Para-Functional Programming: A Paradigm for Programming Multiprocessor Systems

Paul Hudak
Lauren Smith

Yale University
Department of Computer Science
Hudak@Yale, Smith-Lauren@Yale

Abstract

One of the most important pragmatic advantages of functional languages is that concurrency in a program is *implicit* -- there is no need for special constructs to express parallelism as is required in most conventional languages. Furthermore, it is fairly easy for compilers to automatically determine the concurrency as a step toward decomposing a program for execution on a suitable parallel architecture. Yet it is often the case that one knows precisely the *optimal decomposition* for execution on a particular machine, and one can never expect a compiler to determine such optimal mappings in all cases. This paper is concerned with ways to allow the programmer to *explicitly* express this mapping of program to machine, by using annotations that, given a few minor constraints, cannot alter the functional semantics of the program. We show through several detailed examples the expressiveness and conciseness of the resulting "para-functional" programming methodology, using an experimental language called *ParAlfl* based on our ideas. We also give a formal denotational description of the mapping semantics using a notion of *execution trees*.

This research was supported in part by NSF Grants DCR-8403304 and DCR-8451415, and a Faculty Development Award from IBM.

1. Introduction

The advantages of functional languages have been well-argued by functional programming advocates in the past several years. One of the most important *pragmatic* advantages is that functional programs expose parallelism in a "natural way," and that it is easy for a compiler to detect such parallelism for exploitation on a suitable parallel architecture. The lack of side-effects accounts (at least partially) for the well-known Church-Rosser Property [37] that guarantees determinacy in the resulting parallel computation. Indeed, many of the earlier functional languages were developed simultaneously with work on dataflow or reduction machines (for example, VAL and the Static Dataflow Machine [10, 30], ID and the U-interpreter [1, 2], DDN and the dataflow machine DDM1 [7, 8], and FGL and the reduction machine AMPS [25, 26]). In all of these efforts the parallelism is detected *automatically* by the system -- the user is not required to provide extraneous information such as that needed in most imperative programming languages designed for parallel computation.

Recently there has also been a great deal of interest in parallel computers of various sorts, in particular the so-called *multiprocessor*, typically characterized as a collection of autonomous processing elements with only local store, interconnected by a homogeneous communications network, and communicating by "messages." The interest in machines of this type is not surprising, since not only do they avoid the classical "von Neumann bottleneck" by being effectively decentralized, but they are also extensible, and in general quite easy to build. Indeed several existing machines meet this description, such as the Butterfly Multiprocessor [4], Cosmic Cube [34], Intel iPSC [23], and even Cm* [9], to name a few, and there are many proposed machines whose construction is underway.

The combination of functional languages and multiprocessors thus seems like a natural one. Indeed, considerable research is underway in this direction, such as Keller's Rediflow Multiprocessor [28] and the authors' work on DAPS [16, 18, 19]. Such work shows considerable promise: a functional program is automatically decomposed for dynamic distribution on a network of processors, typically by some sort of "load-balancing" or "diffusion-scheduling" strategy, and execution takes place as the processors cooperatively accomplish a global form of graph-reduction. Simulated performance figures are quite encouraging, and it is hoped that the resulting systems will perform quite well on a wide variety of programs.

We prefer to view the aforementioned systems as *general-purpose multi-user computer systems*. Yet it is often the case that one has a *dedicated* parallel machine for a particular application, and furthermore that one knows *precisely* the *optimal mapping* of one's program onto that machine. One can never expect an automated system to determine this optimal mapping in all cases (in general such a task is clearly undecideable), so it is desireable to allow the user to express the mapping explicitly. This need often arises, for example, in scientific computing, where many classical algorithms have been re-designed for optimal performance on particular architectures. As it stands, there are almost no languages providing this capability.

Our goal then, is to remedy this situation. Since functional languages seem generally well-suited to parallel computing, we use it as a basis for the following simple solution: a functional program, being essentially an immutable object, may be mapped to a machine by *annotating* its subexpressions, in such a way that the program's functional behavior is not altered; i.e., the program itself remains unchanged. We refer to the resulting methodology as *para-functional programming*, since not only does it provide a much-needed tool for expressing parallel computation, but it also provides an operational semantics that is truly "extra," or "beyond," the functional semantics of the program. The resulting methodology is quite powerful, for several reasons:

First, *it is very flexible*. Not only is the idea easily adapted to any functional language, but also any network topology may be captured by the notation, since no a priori assumptions are made with regard to the logical structure of the physical system. All of the benefits of conventional scoping disciplines are available to create modular programs that conform to the topology of a given physical system.

Second, *the annotations are natural and concise*. There are no special control constructs, no message-passing constructs, and in general no forms of "excess baggage" to express the rather simple notion of "where and when to compute things." We will show through several non-trivial examples the perspicuous nature of the annotations, and that very few annotations are required to express most typical mappings.

Finally, with some minor syntactic constraints, *if a para-functional program is stripped of its annotations, it is still a perfectly valid functional program*. This means that it can be written and debugged on a uniprocessor without the annotations, and then executed on a parallel system by adding the annotations for increased performance. Portability is enhanced since only the annotations need to change when one moves from one parallel topology to another (unless the algorithm itself changes). The ability to debug a program independently of the parallel machinery is invaluable.

2. ParAlfl = A Simple Functional Language + Annotations

ParAlfl is an experimental para-functional programming language derived from an existing functional language called ALFL [17]. Since the proposed annotations could easily be added to any functional language, we describe the base language separately.[1]

2.1. The Base Language

The base language of ParAlfl is block-structured, lexically-scoped, and has lazy evaluation semantics. It is similar in style to several modern functional programming languages, including SASL (and it's successors KRC and Miranda) [38, 39], FEL [27], and "lazy" variants of ML [31, 24]. We describe the base language only briefly here, with the assumption that the reader is familiar enough with this style of language that the examples will be mostly self-explanatory. The saliant features of the base language are:

- A program is an *equation group*, having the following form:

  ```
  { f1(x1,x2,...,xk_1) == e1;
    f2(x1,x2,...,xk_2) == e2;
    ...
    result exp;
    ...
    fn(x1,x2,...,xk_n) == en }
  ```

  So an equation group is just a collection of *equations* that define local identifiers (**f1** though **fn**), and has a single *result clause* that expresses the value to which the equation-group will evaluate (**result** is a reserved word). Equation groups are just expressions, and may thus be nested to an arbitrary depth.

- Equations are mutually-recursive, are evaluated "by demand," and thus their order is irrelevant. A double equal-sign ("==") is used in equations to distinguish it from the infix operator for equality. Definitions of simple values, such as **x == exp**, can be viewed simply as nullary functions.

- A conditional expression has the form: "**pred -> cons, alt**" and is equivalent to the more conventional "**if pred then cons else alt**."

- Lists are constructed "lazily." The symbols "^" and "^^" are infix operators for cons and append, respectively, and **hd** ("head") and **tl** ("tail") are like car and cdr, respectively, in Lisp. A proper list may also be constructed using brackets, as in **[a,b,c]** (which is equivalent to **a^b^c^[]**).

