{?}@_N{}^{?}x{}^{?}\lambda a.{}^{?}\lambda b.{}^{?}a), {}^{?}\lambda z .{}^{?}@_N{}^{?}@_N{}^{?}z{}^{?}u{}^{?}v, \mathit{value}, \mathbf{I}, @_N)$$
$$<= \mathit{combine}({}^{1}\lambda x.{}^{?}@_N{}^{?}x{}^{?}\lambda a.{}^{?}\lambda b.{}^{?}a, {}^{?}\lambda z .{}^{?}@_N{}^{?}@_N{}^{?}z{}^{?}u{}^{?}v, \mathit{value}, \mathbf{I}, @_N)$$
$$<= \mathit{value}(\mathit{beta}({}^{1}\lambda x.{}^{?}@_N{}^{?}x{}^{?}\lambda a.{}^{?}\lambda b.{}^{?}a, {}^{?}\lambda z .{}^{?}@_N{}^{?}@_N{}^{?}z{}^{?}u{}^{?}v))$$
$$<= \mathit{value}({}^{1}@_N{}^{?}\lambda z.{}^{?}@_N{}^{?}@_N{}^{?}z{}^{?}u{}^{?}v{}^{?}\lambda a.{}^{?}\lambda b.{}^{?}a)$$
$$<= \mathit{combine}(\mathit{make\_ab}({}^{1}\lambda z.{}^{?}@_N{}^{?}@_N{}^{?}z{}^{?}u{}^{?}v), {}^{?}\lambda a.{}^{?}\lambda b.{}^{?}a, \mathit{value}\ \mathbf{I}, @_N)$$
$$<= \mathit{value}(\mathit{beta}({}^{1}\lambda z.{}^{?}@_N{}^{?}@_N{}^{?}z{}^{?}u{}^{?}v, {}^{?}\lambda a.{}^{?}\lambda b.{}^{?}a))$$
$$<= \mathit{value}({}^{1}@_N{}^{?}@_N{}^{?}\lambda a.{}^{?}\lambda b.{}^{?}a{}^{?}u{}^{?}v)$$
$$<= \mathit{combine}(\mathit{make\_ab}({}^{1}@_N{}^{?}\lambda a.{}^{?}\lambda b.{}^{?}a{}^{?}u), {}^{?}v, \mathit{value}, \mathbf{I}, @_N)$$

where $\mathit{make\_ab}({}^{1}@_N{}^{?}\lambda a.{}^{?}\lambda b.{}^{?}a{}^{?}u)$

$$<= \mathit{combine}(\mathit{make\_ab}({}^{1}\lambda a.{}^{?}\lambda b.{}^{?}a), {}^{?}u, \mathit{make\_ab}, \mathbf{I}, @_N)$$
$$<= \mathit{make\_ab}(\mathit{beta}({}^{1}\lambda a.{}^{?}\lambda b.{}^{?}a, {}^{?}u))$$
$$<= \mathit{make\_ab}({}^{1}\lambda b.{}^{?}u)$$
$${}^{1}\lambda b.{}^{?}u$$

so *first*(u:v)

$$<= \mathit{combine}({}^{1}\lambda b.{}^{?}u, {}^{?}v, \mathit{value}, \mathbf{I}, @_N)$$
$$<= \mathit{value}(\mathit{beta}({}^{1}\lambda b.{}^{?}u, {}^{?}v))$$
$$<= \mathit{value}({}^{1}u). \qquad \square$$

Corollary

If on processor 1 the value of *rest*(u:v) is computed, the result will be the value of v computed on processor 1.

Proof

The above proof with slight modification will apply to this result. $\square$

We note that if the evaluation of u were speculatively initiated on processor 2, *first*(# u:v) would evaluate on processor 1 to *value*(${}^{1}\#\,{}^{2}u$), which in turn evaluates as ${}^{1}e$ where ${}^{2}e \equiv \mathit{value}({}^{2}u)$, so u is actually evaluated on processor 2 as we would expect. Similarly, if u contains an **at** annotation (as in our *pipe* example in Section 4) the evaluation of u may move work off the current processor.

6.3. Non-determinism

Let *Amb* be the function corresponding to *amb* in our meta notation. We can define the behaviour of *amb* in the context of an annotated functional program very simply by using *Amb*. (The function *amb* is defined by McCarthy in Ref. 45.)

