Serial Combinators: "Optimal" Grains of Parallelism

Paul Hudak
Benjamin Goldberg

Yale University
Department of Computer Science
Box 2158 Yale Station
New Haven, CT 06520

Abstract

A method is described for translating a high-level functional program into combinators suitable for execution on multiprocessors with no shared memory. It is argued that the granularity of the standard set of fixed combinators is too fine, whereas the granularity of user-defined functions is too coarse. The notion of a *serial combinator* is introduced that in some sense has optimal granularity, and that takes into account pragmatic issues such as the complexity of expressions and communication costs between processors. The technique improves on the standard notion of *super-combinators* by making them larger to retain locality and improve efficiency, and by ensuring that they have no concurrent substructure that could result in lost parallelism. Simulation results demonstrate improved performance on both sequential and parallel computing models.

This research was supported in part by NSF Grant MCS-8302018 and a grant from Burroughs Corporation.

1. Introduction

The advantage of using *graph reduction* as a parallel evaluation model for functional programs is well-known. Indeed several research projects are underway with the goal of automatically decomposing and dynamically distributing functional programs for parallel execution on multiprocessor architectures. Examples include Keller's work on RediFlow [14], Peyton Jones' work on GRIP [17], Kieburtz work on the G-machine [15], and the authors' work on DAPS [5, 6]. The use of some form of *combinators* in all cases seems to be especially attractive, since they eliminate the need for an *environment*, which in the lambda calculus can behave as a bottleneck to effective parallelism.[1] Through the use of combinators the environment can be replaced by a more explicit, decentralized, argument-passing mechanism. In essence the environment becomes distributed into the program structure itself, thus simplifying execution on a distributed architecture.

Although simulation studies have demonstrated that this general approach to parallel computing is very promising, there remain many unanswered questions. In this paper we address one of particular importance, namely the *granularity of program decomposition*. It is argued that for multiprocessor architectures without shared memory, the granularity of combinators is too fine, whereas the granularity of user-defined functions is too coarse. We introduce the notion of a *serial combinator* that in some sense has optimal granularity, and that takes into account pragmatic issues such as the complexity of expressions and

[1] Although combinators are not used explicitly in RediFlow or the G-machine, the explicit importation of free variables accomplishes the same thing.

communication costs between processors. The technique improves on the notion of *super-combinators* by making them larger to retain locality and increase efficiency, and by ensuring that they have no concurrent substructure that could result in lost parallelism. Even if the latter improvement is ignored, the resulting combinators still show improved performance over super-combinators, on both sequential and parallel machines.

For the remainder of the paper we assume the reader to be familiar with the general principles underlying functional languages and graph reduction, although a brief overview of combinators and super-combinators is given in the next section. A more thorough discussion may be found in [1] and [3].

2. A Review of Combinators

Consider a function **f** defined by $\mathbf{f(x) = exp}$. A functional object equivalent to **f** may be obtained by *abstraction* of the free variable **x** from **exp**, which is written as **[x]exp**. Application of this function to a value **v** is written as **([x]exp)v**. The interaction between abstraction and application is defined by the simple rule:

$$\mathbf{([x]exp)x = exp} \qquad (1)$$

In the lambda calculus **[x]exp** is written **λx.exp**, and the process of applying it to a value **v** is called *beta-reduction*. Logically beta-reduction causes substitution of **v** for all free occurrences of **x** in **exp**, but it is more typically implemented by providing an *environment* to **exp** in which the value of **x** is bound to **v**. The environment ensures that $\mathbf{(\lambda x.exp)x = exp}$.

2.1. A Fixed Set of Combinators

Alternatively, using the *combinatory calculus*, one may abstract **x** from **exp** according to the following rules:

$$\mathbf{[x]x = I}$$
$$\mathbf{[x]y = K\ y}, \text{ if } \mathbf{x \neq y}$$
$$\mathbf{[x](e_1\ e_2) = S\ ([x]e_1)\ ([x]e_2)}$$

where **S**, **K**, and **I** are primitive functions called *combinators* that are defined (assuming function application associates to the left) by:

$$\mathbf{I\ x = x}$$
$$\mathbf{K\ y\ x = y}$$
$$\mathbf{S\ f\ g\ x = f\ x\ (g\ x)}$$

Using these rules it is easy to show that this method of abstraction obeys the rule given in equation (1). Note that through repeated abstractions of this sort, all bound variables may be eliminated from a program. Thus the "substitution" operation in the lambda calculus becomes meaningless, and there is no need for an environment.

2.2. Super-combinators

The combinators **S**, **K**, and **I** form a rather small, fixed set of primitives, which is quite attractive. Turner observed that through the introduction of several other fixed combinators the size of the resulting combinatory code could be reduced substantially, and the run-time efficiency could be improved as well [19]. However, as argued earlier, there are reasons for wanting even larger granularity. Technically, a combinator is simply a lambda expression that has no free variables and is a "constant applicative form;" that is, it contains only bound variables and constants that are combined by application. These are the crucial properties for a graph reducer, since they allow computed values to overwrite the nodes from which they were derived, without making copies of the bodies of functions. With this generalization Hughes [12] introduced the notion of a *super-combinator*, as explained below.