- ParAlfl has a *pattern-matcher* through which complex functions may be defined more easily. For example, the function **member** may be defined by:

  ```
  member(x,[]) == false;
        ' (x,x^L) == true;
        ' (x,y^L) == member(x,L);
  ```

  When using the pattern-matcher note that the order of equations defining the *same* function *does* matter. Also note the use of a single quote ' as a shorthand for the function name in consecutive equations defining the same function.

- ParAlfl also has functional vectors and arrays. The primitive function call **mka(n,f)** creates a vector **v** of **n** values, indexed from **0** to **n-1**, such that its **i**th element **v[i]** is the same as **f(i)**. As a generalization, the expression **mka(d1,d2,...,dn,f)** returns an **n**-dimensional array **a** such that **a[x1,...,xn]** = **f(x1,...,xn)**.[2] Arrays are *strict*, and for that reason we assume that all of their elements are computed in parallel (resources permitting).

In the following two subsections we present the syntax and intuitive operational semantics for the extensions to the base language: *mapped expressions* and *eager expressions*. We return to a formal denotational description of ParAlfl in Section 5.

2.2. Mapped Expressions

As mentioned in Section 1, our primary goal is to allow one to map the evaluation of a program onto a particular multiprocessor architecture. We accomplish this by using *mapped expressions*, which have the simple form:

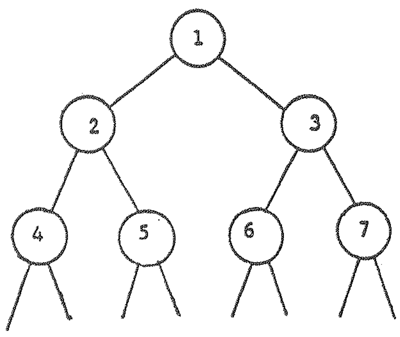
exp $on proc

which intuitively declares that **exp** is to be computed on the processor identified by **proc** (we prefix **on** with $ to emphasize the fact that **$on proc** is an annotation). The expression **exp** is the *body* of the mapped expression, and represents the value to which the overall expression will evaluate (and thus can be any valid ParAlfl expression, including another mapped expression). The expression **proc** must evaluate to a processor id, or *pid*. We will assume, without loss of generality, that processor ids are *integers*, and that there is some pre-defined mapping from those integers to the physical processors they denote. For example, a tree of processors might be numbered as shown in Figure 2-1a, or a mesh as shown in Figure 2-1b. The advantage of using integers is that the user may manipulate them using conventional arithmetic primitives; for example Figure

[1]In order to make ParAlfl more familiar to a broader audience, the base language as presented below is actually somewhat different from ALFL, in that, for example, function application is expressed using "tupled" arguments rather than "currying."

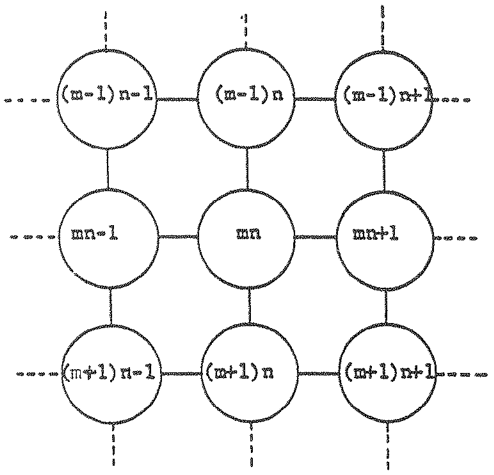
[2]Note that vectors and arrays defined in this way look very much like functions. Indeed, mka can be viewed simply as a caching functional!

2-1 also defines functions that map pids to neighboring pids.[3]



```
left(pid) == 2*pid;
right(pid) == 2*pid+1;
parent(pid) == pid/2;
```

(a) "Infinite" tree



```
up(pid)   == pid<=n -> error, pid-n;
down(pid) == pid>(m-1)*n -> error, pid+n;
left(pid) == (pid-1)\n=0 -> error, pid-1;
right(pid) == pid\n=0 -> error, pid+1;
```

Note: x\y returns the remainder of x divided by y.

(b) Finite Mesh

Figure 2-1: Two possible network topologies

To facilitate the use of mapped expressions, we also introduce a way for the user to access the *currently executing processor*. This is a *dynamic* notion, since a recursive expression, for example, might be evaluated on different processors, depending on the depth of the recursion. We provide this access through the reserved identifier **$self** which when evaluated returns the pid of the currently executing processor. It should be pointed out that since **$self** is essentially a "dynamic variable," its evaluation violates the normal notion of *referential transparency* since it will have different values depending on where and when in the program it is evaluated! This seemingly unfortunate state of affairs, however, may be remedied by imposing the (reasonable) syntactic constraint that the identifier **$self** cannot appear other than in the pid component of a mapped expression. Since the value of a mapped expression **exp $on pid** is **exp**, then with the additional constraint that all pid expressions must terminate, it is easy to see that *the value of a program cannot change, no matter how many expressions are mapped*. This simple determinacy property is a very important one, since it allows a program to be written and debugged on a uniprocessor without the annotations, and then executed on a parallel system by adding the annotations for better performance.

[3]A safer discipline might be to define a pid as a unique type, and to provide primitive functions to manipulate values having that type.

It should also be noted that the information provided by **$self** is philosophically no different from providing the depth of the current execution stack, or the value of the current program counter, or the register containing the value of a certain variable, or any other arbitrary implementation-dependent parameter. It's just that the currently executing processor can be used to great advantage, as the examples in the next section will demonstrate. From a semantical viewpoint, just as the meaning of an expression is normally given as a function of a "current continuation" and "current environment," the operational meaning that we are trying to convey is given as a function of a "currently executing processor." And just as in Scheme [11] one is given access to the current continuation via calls to the primitive function **call/cc**, we are providing access to the currently executing processor via the dynamic variable **$self**.

Simple Examples of Mapped Expressions

As a simple example of the use of mapped expressions, consider the program fragment **f(x)+g(y)**. The semantics of the + operator allows the two subexpressions to be executed in parallel. If we wish to express precisely *where* the subcomputations are to occur, we may do so by annotating the expression, as in:

```
(f(x) $on 0) + (g(y) $on 1)
```

where **0** and **1** are processor ids. Of course, this static mapping is not very interesting, but by using the dynamic variable **$self** we can be more creative. For example, suppose we have a mesh or tree of processors such as in Figure 2-1. We may then write:

```
(f(x) $on left($self)) + (g(y) $on right($self))
```

to denote the computation of the subexpressions in parallel, with the sum being computed on **$self**.

Similar mappings can be made from composite objects such as vectors and arrays to specific multiprocessor configurations. For example, if **f** is defined by **f(i) == i**2 $on i**, then the call **mka(n,f)** will produce a vector of squares, one on each of the processors, such that the **i**th processor contains the **i**th element (namely i^2). Further suppose we have two vectors **v** and **w** (which may be distributed as above, but are not required to be), and we wish to create a third being the sum of the other two, but distributed over the **n** processors. This can be done very simply by: **mka(n,g)** where **g(i) == (v[i]+w[i]) $on i**.