Let us again extend our annotated λ-notation to

$\langle exp \rangle ::=$ as before | **amb**$\langle exp \rangle \langle exp \rangle$

with the obvious translation from annotated functional programs to annotated λ-expressions. The new rules we require are:

$$\mathit{reduce}({}^{1}\mathbf{amb}{}^{2}e_1{}^{3}e_2) <= \mathit{Amb}(\mathit{reduce}({}^{1}e_1), \mathit{reduce}({}^{1}e_2))$$
$$\mathit{value}({}^{1}\mathbf{amb}{}^{2}e_1{}^{3}e_2) <= \mathit{Amb}(\mathit{value}({}^{1}e_1), \mathit{value}({}^{1}e_2))$$
$$\mathit{make\_ab}({}^{1}\mathbf{amb}{}^{2}e_1{}^{2}e_2) <= \mathit{Amb}(\mathit{make\_ab}({}^{1}e_1), \mathit{make\_ab}({}^{1}e_2)).$$

7. IMPLEMENTATION NOTES

In this section we shall very briefly consider several implementation issues.

7.1. Combinator reduction

We are planning to implement an annotated functional language using combinator reduction.[8, 55, 56] The translation of annotated λ-expressions to annotated combinatory expressions described in Ref. 10 carried over to call by speculation. We shall introduce two new combinators, **at** and **anywhere**, having the obvious meaning.

7.2. Speculative computation

There are a couple of issues relating to the evaluation of speculative results which warrant attention.

When a speculative computation is initiated, a cell where the result is to be placed should be created and immediately returned as a result. There should be a semaphore associated with this cell. When a value is placed in the cell, a signal should be sent. Any process using the result of a speculative computation should use the semaphore to avoid accessing the value before it is ready.

The fact that a speculative computation may be discarded means that some speculative computations may never need to be evaluated. For this reason we should prefer speculative work not to compete with other computations in gaining access to processors. Furthermore, we must require that speculative work does not prevent any other work from terminating, since non-terminating speculative work may be discarded.

For these reasons we propose to class work as *mandatory* or *speculative.* All work generated by a **speculation** including computations internal to a speculative computation is speculative until it is found to be needed, at which time it becomes mandatory. All other computation is mandatory. Speculative work should only be run when a processor would otherwise be idle.

Grit and Page[29] have proposed an algorithm for aborting speculative work when it is found to be no longer required. The method will meet our needs. In addition, a simple modification will lead to an algorithm for upgrading speculative work to mandatory work when this is necessary.

7.3. Recursion

When a functional program that does not contain recursive equations is evaluated, no cyclic structure result. It is easy to add recursion without introducing cycles.

In our examples we have allowed recursion in **where** clauses. In terms of the λ-calculus, we will take the statement

e_1 **where** $x <= e_2$,

where x is a variable and e_1 and e_2 are arbitrary expressions, to mean

$(\lambda$ **value** $x.e_1)\ (\mathbf{Y}\lambda x.e_2)$

where

$\mathbf{Y} = \lambda h.((\lambda x.h(zz))(\lambda z.h(zz)))$.

(We take $f(z) <= e$ to be equivalent to $f <= \lambda x.e$. and also allow **let** $x <= e_2$ **in** e_1 to mean the same thing as e_1 **where** $x <= e_2$. Some languages require the programmer to use **let rec** rather than **let** if recursion is desired).

The reader is referred to Refs 20, 50 and 55 for further information as to how the **Y** combinator implements recursion by computing the least fixed point of a function. We note that if x does not occur in e_2 $(\mathbf{Y}\lambda x.e_2)$ quickly reduces to e_2.

(In the interest of efficiency, we would expect **Y** to be implemented as a primitive. We should also expect a compiler to replace $(\mathbf{Y}\lambda x.e_2)$ with e_2 when x does not occur in e_2).

There is no significant problem in allowing annotations in recursive definitions. If *anno* stands for **name, speculation** or **value** (with no annotation implying **value** as usual) then

e_1 **where** *anno* $x <= e_2$

translates to

$(\lambda\ \mathit{anno}\ x.e_1)\ (\mathbf{Y}\lambda x.e_2)$.

Parameters within **Y** are passed by name. That is,

$\mathbf{Y} = \lambda$ **name** $h.((\lambda$ **name** $x.h(xx))\ (\lambda$ **name** $x.(xx)))$.

Clearly, this implementation of recursion does not introduce cyclic structures.