A *free expression* in a lambda expression λ**v.e** is defined as any subexpression of **e** in which **v** does not occur as a free variable. A *maximally free expression* is a free expression which is not part of any larger free expression. In a nutshell, starting with a program **P**, Hughes' algorithm for generating super-combinators is then:

1. Find the leftmost, innermost lambda expression $\mathbf{L} = \lambda\mathbf{v.exp}$.
2. Find the maximally free expressions, $\mathbf{e_1}$ through $\mathbf{e_n}$, of **L**.
3. Create a new combinator (say α) defined by:
   $$\alpha\ \mathbf{i_1} \ldots \mathbf{i_n}\ \mathbf{v} = \mathbf{exp}[\mathbf{i_1}/\mathbf{e_1}, \ldots, \mathbf{i_2}/\mathbf{e_2}]$$
   where formal parameter names $\mathbf{i_1}$ through $\mathbf{i_n}$ do not occur free in **exp**. (The expression **e**[**x**/**y**] denotes the result of substituting **x** for all occurrences of **y** in **e**.)
4. Substitute $(\alpha\ \mathbf{e_1} \ldots \mathbf{e_n})$ for **L** in **P**.
5. Repeat steps 1-4 until step 1 fails.

Together with a few optimizations, the resulting super-combinators have a very useful property: execution of the original program results in a *fully lazy evaluation*. No subexpression internal to a combinator body needs to be recomputed as a result of a partially applied combinator being shared in several places.

It should be pointed out that from a complexity standpoint super-combinators (and serial combinators) seem to have an inherent efficiency advantage over a fixed set of combinators, based on the following argument: Consider a lambda expression **exp** with a free variable **x** that occurs at lexical depth n. Using a standard fixed set of combinators, at least n abstractions are needed to abstract **x** from **exp**, and likewise at least n reductions take place when the resulting expression is applied to an actual argument for **x**! Yet with a super-combinator the overhead is often constant, independent of the depth.

Furthermore, conventional compiler technology can be applied to the bodies of super-combinators, yielding extremely efficient "straight-line" code, including optimal code for trees and single-level environments. This suggests an architecture (such as the multiprocessor that is of interest to us) that combines the elegance and generality of graph reduction with the efficiency and practicality of a conventional register machine. This particular feature is not exploited by Hughes, whose reduction strategy is to expand super-combinators into their graphical equivalents, after which normal graph reduction proceeds. Such a strategy has the advantage of avoiding the depth-n complexity mentioned above, but fails to take advantage of conventional compilation techniques.

2.3. Serial Combinators

Of the many possible parallel computing models, it is probably safe to say that a fixed set of combinators offers in some sense the *finest* granularity of computation, even finer than dataflow. However, if one studies the performance figures for existing multiprocessors, it becomes apparent that the ratio of interprocessor communication time to CPU instruction speed is generally quite high; typically anywhere from 10 to 100. Thus it seems that relatively large "grains" of parallelism need to be found for our overall strategy to be successful. We initially became motivated to do this after observing via simulation that with a fixed set of combinators it is possible for a purely sequential computation (i.e., one whose data-dependencies preclude any parallelism) to become decomposed for execution on several processors [5]. Clearly this is a wasted effort, since such a decomposition can only add communication costs to an already-sequential computation!

Although considerable progress has been made in parallel evaluation models, little work has been done on choosing the right granularity for parallel program decomposition. Our goal is to retain the environment-less nature of combinators and their usefulness in a parallel graph reducer, while maximizing their granularity and ensuring that parallelism is not lost. The strategy used by Keller and Lin [14] relies predominantly on the functions defined by the user, and is similar to the "lambda-lifted" functions used by Johnsson [13] -- neither technique gurantees fully lazy combinators, and no analysis is attempted within function bodies to detect parallism. Super-combinators are a step in the right direction, having considerably larger granularity than a fixed set of combinators, but they are again targeted for sequential machines, and thus are not designed to maximize opportunities for parallelism.

In this paper we introduce objects called *serial combinators* that we argue are in some sense "optimal" because:

1. They are combinators, facilitating their use in a graph reduction machine (especially parallel ones).
2. They result in a fully lazy evaluation, guaranteeing that no extraneous computations are performed.
3. They have no concurrent substructure, guaranteeing that no available parallelism will be lost.
4. There are no larger objects having these same properties, ensuring that no extraneous communication costs are incurred because of too fine a granularity.

Even if the third refinement is ignored, simulations demonstrate that the resulting combinators are more efficient than super-combinators, on both sequential and parallel machines. This is because serial combinators have a larger granularity, including the fact that they may be recursive, eliminating inefficiencies involving the **Y** combinator.

Despite the relatively simple characterization of serial combinators as given above, we make important pragmatic refinements that complicate their analysis. In particular, we take into consideration strictness properties of functions, common subexpressions, complexity of subexpressions, and the overhead for distributing a computation. These refinements are described in detail in the following sections.