Comment on Lexical Scoping and Data Movement

It is important to note that *no* communications primitives and *no* special synchronization constructs are needed in the above examples to "move data" from one processor to another. Together with the annotations, it is accomplished simply through the use of normal lexical scoping mechanisms. For example, consider the program fragment:

```
{ x == exp $on p;
  result ...
        (...x...) $on q
        ...
        (...x...) $on r
}
```

The value of **x** is created on processor **p**, and the two references to **x** essentially cause that value to be "sent" to processors **q** and **r**. Since there are no side-effects, there is no need to synchronize the concurrent accesses to **x**'s value. If "finer" control over **x**'s movement is desired, intervening references can be made, as in:

```
{ x == exp $on p;
  result ...
         { x == x $on s;
           result ...
                  (...x...) $on q
                  ...
                  (...x...) $on r }
}
```

which essentially moves x to processor s prior to its access from processors q and r.

We feel that the environments created through lexical scoping fit naturally into the parallel computing world, an idea supported by all of the examples given in Section 3. It is a simple yet powerful way to express communication in a parallel computer, and eliminates the need for special message-passing constructs as is used in almost every other language for parallel computing.

2.3. Eager Expressions

The second form of annotation arises out of the occasional need for the programmer to "override" the lazy evaluation strategy of ParAlfl, which normally does not evaluate an expression until it is absolutely necessary. Opportunities to override lazy evaluation can often be inferred by a suitable "strictness analysis" [5, 20, 22, 32], but there are cases where such an analysis will fail, and in any case the programmer may wish to make such inferences explicit in the source program. The annotation that we propose would not be needed in a functional language with call-by-value semantics, such as pure Lisp, but we prefer the greater expressiveness afforded by call-by-need semantics. We thus introduce an *eager expression*, which has the simple form:

```
        #exp
```

and intuitively forces the evaluation of **exp** *in parallel* with its "most relevant" surrounding syntactic form, as defined below: If **#exp** appears as:

1. an argument in a function call, then it executes in parallel with the function call; e.g. **f(x,#y,z)**.
2. an element of a list, then it executes in parallel with the list; e.g. **[x, #y, z]**.
3. an arm of a conditional, then it executes in parallel with the conditional; e.g. **p -> #x, y**.
4. an operand of an infix operator, then it executes in parallel with the whole operation; e.g. **x^#y** or **x&#y**.

Thus, for example, in the expression **p -> f(#x,y), z**, the evaluation of **x** begins as soon as **p** has been determined to be **true**, and simultaneously the function **f** is invoked on its two arguments. Note that the evaluation of *some* subexpression begins when *any* expression is evaluated, and thus to eagerly evaluate that subexpression accomplishes nothing. For example, note the following equivalences:

```
#p -> x, y ≡ p -> x, y
#x & y ≡ x & y
(#f)(x,y) ≡ f(x,y)
```

Similarly, there is no need to eagerly evaluate either argument to a strict binary operator such as +, since in ParAlfl it is assumed that both arguments may be evaluated in parallel.

The *value* of an expression containing an eager subexpression is the same as if the expression had no annotation at all. Thus, as with mapped expressions, the annotation only adds an *operational* semantics, and means that the user may invoke a non-terminating computation yet have the overall program terminate. For example, consider:

```
{ f(x) == 1;
  g(y) == g(y+1);
  result f(#g(0)) }
```

The call **g(0)** will be invoked eagerly and will not terminate, but since **f** does not depend on the value of its argument, the program will terminate and return the value **1**. The process that is computing **g(0)** is often called an *irrelevant task* (once its value has been determined to be no longer needed). There exist strategies for finding and deleting irrelevant tasks at run-time [3, 13, 14, 15], but such mechanisms are beyond the scope of this paper. Suffice it to say that given such a "task collection" mechanism there are real situations where one might wish to exercise the option of invoking a non-terminating computation. An example of this is given in Section 3.2.

Simple Examples of Eager Expressions

Together with the conditional, we can define a simple function to evaluate two expressions in parallel:

```
par(a,b) == false -> #a, #b
```

Thus **par(a,b)** will return the value **b**, but **a** is eagerly evaluated in parallel -- note that **par(a,b)** terminates even if **a** doesn't. This function is useful in eagerly starting the evaluation of a subexpression that the programmer either *knows* will eventually be needed, or just *conjectures* will eventually be needed and is willing to allocate the resources to compute it. Another use is given in the example below.

To force the computation of all elements of a "lazy" list, we can use the following function:

```
strong-force(l) == { result sf(l);
                     sf([]) == l;
                     sf(a^lst) == par(a,sf(lst)) }
```

This function actually evaluates *each element* of the list. If all we were interested in was *expanding* the list, but not computing the individual elements, we could define the "less forceful" function:

```
weak-force(l) == { result wf(l);
                   wf([]) == l;
                   wf(a^lst) == wf(lst) }
```

which uses no annotations at all! Unfortunately, both of these functions tend to "strictify" lists; that is, the end of the list must be reached before anything is returned, which might limit parallelism at a higher level in the program, and the functions will not terminate when applied to an infinite list. This can be remedied by changing the result clause in the functions **strong-force** and **weak-force** to: **result par(sf(l),l)** and **result par(wf(l),l)**, respectively.

2.4. Notes on Determinacy

All ParAlfl programs possess the following determinacy property, which we state as a theorem:

> **Theorem 1:** (Informal) A ParAlfl program in which (1) the identifier **$self** appears only in pid expressions and (2) all pid expressions terminate, is functionally equivalent to the same program with all annotations removed. That is, both programs return the same value.

The reason for the first constraint was discussed earlier: **$self** can return different values depending on the mapping strategy used. The purpose of the second constraint should be obvious: if the system diverges when determining on which processor to execute the body of a mapped expression, then it will never get around to computing the value of that expression. We postpone a formal statement and proof of this theorem until a formal denotational semantics for ParAlfl is given in Section 5, at which point the proof becomes trivial.

Although neither determinacy constraint is severe, there are practical reasons for wanting to violate the first one; i.e., for wanting to use the value of **$self** in other than a pid expression. The most typical situation where this arises is in a non-isotropic topology where certain processors form a "boundary" for the network. For example, the leaf processors in a tree, or the edge processors in a mesh. There are many distributed algorithms whose behavior at such boundaries is different from their behavior at internal nodes. To express this, one needs to know when execution is occurring at the boundary of the network, which can be conveniently determined by analyzing the value of **$self**. An example of this is given in Section 3.1.

One final note on determinacy concerns the use of #. Although this annotation does not affect the program's functional behavior, there may be situations where one wishes to make an expression "strict" in one of its subexpressions. Unfortunately this can prevent a program from terminating when otherwise it would, thus changing its functional semantics. Our approach to providing this capability is therefore not through annotations, but through a primitive function that induces the strictness property. We define the (monotonic) predicate **terminate?** such that **terminate?(exp)** returns **true** if **exp** terminates, otherwise it does not terminate either. We can then define:

strictify(e1,e2) == terminate?(e1) -> e2, e2

so that, for example, **strictify(x,f(x,y,z))** will essentially make the call to **f** strict in **x** (and, as an aside, will cause the evaluation of **x** *before* that of **f(x,y,z)**). This definition depends on the fact that in ParAlfl, $(\perp$ -> x,x) == $\perp$ (where $\perp$ is the symbol used in semantics to represent the value of a non-terminating computation).[4]

3. Sample Application Programs

It has been our experience in looking at many multiprocessing algorithms that they are quite functional in their global behavior, perhaps because their parallel nature precludes dependencies on a centralized shared memory, and thus they fall nicely into the functional model. In this section we present four examples that demonstrate the key aspects of the proposed paradigm. We urge the reader to note how few annotations are needed to accomplish the desired mappings.