This acyclic implementation works fine for recursive functions. The one time we need cycles is when we are recursively defining data objects. Whenever an expression of the form **Y***e* is passed by value or speculation the following should happen.

(1) A new cell where the result of the evaluation of **Y***e* is to be placed should be allocated, as with a speculative computation.

(2) The application of e to this new cell should be speculatively initiated. When a result is returned, it should be placed in the cell.

(3) The cell allocated above is now substituted for the original expression, **Y***e*, and the computation proceeds as normal.

We note that the evaluation of an expression such as

ones **where** *ones* $<=$ *eager_cons*(1, *ones*)

will deadlock. On the other hand,

ones **where** *ones* $<=$ *speculative_cons*(1, *ones*)

will return a cyclic data structure. Deadlock will occur only in those cases where an acyclic implementation would result in non-termination.

7.4. Distributed evaluation

The equations in Section 6 indicate that work should be transferred between processors a cell at a time, where a cell corresponds to an expression excluding sub-expressions. In fact, an expression and all sub-expressions which are not contained either in an operand of an $@_N$s, or in an **at** or **anywhere** sub-expression, can be moved at the same time. This should be done both to avoid the very high overheads associated with transferring one cell at a time, and to avoid repeating the transfers every time an expression is used.

7.5. Process management

By using a virtual tree machine approach to scheduling processes,[8, 11] per-processor storage requirements for a parallel or distributed system can be kept close to the single processor requirements. The basic idea is that once all processors have sufficient work, they each go into a sequential mode of operation, reverting to a parallel mode only when it is necessary to generate more work for other machines. In effect, this leads to a dynamic process granularity.

8. CONCLUSION

We have seen that a few simple annotations are sufficient to give a programmer a considerable degree of control over the run-time behaviour of a functional program. This makes it possible to write programs which are efficient with respect to space and communication requirements as well as time.

In most cases, annotations are unnecessary. Reasonable defaults ensure that the best evaluation strategy is selected in most situations. Where annotations are required, they are easy to use and understand. It is possible to encapsulate annotations in functions to produce additional mechanisms for controlling evaluation. For example, *buffer* (*a*), as defined in Section 3, and *pipe*(*a*), as defined in Section 4, are both equivalent to *a*. We can therefore think of *buffer*(...) and *pipe*(...) as user-defined annotations.

We believe that use of the proposed annotations makes it possible to combine the simplicity and sound mathematical foundation of functional programming with the control and asymptotic efficiency of procedural programming.

Acknowledgement

The author would like to thank V. J. Rayward-Smith and G. P. McKeown for their many helpful comments on an earlier draft of this paper.