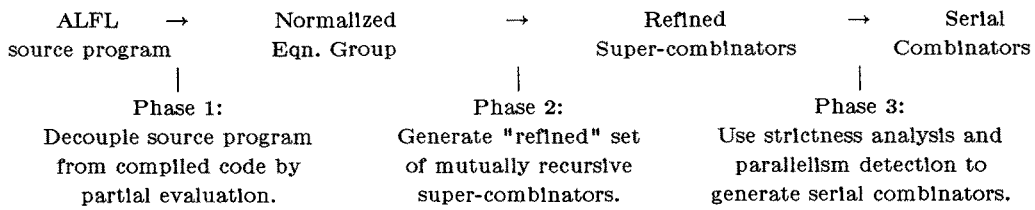


Figure 2-1: Overall Compilation Scheme

The overall translation scheme from source program to serial combinators consists of three phases, shown diagrammatically in Figure 2-1. ALFL [7] is a functional language that is similar to SASL and FEL -- we assume the reader to be familiar enough with this style of language that the examples will be self-explanatory. The ALFL source program is first translated into an "intermediate form" called a *normalized equation group*. A set of "refined

super-combinators"[2] is then generated, having the improved properties mentioned earlier. Finally, issues of parallel computation are taken into account as the super-combinators are further refined into serial combinators. These three translation phases are described in turn in the next three sections.[3]

3. Phase 1: Normalized Equation Groups

An important goal of our research is to devise a translation scheme that relies as little as possible on the format of the source program. Indeed, this feature seems to be more of an issue in parallel systems than sequential ones. For example, in sequential systems one may perform "procedural abstractions" for stylistic reasons, often with little impact on performance (i.e., no more impact than the overhead of a procedure call). However in parallel systems it is often the case that function boundaries play an essential role in the detection of parallelism, and thus such stylistic decisions can drastically affect performance. Another example concerns common subexpressions -- if the user expresses something as shared, should it remain that way, even if the cost to recompute it is less than the cost to access its value remotely? Similarly, if the user duplicates code, should it be detected as a common subexpression and transformed into a shared structure to avoid recomputation?

The best strategy that we know of for decoupling the source program from the compiler output is *partial evaluation*. Our compiler uses the partial evaluation strategy described in [8], where "constant-folding" is taken to its limit and the program is "eagerly" executed except for recursive function calls.[4] The result is a form of "normalization" that is quite useful, allowing the programmer to worry less about the form of the source program, and more about its correctness. On the other hand, the primary purpose of this paper is to describe the translation process from lambda expressions (or some equivalent) to serial combinators (i.e., Phases 2 and 3) and therefore the details of partial evaluation are not discussed here. Indeed, the systems designer may wish to perform only standard transformations on the source program, such as constant folding and common subexpression elimination, in order to provide the user with more direct "control" over the compilation process. Although we suggest the use of partial evaluation, we make no assumptions in Phase 2 as to how the intermediate form is obtained.

The intermediate form that we *do* assume departs somewhat from the standard representation of a program as a lambda expression. In our representation a program is a *graph*, represented lexically as a set of mutually recursive equations called a *normalized equation group*, having the form:

$$\begin{aligned} &\textbf{exp whererec } f_1 = e_1, \\ &\qquad\qquad\quad f_2 = e_2, \\ &\qquad\qquad\quad \ldots, \\ &\qquad\qquad\quad f_n = e_n \end{aligned}$$

where $n \geq 0$. The expression **exp** represents the *value* of the equation group, and the f_i are

---

[2] We refrain from calling them *super-duper-combinators*!

[3] In a previous paper [9] we described a translation scheme that began with Hughes' algorithm, and through a series of refinements to the super-combinators gradually arrived at serial combinators. The three-phase strategy given here is quite different and considerably clearer, although it generates essentially the same combinators.

[4] A compiler option allows even recursive calls to be "unwound" a small number of steps.

local identifiers. Each e_i must be either a lambda abstraction, in which case f_i is intended to be a combinator, or some other expression, in which case f_i is interpreted as a common subexpression. (Perhaps a better way to view this is that common subexpressions are simply nullary combinators.) Furthermore, the expressions e_i are not allowed to contain nested equation groups, which can be accomplished by "lambda lifting" [13] or some other suitable transformation. These constraints on e_i are why we refer to the equation group as *normalized* -- it is assumed that a suitable source-to-source translation has converted the source program into a form meeting the "requirements" of Phase 1. The partial evaluation used by our compiler, for example, accomplishes this form trivially, and furthermore guarantees that the f_i that are lambda abstractions are recursive -- all other functions become integrated wherever they are applied.

Although a normalized equation group can be converted into a single non-recursive lambda expression using the **Y** combinator, the graph implied by the equation group provides us with sharing information (including that implied by recursive references) that is difficult to attain from a pure lambda calculus expression. Furthermore, we want to be able to derive *mutually recursive combinators*; i.e., program-derived combinators that are defined in terms of one another. The fixed **Y** combinator simply gets in the way of achieving that goal.