3.1. Parallel Factorial

Figure 3-1 shows a simple "parallel factorial" program, annotated for execution on a finite binary tree of n = 2^d processors. Although computing factorial, even in parallel, is a rather simple task, the example demonstrates several important ideas, and most other "divide and conquer" algorithms could easily fit into the same framework.

```
{ result pfac(1,k) $on root;

  pfac(lo,hi) ==
        lo=hi -> lo,
        lo=(hi-1) -> lo*hi,
        { result (pfac(lo,mid) $on left($self)) *
                 (pfac(mid+1,hi) $on right($self));
          mid == (lo+hi)/2 };

  left(pe) == (2*pe > n) -> pe, 2*pe;
  right(pe) == (2*pe > n) -> pe, 2*pe+1;
  root == 1;
}
```

Figure 3-1: Divide-and-conquer factorial

[4]It may seem as if terminate? need not be primitive if at least one other type-predicate is primitive. For example, suppose there is a predicate integer? defined in the normal way. Then it seems as if the predicate terminate? could be defined by:

terminate?(exp) == integer?(exp) -> true, true

Semantically speaking, this version of terminate? has the property that terminate?(x) == $\perp$ if x=$\perp$, otherwise true. Although this is also true of the primitive version of terminate? described earlier, it does not fully capture the intended operational semantics, because with such a definition it might be possible for a compiler to infer that an expression will be an integer, thus constant-folding the overall expression to true! Our intent is more operational -- i.e., we want exp to actually be evaluated (if it hasn't been already), and return true only if it terminates. Thus terminate? must be primitive so that the compiler can decide if certain transformations are applicable (in particular, it cannot reduce terminate?(exp) to true even if it can infer that exp will terminate).

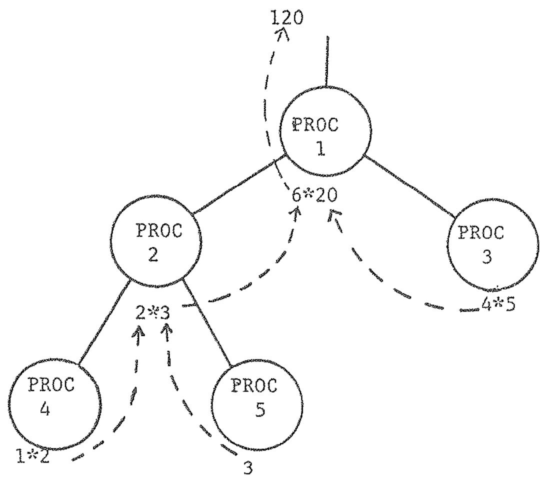


Figure 3-2: Dataflow for parallel factorial

The algorithm is based on splitting the computation into two parts at each iteration, and mapping the two subtasks onto the "children" of the current processor. Note that through the normal lexical scoping rules, **mid** will be computed on the "current" processor and passed to the child processors as needed (recall the discussion in Section 2.2). The functions **left**, **right**, and **root** describe the network mapping necessary for this topology, and Figure 3-2 shows the process mapping and flow of data between processes when k=5.

It should be noted that this program observes the constraints stated in Theorem 1, and thus it is deterministic regardless of the annotations. Note further that with the mapping used, when processing reaches a leaf node all further calls to **pfac** are executed on the leaf processor. In practice, more complex routing functions could be devised that, for example, might reflect the computation upward when a leaf processor is reached. Alternatively, it may be desirable to use a more efficient factorial algorithm at the leaf nodes. An example of this is given in Figure 3-3, where the tail-recursive function **sfac** is invoked at the leaves. Determining that execution has reached a leaf processor requires inspection of **$self**, and thus the constraints of Theorem 1 are violated, yet *the program is still deterministic*. This is, of course, usually the case, but we cannot guarantee it in the general case without the previously discussed constraints.

```
{ result pfac(1,k) $on root;

  pfac(lo,hi) ==
        lo=hi -> lo,
        lo=(hi-1) -> lo*hi,
        { result leaf?($self) -> sfac(lo,hi,1),
                 (pfac(lo,mid) $on left($self)) *
                 (pfac(mid+1,hi) $on right($self));
          mid == (lo+hi)/2 };

  sfac(lo,hi,acc) == lo=hi -> lo*acc,
                     sfac(lo+1,hi,lo*acc);

  leaf?(pe) == pe >= 2**(d-1);
  left(pe) == 2*pe;
  right(pe) == 2*pe+1;
  root == 1;
}
```

Figure 3-3: Factorial, with unique behavior at leaves

3.2. A Prime Number Generator

Figure 3-4 shows a program to compute the first n prime numbers, using the well-known "sieve of Erastothenes." Ignoring the annotations for a moment, this program demonstrates a classical use of "infinite lists." First the infinite list of integers starting with 2 is generated (**ints**). The function **sift** then removes the first element, which it takes as prime, and elides multiples of that prime from the remaining list (by calling **filter**). This "filtered" list is then passed recursively to **sift** to form the rest of the infinite list of primes. The primitive function call **prefix n primes** then selects the first n elements from the result.

```
{ result prefix(n,primes);

  primes == sift(ints);

  sift(p^rest) ==
      { result p^(sift(filter(rest)) $on right($self));
  filter(n^l) == n\p=0 -> filter(l), n^filter(l) };

  ints == numsfrom(2);
  numsfrom(n) == n^numsfrom(n+1);
}
```

Figure 3-4: Sieve of Erastothenes on infinite vector

With the single mapping annotation shown, the subsequent calls to **sift** are mapped onto successive processors to the "right" of the current one. We will ignore the details of the function **right** that accomplishes this, recognizing that the conceptually infinite vector of processors could be a ring, "twisted torus," hypercube, or other topology. However, there is something terribly wrong with this program: It has no parallelism! Remember that lists are computed lazily, and thus there are no strict operators that could create parallelism, other than those in the trivial subexpression **n\p=0**.