REFERENCES

1. Arvind and R. E. Thomas, *I-structures: an Efficient Data Structure for Functional Languages*. Report MIT/LCS/TM-178, Laboratory for Computer Science, Massachusetts Institute of Technology (1980, revised 1981).
2. Arvind, D. E. Culler, R. A. Iannucci, V. Kathail, K. Pingali and R. E. Thomas, *The Tagged Token Dataflow Architecture*. Laboratory for Computer Science, Massachusetts Institute of Technology (1983).
3. Arvind, M. L. Dertouzos and R. A. Iannucci, *A Multiprocessor Emulation Facility*, Technical Report 302, Laboratory for Computer Science, Massachusetts Institute of Technology (1983).
4. Arvind, *Sharing of Computations in Functional Language Implementations*. Laboratory for Computer Science, Massachusetts Institute of Technology (1984).
5. L. Augustsson, A compiler for lazy ML. *Conf. Rec. 1984 ACM Symp. on LISP and Functional Programming, Austin, Texas*, 218–227 (1984).
6. J. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. of the ACM* **21**, (8) 613–641 (1978).
7. R. M. Burstall, D. B. MacQueen and D. T. Sannella, *HOPE: an Experimental Applicative Language*. Report CSR-62-80, University of Edinburgh Department of Computer Science, (1980).
8. F. W. Burton and M. R. Sleep, Executing functional programs on a virtual tree machine. *Proc. 1981 Conf. Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire*, 1987–194 (1981).
9. F. W. Burton, A linear space translation of functional programs to Turner combinators. *Information Processing Letters*, **14**, (5) 201–204 (1982).
10. F. W. Burton, Annotations to control parallelism and reduction order in the distributed evaluation of functional programs. *Transactions on Progress in Language and Systems* **6**, (2) 159–174 (1984).
11. F. W. Burton and M. M. Huntbach, Virtual tree machines. *IEEE Trans. on Computers*, **C-33**, (3) 278–280 (1984).
12. F. W. Burton, Speculative computation, parallelism, and functional programming. *IEEE Trans. on Computers*, **C-34**, (12) 1190–1193 (1985).
13. F. W. Burton, Controlling speculative computation in a parallel functional programming language. *Proc. Fifth International Conference on Distributed Computing Systems, Deriver, Colorado, May 1985*, pp. 453–458.
14. F. W. Burton, Nondeterminism with referential transparency in functional programming languages (submitted for publication).
15. L. Cardelli, Compiling a functional language. *Conf. Rec. 1984 ACM Symp. on LISP and Functional Programming, Austin, Texas, August 1984*, pp. 208–217.
16. R. Cartwright and J. Donahue, The semantics of lazy (and industrious) evaluation. *Proc. 1982 ACM Symposium on LISP and Functional Programming, Pittsburgh, Penn., Aug. 1982*, pp. 253–264.
17. T. J. W. Clarke, P. J. S. Gladstone, C. D. McLean and A. C. Norman, SKIM – The S. K. I. reduction machine. *Proc. 1980 LISP Conf., Stanford, California, Aug. 1980*, pp. 128–135.
18. J. S. Conery and D. F. Kibler, Parallel interpretation of logic programs. *Proc. 1981 Conf. Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire, Oct. 1981*, pp. 163–170.
19. J. S. Conery, The AND/OR process model for parallel interpretation of logic programs. *Ph.D. dissertation*, Technical Report 204, Department of Computer Science, University of California at Irvine (1983).
20. H. B. Curry and R. Feys, *Combinatory Logic*, Vol. 1, North-Holland, Amsterdam (1958).
21. J. Darlington and M. Reeve, ALICE: a multi-processor reduction machine for the parallel evaluation of applicative languages. *Proc. 1981 Conf. Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire, Oct. 1981*, pp. 65–75.
22. J. Darlington, Program transformation. In *Functional Programming and Its Applications*, edited J. Darlington, P. Henderson and D. A. Turner, 193–215. Cambridge University Press (1982).
23. J. B. Dennis, Data flow supercomputers, *Computer* **13** (11) 48–56 (1980).
24. J. B. Dennis, An operational semantics for a language with early completion data structures. *Proc. Internat. Colloquium on Foundations of Programming Concepts, Peniscola, Spain, Apr. 1981*, pp. 260–267.
25. D. P. Friedman and D. S. Wise, Cons should not evaluate its arguments, *3rd Int. Coll. Automata Languages and Programming, Edinburgh, Scotland, 1976*, pp. 257–284.
26. D. P. Friedman and D. S. Wise, An indeterminate constructor for applicative programming, *Conf. Rec. 7th ACM Symp on Prin. of Prog. Lang., 1980*, pp. 245–250.
27. R. P. Gabriel and J. McCarthy, *Conf. Rec. 1984 ACM Symp. on LISP and Functional Programming, Austin, Texas, August 1984*, p. 2544.
28. D. H. Grit and R. L. Page, A multiprocessor model for parallel evaluation of applicative programs. *Journal of digital Systems* **4**, (2) 135–151 (1980).
29. D. H. Grit and R. L. Page, Deleting irrelevant tasks in an expression-oriented multiprocessor system. *Transactions on the Progress in Language and Systems* **3** (1) 49–59 (1981).
30. J. R. Gurd, C. C. Kirkham and I. Watson, The Manchester prototype dataflow computer. *Comm. of the ACM* **28**, (1) 34–52 (1985).
31. R. H. Halstead, Jr, Multilisp: a language for concurrent symbolic computation. *Transactions on Progress in Languages and Systems* **7**, (4) 501–538 (1985).
32. P. Henderson and J. M. Morris, A lazy evaluator. *Conf. Rec. 3rd ACM Symp. on Prin. of Prog. Lang., Atlanta, Ga., Jan. 1976*, pp. 95–103.
33. P. Henderson, *Functional Programming: Application and*

Implementation. Prentice-Hall, Englewood Cliffs, N.J. (1980).