4. Phase 2: Conversion to Refined Super-combinators

A major emphasis of Hughes' algorithm for generating super-combinators is to ensure the property of "fully lazy evaluation." Although one could argue that in a parallel system this is not as important a goal, it is still generally worthwhile, and something that we also wish to accomplish. However, Hughes' algorithm makes one important, but overly restrictive, assumption: namely, that *any* partially applied combinator could be *shared.* For example, in the definition **f x y = x*x + y**, one would not want **f** to be a combinator if, say, **f a b** and **f a c** appeared in the source program, because **a*a** would get computed twice, even if **f a** is detected as a common subexpression. It would be better to define **f** as **f x = f' (x*x)** where **f' x y = x + y**.

The reason the assumption about sharing is overly restrictive is that *a partially applied function is rarely shared.* If it is not shared, then one combinator could be used in place of the two required by Hughes' algorithm. In general, if no partial applications of an n-ary combinator are shared, then one combinator can be used in place of n. For example, although **f** as defined above might appear in many contexts in the source program, it will usually be the case that it will receive both of its arguments and that no two occurrences of the first one can be determined to be the same. Thus it is acceptable to treat **f** as a single combinator instead of creating two as suggested earlier.

Why is this important? That is, what is wrong with having several combinators where one would do? The answer is simply that under most reasonable assumptions, the execution of two combinators is likely to be less efficient than the execution of one that performs the same task. Consider:

- In a graph reduction implementation it means that a fully applied combinator can be represented more compactly, akin to a conventional activation record.

- It also means that fewer "templates" are needed for expressions passed as arguments. With lazy evaluation, even if an expression is not evaluated, it needs a representation, and this "hidden cost" of graph reduction can be non-trivial.

- On most machines it means that argument access will be more efficient.

- Finally, and perhaps most importantly, it means that on a multiprocessor locality will be enhanced since sequential code is less likely to be needlessly distributed across several processors.

One of our goals, then, is to create the *largest* combinators possible; larger, at least, than "standard" super-combinators.

4.1. Detecting Sharing

To detect whether or not a partially applied combinator is shared, it is clear that one must first detect common subexpressions.[5] Unfortunately, this is not completely sufficient -- sharing may also occur when a partial application is passed as an argument to another function. More precisely, consider an n-ary combinator α that is partially applied to m arguments, where $0 \leq m < n$. If this partial application appears as an argument to another combinator, then we assume that the partial application of α to k arguments, for all k, $m \leq k < n$, might be shared.

The above analysis is actually rather *conservative*, although it appears to do extremely well in practice. It is conservative because it is possible (indeed likely) for a partially applied combinator to be passed as an argument to another combinator yet not be shared. Performing a more liberal analysis requires a more sophisticated inferencing strategy, such as abstract interpretation, which is beyond the scope of this paper.

It should also be pointed out that it is possible for one to create different versions of a combinator to account for its different sharing properties. In the earlier example, one might use the combinators **f** and **f'** at points in the program where sharing happens, and use the original **f** in situations where it can be inferred that no sharing will take place. This has the disadvantage of increasing code size, but is likely to reduce program execution time, and avoids the degradation in performance of a function applied in one place due to its application in some other place -- such "non-local" effects are bothersome. The algorithm given in Section 4.3 creates as many combinators as it needs to cover the various sharing possibilities.

4.2. Preliminary Notation

An identifier **f** may occur in many places; its ith occurrence is denoted $\mathbf{f}^{\mathbf{i}}$. Each occurrence $\mathbf{f}^{\mathbf{i}}$ of a function **f** implies distinct sharing properties for **f**. To capture these properties for a function **f** that occurs in **n** places, we define the two functions *shares* and *all-shares* by:

$$shares(\mathbf{f}^{\mathbf{i}}) = \{\ \mathbf{k} \mid \mathbf{f}^{\mathbf{i}} \text{ applied to } \mathbf{k} \text{ arguments might be shared}\ \}$$

$$all\text{-}shares(\mathbf{f}) = \{\ shares(\mathbf{f}^{\mathbf{i}}) \mid \mathbf{i=1,\ldots,n}\ \}$$

Note that *all-shares*(**f**) is a set of sets.

When the context is clear, we write $\mathbf{e_1} \subset \mathbf{e_2}$ to indicate that $\mathbf{e_1}$ is a proper subexpression of $\mathbf{e_2}$. Similarly, $\mathbf{x} \in \mathbf{e}$ indicates that the identifier **x** occurs free in **e**. Given an arbitrary

---

[5] By our assumptions about normalized equation groups, this has already been done -- if a partially applied combinator is shared it will appear as the right-hand side of some equation. In a graphical sense one needs only to find partial applications whose in-degree, or reference count, is greater than one.

expression **exp** and set of variables **s**, the set of *maximally free expressions of* **exp** *with respect to* **s**, denoted **mfe(exp,s)**, is defined by:

$$\mathbf{mfe(exp,s)} = \{\ \mathbf{e} \mid \mathbf{e} \subset \mathbf{exp} \wedge (\forall \mathbf{x} \in \mathbf{s},\ \mathbf{x} \notin \mathbf{e}) \wedge (\forall \mathbf{f} \in \mathbf{mfe(exp,s)},\ \mathbf{e} \not\subset \mathbf{f})\ \}$$

The last term in the conjunction ensures that the *largest* free expressions are found. As an optimization to this definition, we avoid creating "trivial closures" (the partial application of primitive functions) by not abstracting free expressions that consist solely of a primitive operator with an incomplete set of arguments, such as "+ x."