To fix this problem, one should first note that the function **filter** is essentially doing all of the work -- it must check successive elements of **rest** until it finds one that is not a multiple of p. To get parallelism one needs to have several invocations of filter operating together, one "feeding" the next in an effectively pipelined manner. However, there is no *simple* way to call **filter** and have it return just enough elements to satisfy subsequent calls to **filter** so that exactly **n** primes are generated. The simplest solution is to make **filter** behave eagerly, extract the first **n** primes from the result eagerly, and let the system's task manager "kill off" the then irrelevant processes computing the filtered streams (recall the discussion in Section 3.2). This solution is shown in Figure 3-5 -- note the eager call to **filter** and the redefinition of **prefix** to evaluate its elements eagerly (the other two calls to **filter** could also be made eagerly, but little additional parallelism would be attained, since they get evaluated "immediately" anyway).

```
{ result prefix(n,primes);

  primes == sift(ints);

  sift(p^rest) ==
      { result p^(sift(filter(rest)) $on right($self));
  filter(n^l) == n\p=0 -> filter(l), n^#filter(l) };

  ints == numsfrom(2);
  numsfrom(n) == n^numsfrom(n+1);

  prefix(0, l) == [];
     ' (n,a^l) == a^#prefix(n-1,l)
}
```

Figure 3-5: Parallel sieve

As an aside, we should point out that this example demonstrates an important use of, and need for, an automatic task deletion mechanism as discussed in Section 2.3. In a "conventional" multiprocessing environment one might instead have a global variable that could be set after the **n** primes were extracted, which could serve as a signal for the **filter** processes to "kill themselves." This is not as clean a solution for several reasons: First, it requires the explicit coding of the termination condition into the function **filter**. Second, providing such explicit mechanisms introduces the possibility of programmer error, either terminating a process too soon or failing to terminate one that should. Finally, such a centralized signal will behave as a bottleneck to effective parallelism. Providing an automatic task deletion mechanism in a parallel system is akin to providing automatic garbage collection in a sequential system -- it frees the programmer from the need to explicitly deallocate objects, whether they be processes or cons cells.

It is interesting to note that there is yet another useful parallel version of this program. Consider the following request from a client: create a lazy, infinite list, but every time an element is "demanded" from the list, return the element and simultaneously compute the next element in parallel so that it will be immediately available upon the *next* demand. Surprisingly, this seemingly complex request can be filled simply by eagerly evaluating **filter(rest)** when **sift** is called, thus computing the next non-multiple of p in advance. This is accomplished in Figure 3-6 through the use of the **par** function defined earlier.[5] It is not hard to imagine generalizing this technique so that a "buffer" of n primes is maintained, ready to be "consumed" by some other process.

If nothing else, these examples of prime number generators amply demonstrate that parallelism in a program is not an "all-or-nothing" proposition. There are many subtle degrees of parallelism and mapping possibilities that the programmer may wish to express. In a conventional language augmented for parallel computing, expressing these alternatives typically involves changing the program's fundamental functional behavior. We believe that the functional and operational behaviors should be kept separate, and that the para-functional approach accomplishes this separation quite well. One can control not only the mapping of program to machine, but also the degree of parallelism manifested in the evaluation strategy, without jeopardizing the functional correctness of the program.

```
{ result sift(ints);

  sift(p^rest) ==
      { new == filter(rest);
  result par( new, p^(sift(new) $on right($self)) );
  filter(n^l) == n\p=0 -> filter(l), n^filter(l) };

  ints == numsfrom(2);
  numsfrom(n) == n^numsfrom(n+1);
}
```

Figure 3-6: Sieve using "piece-meal" parallelism

3.3. Solution to Upper Triangular Block Matrix

The next example is typical of those found in scientific computing: the problem is to solve for the vector **x** in the equation **Ux=b**, where **U** is an upper triangular block matrix. We wish to use an algorithm designed for efficient execution on a *ring* of **n** processors. Although the topology of a ring is rather simple, its limited interprocessor communications make it rather difficult to use effectively, and thus algorithms designed for it are often rather complex, making them a good test-bed for para-functional programming.

We assume that the processors are labeled consecutatively around the ring, from **0** through **n-1**. We further assume that the vector

[5] For an alternative (and simpler) way to accomplish this annotation, see section 4.6.

length is **n**, and that each row of **U** and the corresponding elements of **b** are distributed uniformly across the ring. We wish the solution vector **x** to be distributed as well.

```
Solve $U_{n-1,n-1}x_{n-1}=b_{n-1}$ on processor n-1.          (step 1)
For i=n-2 down to 0 do:
Begin For j=i down to 0 do in parallel on processor j:
        $b_j := b_j - U_{j,i+1} * x_{i+1}$.                    (step 2)
    Solve $U_{i,i}x_i=b_i$ on processor i.                     (step 3)
End.
```

Figure 3-7: Algorithm for solving **Ux=b** on a ring

Figure 3-7 outlines a parallel "block-row" algorithm for solving Ux=b. Initially, the last element x_{n-1} of the solution vector is computed on processor **n-1** (step 1). Then, a "back-substitution" step takes place in parallel on the remaining **n-1** processors (step 2). Completing this, the next-to-the-last element of the solution (x_{n-2}) can be computed (step 3). This alternating process (steps 2 and 3) of solving and back-substituting continues until all elements of **x** are found.

A ParAlfl program for this algorithm is shown in Figure 3-8. For simplicity we assume that the blocks are of size 1 (and thus are represented simply as scalar quantities), but conceptually any size block could be used. For block size 1, the solution to $U_{i,i}x_i=b_i$ is simply $x_i= b_i/U_{i,i}$, and is captured by the function **uxb**.

```
{ result iter(n-2,[lastx],b) $on n-2;

  lastx == uxb(u[n-1,n-1], b[n-1]) $on n-1;
  uxb(u,b) == b/u;

  iter(0,x,b) == x;
   ' (i,x,b) ==
    { result iter(i-1, sol^x, newb) $on i-1;
      newb == mka(i,backsub);
      backsub(j) == b[j] - U[j,i+1]*hd(x) $on j;
      sol == uxb(U[i,i], newb[i]) };
}
```

Figure 3-8: Program for Ux=b, annotated for ring

The solution vector is represented as a list, since it is computed element-by-element with intervening back-substitution steps. The last element, and first to be computed, is **lastx** (step 1 of the previous description). The function **iter** accomplishes the iteration of steps 2 and 3, and collects the elements of the solution in the parameter **x**. Note how **iter** recursively performs the back-substitution by creating a new version of **b** (**newb**) on each iteration. This vector decreases in length by 1 on each iteration, and is computed in parallel as the previous description of the algorithm suggests (indeed, all of the parallelism in the program is manifested here). It is then used to compute **sol**, the next element of the solution vector. Figure 3-9 shows the temporal progression of the program, alternating between computing a new element of the solution and doing a back-substitution. Synchronization is achieved automatically by the "call-by-need" evaluation strategy.

Note finally that if true *block* matrices and vectors are used, the only change necessary is the redefinition of the functions **uxb** and **backsub**.

Suppose now that one wishes to run this program on a hypercube. One way to accomplish this would be to "simulate" a ring in a hypercube by some suitable embedding. One such embedding is the reflected grey-code, captured by the following ParAlfl code:

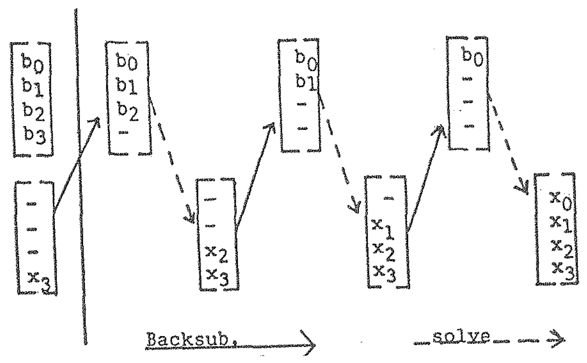


Figure 3-9: Time sequence of program to solve Ux=b

```
ringtocube(i) == v[i];
v == mka(n,graycode);
graycode(i) == i<2 -> i,
                  { result v[2*mid-i-1] + mid;
                    mid == 2**log2(i) }
```

where **log2(i)** returns the base-2 logarithm of **i**, rounded down to the nearest integer. For example, Figure 3-10 shows the embedding of a ring of size 8 into a 3-cube.