34. P. Henderson, Purely functional operating systems. In *Functional Programming and Its Applications*, edited J. Darlington, P. Henderson and D. A. Turner, pp. 177–192. Cambridge University Press (1982).
35. P. Hudak and D. Kranz, A combinator based compiler for functional languages. *Conf. Rec. 11th ACM Symp. on Prin. Programming Languages, Salt Lake City, Utah, Jan. 1984*, pp. 121–132.
36. P. Hudak and B. Goldberg, Experiments in diffused combinator reduction. *Conf. Rec. 1984 ACM Symp. on LISP and Functional Programming, Austin, Texas, August 1984*, pp. 167–176.
37. P. Hudak and L. Smith, Para-functional programming: a paradigm for programming multiprocessor systems (draft), Department of Computer Science, Yale University, New Haven, CT (1985).
38. P. Hudak and J. Young, Higher-order strictness analysis in untyped lambda calculus. *Conf. Rec. 13th ACM Symp. on Prin. Programming Languages, St Petersburg Beach, Florida, Jan. 1986*, pp. 97–109.
39. R. J. M. Hughes, Super combinators: a new implementation method for applicative programs. *Proc. 1982 ACM Symposium on LISP and Functional Programming, Pittsburgh, Penn., Aug. 1982*, pp. 1–10.
40. R. J .M. Hughes, *Parallel Functional Languages Use Less Space*. Programming Research Group, Oxford University (1984).
41. R. J. M. Hughes, *Why Functional Programming Matters*, Programming Methodology Group memo PMG-40, Department of Computer Science, Chalmers University of Technology and University of Gothenburg (1984).
42. T. Johnson, Efficient evaluation of lazy evaluation. *Proc. ACM Symp. Compiler Construction, Montreal, Canada, June 1984*, pp. 58–69.
43. G. Lindstrom, Static evaluation of functional programs, to appear in *Proc. 1986 ACM SIGPLAN Notices Symposium on Compiler Construction.*
44. G. A. Mago, A network of microprocessors to execute reduction languages. *Int. J. Comput. Inform. Sci.* **8**, (5, 6) 349–385, 435–471 (1979).
45. J. McCarthy, A basic mathematical theory of computation. In *Computer Programming and Formal Systems*, edited P. Braffort and D. Hirschberg, pp. 33–70. North-Holland, Amsterdam (1963).
46. A. Mycroft, *The Theory and Practice of Transforming Call by need into Call by value*. Internal Report CSR 88–81, Department of Computer Science, University of Edinburgh (1981).
47. W. Myers, Lisp machines displayed at AI conference. *Computer* **15** (11), 79–82 (1982).
48. J. Schwarz, Using annotations to make recursive equations behave. *IEEE Trans. on Software Eng.* **SE-8** (1), 21–33 (1982).
49. C. L. Seitz, The cosmic cube. *Comm. of the ACM* **28** (1) 22–33 (1985).
50. J. E. Stoy, *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*, Cambridge, Mass. (1977).
51. J. E. Stoy, Some mathematical aspects of functional programming. In *Functional Programming and Its Applications*, edited J. Darlington, P. Henderson and D. A. Turner, pp. 217–252. Cambridge University Press (1982).
52. W. R. Stoye, T. J. W. Clarke and A. C. Norman, Some practical methods for rapid combinator reduction. *Conf. Rec. 1984 ACM Symp. on LISP and Functional Programming, Austin, Texas, August 1984*, pp. 159–166.
53. L. J. Thomas, *Untitled Letter Circulated to Various Interested Researchers*. Burroughs Corporation, Austin Research Center (1984).
54. P. C. Treleaven, D. R. Brownbridge and R. C. Hopkins, Data-driven and demand-driven computer architecture. *Computing Surveys* **14**, (1) 93–143 (1982).
55. D. A. Turner, A new implementation technique for applicative languages. *Software – Practice & Experience* **9**, 31–49 (1979).
56. D. A. Turner, Another algorithm for bracket abstraction. *Journal of Symbolic Logic* **44** (2), 267–270 (1979).
57. D. A. Turner, Recursion equations as a programming language. In *Functional Programming and Its Applications*, edited J. Darlington, P. Henderson and D. A. Turner, pp. 1–28. Cambridge University Press (1982).
58. D. A. Turner, Combinator reduction machines. *Proc. Internat. Workshop on High Level Computer Architecture, Los Angeles, May 1984.*
59. D. A. Turner, Functional programs as executable specifications. In *Mathematical Logic and Programming Languages*, edited C. A. R. Hoare and J. Shepherdson, pp. 29–54. Prentice-Hall, Englewood Cliffs, N.J. (1985).
60. C. P. Wadsworth, *Semantics and Pragmatics of the Lambda-Calculus*. Oxford University Press. (1971).