4.3. Algorithm for Generating Refined Super-combinators

Given the above definitions and a normalized equation group of form:

$$\begin{aligned}&\mathbf{exp\ where rec}\ \mathbf{f_1} = \mathbf{e_1},\\ &\qquad\qquad\quad \mathbf{f_2} = \mathbf{e_2},\\ &\qquad\qquad\quad \ldots,\\ &\qquad\qquad\quad \mathbf{f_n} = \mathbf{e_n}\end{aligned}$$

the algorithm for generating refined super-combinators is:[6]

For each $\mathbf{f_j} = \lambda \mathbf{x_1} \ldots \mathbf{x_m}.\ \mathbf{e}$, and each $\mathbf{s} \in \mathit{all\text{-}shares}(\mathbf{f_j})$, do:

1. Create a new function $\mathbf{g_j} = \lambda \mathbf{x_1} \ldots \mathbf{x_m}.\ \mathbf{e}$, and substitute $\mathbf{g_j}$ for each occurence $\mathbf{f_j^i}$ for which $\mathit{shares}(\mathbf{f_j^i}) = \mathbf{s}$.
2. For each $\mathbf{k} \in \mathbf{s}$ *from largest to smallest*, do:
   a. Create a new combinator α of form:
      $$\alpha\ \mathbf{v_1} \ldots \mathbf{v_p}\ \mathbf{x_{k+1}} \ldots \mathbf{x_m} = \mathbf{e}[\mathbf{v_1}/\mathbf{y_1}, \ldots, \mathbf{v_p}/\mathbf{y_p}]$$
      where $\mathbf{mfe(e,\{x_{k+1},\ldots,x_m\})} = \{\mathbf{y_1},\ldots,\mathbf{y_p}\}$.
   b. Replace the definition of $\mathbf{g_j}$ by:
      $$\mathbf{g_j} = \lambda \mathbf{x_1} \ldots \mathbf{x_k}.\ \alpha\ \mathbf{y_1} \cdots \mathbf{y_p}$$

As a final step, "redundant" combinators of the form $\alpha = \beta$ are eliminated.

4.4. An Example

Consider this contrived example:

$$\begin{aligned}\mathbf{f} = \lambda \mathbf{x\ y\ z.\ if}\ &(= (\text{-}\ \mathbf{x\ y})\ (+\ \mathbf{y\ z}))\\ &(+\ (\text{**}\ \mathbf{x\ 2})\ (\text{-}\ \mathbf{y\ z}))\\ &(\text{-}\ \mathbf{z}\ (+\ \mathbf{x\ y}))\end{aligned}$$

Using Hughes' algorithm, abstracting all mfe's, even ones that we know to be partial applications, we get:

$$\alpha\ \mathbf{a\ b\ c\ d\ e\ z} = \mathbf{if}\ (\mathbf{a}\ (\mathbf{b\ z}))\ (\mathbf{c}\ (\mathbf{d\ z}))\ (\text{-}\ \mathbf{z\ e})$$
so $\mathbf{f} = \lambda \mathbf{x\ y}.\ \alpha\ (= (\text{-}\ \mathbf{x\ y}))\ (+\ \mathbf{y})\ (+\ (\text{**}\ \mathbf{x\ 2}))\ (\text{-}\ \mathbf{y})\ (+\ \mathbf{x\ y})$

$$\beta\ \mathbf{a\ b\ c\ y} = \alpha\ (= \mathbf{a\ y})\ (+\ \mathbf{y})\ \mathbf{b}\ (\text{-}\ \mathbf{y})\ \mathbf{c}$$
so $\mathbf{f} = \lambda \mathbf{x}.\ \beta\ (\text{-}\ \mathbf{x})\ (+\ (\text{**}\ \mathbf{x\ 2}))\ (+\ \mathbf{x})$

$$\gamma\ \mathbf{x} = \beta\ (\text{-}\ \mathbf{x})\ (+\ (\text{**}\ \mathbf{x\ 2}))\ (+\ \mathbf{x})$$
and thus $\mathbf{f} = \gamma$

---

[6] The reader should compare this to Hughes' algorithm given in Section 2.2.