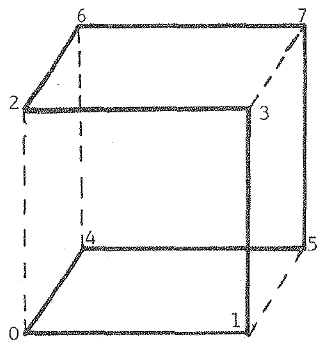


Figure 3-10: Embedding of ring of size 8 into 3-cube

If we then replace each pid expression **exp** in Figure 3-8 with **ringtocube(exp)**, we arrive at the desired embedding. Note that the code for the algorithm itself did not change at all, just the annotations. Of course, a more efficient algorithm for the hypercube might exist, and would naturally require recoding of the main functions.

3.4. Jacobi's Method for Solving PDE's

The next example solves Laplace's partial differential equation $u_{xx}+u_{yy} = 0$ on a grid, where the grid edges have fixed initial conditions. Jacobi's method is one in which the grid is iteratively refined until a certain convergence criterion has been met. Each element of the refined grid is computed by taking the average of the four adjacent neighbors to the corresponding element in the previous grid. For this example we assume that the convergence criterion is given by a maximal allowable error **maxerr**.

Ignoring the annotations for a moment, the ParAlfl program in Figure 3-11 carries out this basic algorithm. The initial grid with fixed boundary conditions is **igrid**. The function **solve** computes a new grid from the current one, using **newval** to compute individual elements. Note that in computing elements for the new grid, the boundary elements are not changed. The function **findmaxerr** computes the new error by finding the maximum of the errors at individual elements; each individual error is just the absolute difference between the old value and the new. Once that maximum error falls below **maxerr**, the computation is complete.

Now let's consider mapping this program for parallel execution on a *mesh* of processors, a fairly natural topology for this algorithm. Clearly, for an mXn grid, up to m*n processors could be executed in parallel on each iteration. The annotation in Figure 3-11 accomplishes this using the function **mesh**, which maps each point in the grid to the corresponding processor id in the mesh, using the numbering in Figure 2-1b.

On the other hand, if there is an insufficient number of processors, or if the communications costs are high, then it may be better to let individual elements in the grid be grids themselves (call them sub-grids), thus increasing the granularity of the computation. Suppose in such a situation that the top-level grid G is of size nXn, and the sub-grids are of size kXk. Now consider the computation of a new sub-grid S' at point (i,j) in G. Then to compute an element, for example, on the "top" row of S' requires looking at the sub-grid on processor (i-1,j). More formally, the ParAlfl expression for the new element S'[0,c] is:

```
(S[0,c-1]+S[0,c+1]+S[1,c]+(G[i-1,j])[k-1,c])/4
```

where S == G[i,j]. Given this start, it should not be hard for the reader to revise the program in Figure 3-11 so that this coarser granularity is achieved.

```
{ Result solve(#igrid);

  solve(grid) ==
   { result newerr<maxerr -> newgrid,
                              solve(newgrid);

     newgrid == mka(m,n,newval);
     newval(i,j) == ( i=m-1 | j=n-1 -> grid[i,j],
                     ((grid[i-1,j] + grid[i+1,j] +
                       grid[i,j-1] + grid[i,j+1])/4) )
                     $on mesh(i,j);
     newerr ==
       { result finderr(1,1,0.0);
         finderr(i,j,lerr) ==
            i=m-1 -> lerr,
            j=n-1 -> finderr(i+1,1,lerr),
            { result finderr(i,j+1,max(currerr,lerr));
              currerr == abs(newgrid[i,j]-grid[i,j]) }
   } }
  mesh(row,col) == row*m + col
}
```

Figure 3-11: Program for Jacobi's method -- grid size: mXn, initial grid: **igrid** maximum error: **maxerr**

We should point out that one cannot map this algorithm very easily using "lists of lists" to represent the grid, since it is difficult to express both "up-down" and "left-right" communication with such an approach. This is not true of all array problems; for example matrix multiplication can be done fairly easily using lists of lists, since the inner products only require communication in two of the four directions [35].

4. Extensions

Once the basic para-functional programming methodology is understood, many variations of it become apparent, and alternative language design features present themselves. We discuss several of them below.

4.1. Alternative Reflections

The dynamic variable **$self** is one of many that could be provided to the programmer. Consider, for example, the use of the dynamic variable **$load** to give the programmer a measure of the "processing load" on the currently executing processor (perhaps a count of the number of tasks waiting in the process queue). Similarly, **$memory-utilization** might provide information about how much memory ("heap space") is left on a node. Treated as functions, these alternatives allow querying the load or memory-utilization of other processors, in particular neighbors, as in **$load(left($self))**. With this capability a programmer could implement a dynamic load-balancing strategy tailored for a particular application. Another useful hook might be **$number-of-processors**.

4.2. Mapping to Operating System Resources

In addition to allowing one to map processes to processors, one could allow mapping data objects to storage devices or I/O channels. For example, the expression **lst $on tape-drive-2** might cause the list **lst** to be written out to a particular tape drive. In a functional setting this is quite useful, since traditionally a purely functional program simply returns the single top-level value to one place (usually the terminal display). Suppose, for example, one wishes **msg** to appear on the terminal, while writing **stream** into a file called **out**. The top-level result might look like:

```
result strong-force([msg $on std-terminal,
                     stream $on file("out")])
```

Another use might be in a heterogeneous environment where certain specialized processors exist. For example, the expression **fft(a) $on sig-proc(1)** might map an FFT computation to a particular signal processor.

4.3. An Inverted Viewpoint of Mapping

It may also be desirable to perform an operation on a data structure *at the processor on which the data structure resides*. For example, the special function **$location** could be used to return the pid of its argument's "home." Thus the expression **f(obj) $on $location(obj)** could be used to apply the function **f** to **obj** on whichever processor **obj** happens to reside.

4.4. Automatic Scheduling

One of the more standard approaches to obtaining parallelism from a functional program is to allow the compiler and run-time system to automatically decompose and dynamically distribute the program for execution on a suitable multiprocessor [18, 19, 28]. An obvious compromise is to combine this automatic approach with the more explicit para-functional approach. For example, the annotated expression **exp $on $choice** might indicate that **exp** is to be decomposed and distributed as the system sees fit, but that subexpressions internal to it may still be mapped using the para-functional approach when needed.

4.5. Non-Deterministic Applications

Many sceptics of functional programming point out that functional languages cannot deal with non-determinism because of their lack of an assignment operator. This is misleading, first of all because non-determinism does not require assignment (although it can arise *because* of assignment), and second because non-determinism *can* be effectively introduced into functional languages by a suitable set of primitive operators. The difference in the result is then manifested as a functional language's insistence that the non-determinism be made *explicit* rather than *implicit* as in conventional languages. The argument as to which is better then becomes for the most part an argument about style.

These issues are beyond the scope of this paper, but are mentioned here because para-functional programming is in no way in conflict with the introduction of non-determinism, and should work well with most approaches to non-determinism in functional language research. The right combination of annotations and non-deterministic operators could indeed provide a powerful set of tools to build a "purely functional" operating system.

4.6. Alternative Syntax

Our language design choices were somewhat arbitrary, and thus other choices may be more suitable for particular applications. One of the more obvious examples of this is in the use of # -- currently this annotation "parallelizes" the subexpression relative to, more or less, the most immediately surrounding syntactic form. Another possibility would be to parallelize the subexpression relative to the innermost surrounding *equation body* or *result clause*. Suppose that the annotation ! is used for this purpose, and consider the function **sift** defined in Section 3.2 (Figure 3-6):

```
sift(p^rest) ==
{ new == filter(rest);
  result par(new,p^(sift(new) $on right($self)));
  filter(n^l) == n\p=0 -> filter(l), n^filter(l) }
```

With ! this could be rewritten as:

```
sift(p^rest) ==
{ result p^sift(!filter(rest)) $on right($self);
  filter(n^l) == n\p=0 -> filter(l), n^filter(l) }
```

At least for this example, the use of ! is much simpler and clearer.

5. A Formal Semantics for ParAlfl

Hopefully the reader at this point has a good intuitive feel for the operational semantics of ParAlfl. However, we have glossed over many non-trivial details. For example: (1) In a mapped expression, where is the pid expression evaluated? (2) Where is an unmapped expression evaluated that appears at the top level? (3) The identifier $self in a function body is evaluated when the function is *applied*, not when it is created, but what exactly does that mean? One approach to providing these details is to anticipate as many of the above questions as possible, and answer each of them in turn. This informal approach is unfortunately error-prone and susceptible to ambiguity because of the English language. A better approach is to give a formal denotational semantics that captures the desired operational properties, and that is what we do.

For the following we assume that the reader is familar with the standard notational conventions for denotational semantics, such as outlined in [12]. For clarity we also assume that the following two syntactic transformations have occurred on the ParAlfl source program:

First, the equations defining functions, including pattern-matching, have been mapped into "curried" lambda expressions. For example:

```
fac(0,acc) == acc;
 ' (n,acc) == fac(n-1,n*acc);
```

will be converted into:

```
fac == λn. λacc. n=0->acc,
                 fac (n-1) (n*acc)
```

Second, we assume that lists have been converted into an equivalent infix cons form. For example, **[x,y,z]** is converted into **x^y^z^[]**.

5.1. Standard Interpretation

To give a formal operational semantics for ParAlfl we first need to define its standard functional semantics, which after the above syntactic transformations is relatively straightforward. It is captured by the semantic function **E: Exp->Env->D**, where **Exp** is the syntactic domain of ParAlfl expressions, **Env** is the domain of environments that map identifiers to values, and **D** is the standard domain of expressible values. For convenience we will omit the details of the syntactic and semantic domains, as well as errors; this will allow us to concentrate on the issues of mapping. We adopt the normal convention of enclosing syntactic objects in double-brackets, and thus $E[\![exp]\!]e$ denotes the value of **exp** in the environment **e**, and is defined by:

$E[\![constant]\!]e = B[\![constant]\!]$

where **B** maps constants to semantic values.

$E[\![id]\!]e = e[\![id]\!]$

$E[\![unop\ e1]\!]e = B[\![unop]\!]\ (E[\![e1]\!]e)$

$E[\![e1\ binop\ e2]\!]e =$
$\quad B[\![binop]\!]\ (E[\![e1]\!]e)\ (E[\![e2]\!]e)$

$E[\![pred\text{->}con,alt]\!]e =$
$\quad \text{if } (E[\![pred]\!]e) \text{ then } (E[\![con]\!]e)$
$\qquad\qquad\qquad \text{else } (E[\![alt]\!]e)$

$E[\![\lambda x.exp]\!]e = \lambda\hat{x}.\ E[\![exp]\!]\ e[\hat{x}/x]$

$E[\![e1\ e2]\!]e = (E[\![e1]\!]e)\ (E[\![e2]\!]e)$

$E[\![\{\ fi == ei\ \ldots$
$\quad result\ exp\ \}]\!]e = E[\![exp]\!]e'$
$\quad \text{whererec } e' = e[\ldots E[\![ei]\!]e'/fi \ldots]$

$E[\![exp\ \$on\ pid]\!]e =$
$\quad \text{if } (E[\![pid]\!]e){=}\perp \text{ then } \perp$
$\quad \text{else } (E[\![exp]\!]\ e[E[\![pid]\!]e/\$self])$

$E[\![\#exp]\!]e = E[\![exp]\!]e$

The semantics of a complete ParAlfl program (which must be either an equation-group or a mapped expression whose body is an equation group) is given by the semantic function **Ep**, defined by:

$Ep[\![program]\!] = E[\![program]\!]\text{initial-env}$

where **initial-env** contains bindings for primitive functions such as **plus**, **and**, etc., a default value for $self, and constants such as **true** and **false**.

5.2. Determinacy Theorem Revisited

We return now to Theorem 1, which we restate formally below:

Theorem 1: (Formal) Let **P** be a ParAlfl program in which (1) the identifier **$self** appears only in pid expressions and (2) for each pid expression **pid** in environment **e**, $E[\![pid]\!]e \neq \perp$. Further, let **P'** be the same ParAlfl program but with all occurrrences of mapped expressions of form **exp $on pid** replaced with the body **exp**, and all occurrences of eager expressions of form **#exp** replaced with **exp**. Then $Ep[\![P]\!]{=}Ep[\![P']\!]$.

Proof. We must prove that $E[\![P]\!]\text{initial-env} = E[\![P']\!]\text{initial-env}$. We do this by structural induction on **P**. Consider any subexpression **sexp** and its "stripped" version **sexp'**. For all but mapped expressions and eager expressions, it is clear that $E[\![sexp]\!]e = E[\![sexp']\!]e$, since **$self** will never be evaluated. We consider mapped expressions and eager expressions below.

Suppose **sexp** = **exp $on pid** and thus **sexp'** = **exp**. Then:

$E[\![sexp]\!]e = \text{if } (E[\![pid]\!]e) = \perp \text{ then } \perp$
$\qquad\qquad \text{else } (E[\![exp]\!]\ e[E[\![pid]\!]e/\$self])$

But we are given that $E[\![pid]\!]e \neq \perp$, so: $E[\![sexp]\!]e = E[\![exp]\!]\ e[E[\![pid]\!]e/\$self]$. Furthermore, since **$self** will never be evaluated, we have that $E[\![sexp]\!]e = E[\![exp]\!]e = E[\![sexp']\!]e$.

The remaining case is an eager expression. Suppose **sexp** = **#exp** and thus **sexp'** = **exp**. Directly from the semantic equations for **E**, we then have that $E[\![sexp]\!]e = E[\![exp]\!]e = E[\![sexp']\!]e$. This covers all cases and thus the theorem holds. □

5.3. Execution Trees

The operational semantics that we wish to capture is a notion of "where" (i.e., on which processor) an expression will be evaluated. For each expression **exp** we associate an *execution tree* that reflects

7. Implementation Considerations and Future Work

We have concentrated in this paper on the issues of *expressing* parallel computation, in particular the mapping of programs to machines. In so doing we have left unanswered many questions about how one might *implement* a para-functional programming language. For example:

- We have implied that it would be very useful to have a system to automatically collect "irrelevant tasks." Although such systems appear tractable, none have been built to our knowledge.
- We have not discussed the details of how distributed arrays are accessed efficiently. Although we have made some progress in this area, the work is too premature to report here.
- There is a great need for optimizations to automatically reuse discarded arrays in recursive definitions. Although a fair amount of activity is current in this area [21], more work remains.