Again using Hughes' algorithm, but not abstracting expressions that we know to be partial applications, we arrive at:

α **a b c y z = if (= a (+ y z)) (+ b (- y z)) (- z c)**
so **f** = λ**x y.** α **(- x y) (** x 2) (+ x y)**

β **a x y** = α **(- x y) a (+ x y)**
so **f** = λ **x.** β **(** x 2) x**

γ **x** = β **(** x 2) x**
and thus **f** = γ

Consider now using our method, which still results in full laziness, but passes variables instead of expressions where there is no loss in doing so. There are four sharing possibilities:

(1) Suppose *no* partial application of **f** is shared. Then there is no reason to incur the cost of passing expressions. In this case the algorithm essentially degenerates into "lambda lifting" -- combinators are generated by explicitly passing all free variables that occurred in the body of **f**. The above example becomes:

α **x y z = if (= (- x y) (+ y z)) (+ (** x 2) (- y z)) (- z (+ x y))**
and thus **f** = α

(2) Suppose only **(f n)** is shared, for some argument **n**. We then only need to abstract the maximally free expressions with respect to **y** and **z** from the body of **f**. For the above example this results in:

α **a x y z = if (= (- x y) (+ y z)) (+ a (- y z)) (- z (+ x y))**
so **f** = λ**x.** α **(** x 2) x**

β **x** = α **(** x 2) x**
and thus **f** = β

(3) Suppose only **(f n p)** is shared, for some arguments **n** and **p**. We then only need to abstract the maximally free expressions with respect to **z** from the body of **f**, yielding:

α **a b c y z = if (= a (+ y z)) (+ b (- y z)) (- z c)**
so **f** = λ**x y.** α **(- x y) (** x 2) (+ x y)**

β **x y** = α **(- x y) (** x 2) (+ x y)**
and thus **f** = β

since we know that **(f n)** is never shared.

(4) It may be that both **(f n)** and **(f p q)** are shared for some arguments **n**, **p**, and **q**. If possible we could generate both sets of combinators described in (2) and (3) above, and replace an occurrence of **f** by one of the appropriate combinators depending on the sharing properties of that particular occurrence. In the worst case the algorithm would degenerate into Hughes' super-combinator algorithm, which maintains full laziness regardless of how the partial applications are shared.

5. Phase 3: Conversion to Serial Combinators

So far the translation process has had little to do with parallel computation, yet even on sequential machines the resulting "refined super-combinators" show a significant improvement in performance over regular super-combinators (examples demonstrating this are given in a later section). The task now is to consider the requirements of a parallel system, or more specifically the requirements of a *non-shared memory multiprocessor*, which is the sort of architecture that is of most interest to us. The processors in such a machine are typically rather conventional, and are interconnected by some homogeneous message-passing communications network, in which communications time is non-trivial. The ease in which one can build such a machine has resulted in the availability of several commercial multiprocessors.

It may seem as if basing our work on a multiprocessor whose nodes are conventional "von Neumann" machines instead of special-purpose graph reduction engines is wrong. However, we feel that conventional architectures will continue to be extremely efficient at executing sequential code. Since even parallel programs have a great deal of sequential dependencies, we wish to be able to take advantage of standard architectures and their associated compiler technology whenever possible. To do this we try to limit the use of graph reduction to just those situations where it is needed to support parallel computation. On the other hand, although we identify the sequential portions of a program, the actual compilation of that code is for the most part left unspecified. Our personal interest is in compiling into sequential code for a conventional machine, but we do not preclude the ultimate use of specialized sequential graph reduction engines.

5.1. General Discussion

Even though the compilation of sequential code is left unspecified, there are certain temporal dependencies that we wish to make explicit, and that require a notation for sequencing. In particular, a distinguishing feature of serial combinators is that they alternately become "active" and "suspended" at various times, depending on the availability of data, and furthermore that they occasionally fork the parallel evaluation of subexpressions (via a call to another serial combinator). In general the details of the former feature can be left implicit; i.e., one can assume that if a value is not yet available that the process suspends until it is. Mechanisms for implementing such synchronization are well-understood. However, the forking of subexpression evaluation needs to be made more explicit, since we want such evaluation to begin as soon as possible. To do this, the following construct is introduced into our "equational lambda calculus:"

```
(spawn ( (v1 exp1)
         (v2 exp2)
         ...
         (vn expn) )
   exp)
```

where **spawn** is a keyword that signifies the parallel evaluation of the expressions **exp1** through **expn**, which are referenced in the result expression **exp** as identifiers **v1** through **vn**. Thus by using the **spawn** form we can essentially express a "sequential thread" of execution, namely the body **exp**. Each concurrent subexpression within **exp** has been replaced by a simple identifier that refers to one of the spawned subexpressions. Note that this construct still leaves implicit the mechanisms for accessing the data. In this way we achieve the goal of not *overly* specifying the sequential code.

That there is inherent concurrency in an expression, however, does not mean that

distributing its evaluation among several processors is the most advantageous thing to do. There may be cases where the cost of distributing concurrent subexpressions far outweighs the cost of computing the whole expression locally. We would like to avoid these cases whenever possible.

Consider an expression **exp** of the form **op e1 e2**. The total time T(**exp**) to compute **exp** on a single processor is roughly T(**e1**) + T(**e2**) + C(**op**), where C(**op**) is the cost of executing the primitive operation **op**. If an extra processor is available, the total time is roughly MAX(T(**e1**), T(**e2**) + distribution-costs) + C(**op**), assuming that **e2** was evaluated on the extra processor. Clearly if T(**e2**) + distribution-costs < T(**e1**) + T(**e2**), then distributing the evaluation of **e2** is a win; otherwise we are slowing down the overall execution, even though there is "apparent" parallelism!