We are currently contemplating an implementation of ParAlfl on an Intel iPSC 128-node hypercube. Hopefully that experience will help answer most of the questions raised above.

8. Acknowlegements

This research has benefited from useful discussions with Bob Keller at the University of Utah, and Joe Fasel and Elizabeth Williams at Los Alamos National Laboratory. Some of the research was carried out while the second author was employed at Los Alamos.

References

1. Arvind and Kathail, V. A multiple processor data flow machine that supports generalized procedures. Proc. 8th Annual Sym. Comp. Arch., ACM -- SIGARCH 9(3), May, 1981, pp. 291-302.

2. Arvind and Gostelow, K.P. "The U-interpreter". *Computer 15*, 2 (Feb. 1982), 42-50.

3. Baker, H.G. and Hewitt, C. The incremental garbage collection of processes. AI Working Paper 149, Mass. Institute of Technology, July, 1977.

4. Beeler, M. "Beyond the baskett benchmark". *Computer Architecture News 19*, 3 (March 1984).

5. Burn, G.L., Hankin, C.L., and Abramsky, S. The theory and practice of strictness analysis for higher order functions. DoC 85/6, Imperial College of Science and Technology, Department of Computing, April, 1985.

6. Burton, F.W. "Annotations to control parallelism and reduction order in the distributed evaluation of functional programs". *ACM Trans. on Prog. Lang. and Sys. 6*, 2 (April 1984).

7. Davis, A.L. The architecture and system method of DDM-1: A recursively-structured data driven machine. Proc. Fifth Annual Symposium on Computer Architecture, 1978.

8. Davis, A.L. Data driven nets: A maximally concurrent, procedural, parallel pocess representation for distributed control systems. UUCS-78-108, Univ. of Utah, July, 1978.

9. Deminet, J. "Experience with multiprocessor algorithms". *IEEE Trans. on Computers C-41*, 4 (April 1982), 278-287.

10. Dennis, J.B., and Misunas, D.P. A preliminary architecture for a basic data-flow processor. Proc. of the 2nd Annual Symposium on Computer Architecture, ACM, IEEE, 1974, pp. 126-132.

11. Friedman, D.P. and Haynes, C.T. Constraining control. 12th ACM Sym. on Prin. of Prog. Lang., ACM, 1985, pp. 245-254.

12. Gordon, J.C.. *The Denotational Description of Programming Languages*. Springer-Verlag, New York, 1979.

13. Grit, D.H. and Page, R.L. "Deleting irrelevant tasks in an expression oriented multiprocessor system". *ACM Transactions on Programming Languages and Systems 3*, 1 (Jan. 1981), 49-59.

14. Hudak, P. and Keller, R.M. Garbage collection and task deletion in distributed applicative processing systems. Sym. on Lisp and Functional Prog., ACM, Aug., 1982, pp. 168-178.

15. Hudak, P. Distributed Task and Memory Management. Proc. of Sym. on Prin. of Dist. Comp., ACM, Aug., 1983, pp. 277-289.

16. Hudak, P. and Goldberg, B. Experiments in diffused combinator reduction. Sym. on LISP and Functional Programming, ACM, Aug., 1984, pp. 167-176.

17. Hudak, P. ALFL Reference Manual and Programmers Guide. Research Report YALEU/DCS/RR-322, Second Edition, Yale University, Oct., 1984.

18. Hudak, P. and Goldberg, B. Distributed execution of functional programs using serial combinators. Proceedings of 1985 Int'l Conf. on Parallel Proc. (and IEEE Trans. on Computers October 1985), Aug., 1985, pp. 831-839.

19. Hudak, P. and Goldberg, B. Serial combinators: "optimal" grains of parallelism. Functional Programming Languages and Computer Architecture, Sept, 1985, pp. 382-388.

20. Hudak, P. and Young, J. A set-theoretic characterization of function strictness in the Lambda Calculus. Research Report YALEU/DCS/RR-391, Yale University, June, 1985.

21. Hudak, P. and Bloss, A. The aggregate update problem in functional programming systems. 12th ACM Sym. on Prin. of Prog. Lang., ACM, 1985, pp. 300-314.

22. Hudak, P., and Young, J. Higher-order strictness analysis for untyped lambda calculus. 12th ACM Sym. on Prin. of Prog. Lang., Jan., 1986, pp. to appear.

23. Intel Corporation. IPSC User's Guide -- Preliminary. 175455-001, Intel Corporation, July, 1985.

24. Johnsson, T. The G-machine: an abstract machine for graph reduction. PMG, Dept. of Computer Science, Chalmers Univ. of Tech., Feb., 1985.

25. Keller, R.M., Lindstrom, G., and Patil, S. A loosely-coupled applicative multi-processing system. AFIPS, AFIPS, June, 1979, pp. 613-622.

26. Keller, R.M., Jayaraman, B., Rose, D., Lindstrom, G. FGL programmer's guide. AMPS Technical Memo 1, Department of Computer Science, University of Utah, July, 1980.

27. Keller, R.M. FEL programmer's guide. AMPS TR 7, University of Utah, March, 1982.

28. Keller, R.M. and Lin, F.C.H. "Simulated performance of a reduction-based multiprocessor". *IEEE Computer 17*, 7 (July 1984), 70-82.

29. Keller, R.M. and Lindstrom, G. Approaching distributed database implementations through functional programming concepts. Int'l Conf. on Distributed Systems, May, 1985.

30. Mcgraw, J.R. "The VAL language: description and analysis". *TOPLAS 4*, 1 (Jan. 1982), 44-82.

31. Milner, R. A proposal for standard ML. Sym. on LISP and Functional Programming, ACM, Aug., 1984, pp. 184-197.

32. Mycroft, A. *Abstract Interpretation and Optimizing Transformations for Applicative Programs.* Ph.D. Th., Univ. of Edinburgh, 1981.

33. Pappert, S.. *Mindstorms: Children, Computers, and Powerful Ideas.* Basic Books, 1980.

34. Seitz, C.L. "The cosmic cube". *CACM 28*, 1 (Jan. 1985).

35. Shapiro, E. Systolic programming: a paradigm of parallel processing. Dept. of Applied Mathematics CS84-21, The Weizmann Institute of Science, Aug., 1984.

36. Smith, B.C. Reflection and semantics in LISP. 11th ACM Sym. on Prin. of Prog. Lang., ACM, Jan., 1984, pp. 121-132.

37. Stoy, J.E.. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, Mass., 1977.

38. Turner, D.A. SASL language manual. University of St. Andrews, 1976.

39. Turner, D.A. Miranda: a non-strict functional language with polymorphic types. Functional Programming Languages and Computer Architecture, Sept, 1985, pp. 1-16.