A little thought should convince the reader that one should retain the *most* complex subexpression for inclusion in the "sequential thread" of execution, making the other subexpression available for possible parallel execution. In the above example that would mean T(**e1**) > T(**e2**), guaranteeing a minimal value for MAX(T(**e1**), T(**e2**) + distribution-costs). If the analysis then shows that the other subexpression is worth distributing, it is compiled as a separate combinator, otherwise not. The details of this will be given shortly.

Since accurately determining the time needed to evaluate an expression is impossible in the general case, we use a heuristic in which expressions are weighted by the complexity of the primitive operations. These heuristics may be "fine-tuned" to the performance of a particular machine (for example, a significantly different analysis should take place on a machine with floating-point hardware compared to one without), but the details are not important to this paper.

Phase 3 uses one other analysis of expressions that we take as given: *strictness analysis* to determine which subexpressions must be evaluated in order to evaluate an expression containing them. Techniques for doing this are well-known, and can be found in [16, 10].

5.2. Overview

The result of Phase 2 of the translation process is a set of refined super-combinators. Phase 3 essentially performs a tree-walk of the body of each of these combinators, looking for parallel segments to be "serialized." Such serialization results in the generation of new combinators as well as the transformation of the old ones to include, for example, the use of **spawn.**

Intuitively, the strategy for generating serial combinators from a refined super-combinator is:

1. Determine its concurrent substructure. In its basic form, this simply means finding primitive operators that are strict in more than one of their arguments, such as the basic arithmetic and relational operators (+, −, =, >, etc.). With a suitable strictness analysis, the same thing can be done for the arguments to derived combinators.
2. For each subtree containing several concurrent subexpressions, retain one of the subexpressions in the definition of the current combinator (since it will represent part of the combinators "sequential thread"), and compile each of the other subexpressions into a separate serial combinator (to allow them to be computed in parallel).

Interestingly, there are two fundamentally different ways to tackle the second step. Although it is clear that one should want parallel subexpressions to be spawned as early as possible, it is not clear how to do this when such expressions are nested within one another. For example, consider the expression **(e1...(e2...(e3...)))** in which **e1** is strict in **(e2...(e3...))** and **e2** is strict in **(e3...)**. Should both **e2** and **e3** be spawned at the level of **e1**, or should just **e2** be spawned there, which in turn will spawn the evaluation of **e3**? The answer depends somewhat on the details of the implementation, which we wish to avoid. In the algorithm given below, the first option is chosen; that is, such expressions are "flattened" to yield the most parallelism.

5.3. An Algorithm for Generating Serial Combinators

Let *complexity* be the auxilliary function that given an expression returns an estimate of its execution-time complexity. Let c be a complexity threshold below which an expression is not complex enough to warrant decomposition into finer pieces, regardless of any internal parallelism.

The bulk of the work in Phase 3 is done by the function *serialize*, which takes an expression **exp** and returns a tuple **<new-exp,spawn-list,new-combs>**, where **new-exp** is a modified version of **exp**, **spawn-list** is an "association list" mapping identifiers to expressions to be evaluated in parallel, and **new-combs** is a list of new serial combinator definitions. *Serialize* is defined by:

serialize(**exp**) =

1. If *complexity*(**exp**) $< c$, then return **<exp,(),()>**.
2. Else if **exp** is of form **op** $\mathbf{e_1}$ $\mathbf{e_2}$, where **op** is a primitive strict operator, then determine the complexities of $\mathbf{e_1}$ and $\mathbf{e_2}$ -- assume that $\mathbf{e_1}$ is the most complex. If *complexity*($\mathbf{e_2}$) $> c$, then let:
   serialize($\mathbf{e_1}$) = <$\mathbf{e_1}$', spawn-list$_1$, new-combs$_1$>
   serialize($\mathbf{e_2}$) = <$\mathbf{e_2}$', spawn-list$_2$, new-combs$_2$>
   Suppose spawn-list$_2$ is of form: $((\mathbf{p_1}\ \mathbf{exp_1})\ \ldots\ (\mathbf{p_j}\ \mathbf{exp_j}))$. Then define the combinator α by:
   $\alpha\ \mathbf{p_1 \cdots p_j\ a_1 \cdots a_k} = \mathbf{e_2}'$
   where $\mathbf{a_1}$ through $\mathbf{a_k}$ are the *remaining* free variables in $\mathbf{e_2}'$. Let $\mathbf{u_1}$ be a new identifier. Finally, return:
   <(**op** $\mathbf{u_1}$ $\mathbf{e_1}'$),
   append(spawn-list$_1$, spawn-list$_2$,
   [($\mathbf{u_1}$ (α $\mathbf{p_1 \cdots p_j}$))]),
   append(new-combs$_1$, new-combs$_2$,
   [$\alpha\ \mathbf{p_1 \cdots p_j\ a_1 \cdots a_k} = \mathbf{e_2}'$]) >
3. Else if **exp** is a conditional of form **(if pred con alt)** then let:
   serialize(**pred**) = <**pred'**, spawn-list$_1$, new-combs$_1$>
   serialize(**alt**) = <**alt'**, spawn-list$_2$, new-combs$_2$>
   serialize(**con**) = <**con'**, spawn-list$_3$, new-combs$_3$>
   and return:

$<$(**if pred' (spawn** spawn-list$_2$ **alt')**
(spawn spawn-list$_3$ **con')**),
spawn-list$_1$,
append(new-combs$_1$, new-combs$_2$, new-combs$_3$)$>$

4. Else if **exp** is of form (**seqop** $\mathbf{e_1}$ $\mathbf{e_2}$), where **seqop** is a "sequential" primitive such as **and** or **or**, then let:

serialize($\mathbf{e_1}$) = $<\mathbf{e_1}'$, spawn-list$_1$, new-combs$_1>$
serialize($\mathbf{e_2}$) = $<\mathbf{e_2}'$, spawn-list$_1$, new-combs$_1>$

and return:

$<$(**seqop** $\mathbf{e_1}'$ (**spawn** spawn-list$_2$ $\mathbf{e_2}'$))
spawn-list$_1$,
append(new-combs$_1$, new-combs$_2$)$>$

5. Otherwise, **exp** must be of the form ($\mathbf{e_0}$ $\mathbf{e_1}$ ... $\mathbf{e_n}$). Let:

serialize($\mathbf{e_i}$) = $<\mathbf{e_i}'$, spawn-list$_i$, new-combs$_i>$.

for i=1,...,n. By a suitable strictness analysis compute the set S = { j | $\mathbf{e_0}$ is strict in $\mathbf{e_j}$ }. For each j ∈ S, create a new identifier $\mathbf{u_j}$. Further, suppose spawn-list$_j$ is of form: (($\mathbf{p_{j1}}$ $\mathbf{exp_{j1}}$) ... ($\mathbf{p_{jk}}$ $\mathbf{exp_{jk}}$)). Then define the combinator α_j by:

α_j $\mathbf{p_{j1}}$...$\mathbf{p_{jk}}$ $\mathbf{a_{j1}}$...$\mathbf{a_{jm}}$ = $\mathbf{e_j}'$

where $\mathbf{a_{j1}}$ through $\mathbf{a_{jm}}$ are the *remaining* free variables in $\mathbf{e_j}'$. Furthermore, for each h ∉ S, define:

β_h $\mathbf{a_{h1}}$...$\mathbf{a_{hm}}$ = $\mathbf{e_h}'$

where $\mathbf{a_{h1}}$ through $\mathbf{a_{hm}}$ are the free variables in $\mathbf{e_h}$. Finally, return:

$<$($\mathbf{e_0}'$ E1 ... En),
append(spawn-list$_0$, ..., spawn-list$_n$,
[($\mathbf{u_j}$ (α_j $\mathbf{p_{j1}}$...$\mathbf{p_{jk}}$)), for each j ∈ S]),
append(new-combs$_1$, ..., new-combs$_n$,
[(α_j $\mathbf{p_{j1}}$...$\mathbf{p_{jk}}$ $\mathbf{a_{j1}}$...$\mathbf{a_{jm}}$ = $\mathbf{e_j}'$)
...
(β_h $\mathbf{a_{h1}}$...$\mathbf{a_{hm}}$ = $\mathbf{e_h}'$)])$>$

where each **Ei** is either (1) $\mathbf{u_i}$, for strict arguments,
(2) β_i $\mathbf{a_{h1}}$...$\mathbf{a_{hm}}$ for non-strict arguments,
and (3) $\mathbf{e_i}$ for arguments whose complexity is below *c*, or

5.4. Some Refinements

Strictness analysis has played an essential role in deriving serial combinators. The **spawn** form essentially introduces *local* variables and calls for their immediate evaluation. However, we can also be more explicit about when each *bound* variable needs to be accessed during the execution of a serial combinator. This information is essentially already available, and can be made more explicit in any number of ways. The details are left to the reader.

Another refinement concerns the treatment of common subexpressions. Using the same complexity measures described earlier, it may be determined that a certain common subexpression is so trivial that recomputing its value may be cheaper than accessing its value remotely. In such cases the refined super-combinators can be modified so that the common subexpression is copied wherever it is referenced.

There are any number of refinements dealing with the evaluation model itself, and are perhaps best considered in a fourth phase for "code generation." For example, a very important optimization is that of generalized *tail recursion*, in which a combinator "template" (or activation record) is reused during any tail-recursive function call. Indeed, there are several other conventional compiler techniques that can be exploited in compiling a serial combinator body into code for a conventional machine.

5.5. An Example

Consider the following ALFL program to generate all solutions to the "queens" problem on a six-by-six board:

```
{ n == 6;
  Result queens n;
  queens n ==
  { Result allboards [] 0;
    allboards board col ==
    { Result col=n → [board], findboards 0;
      findboards row ==
        { Result row=n → [], newbds ^^ findboards (row+1);
          newbds == safe board (col-1) →
                        allboards (row^board) (col+1), [];
          safe   []   col1 == true;
          safe (r^bd) col1 == r ≠ row & col1 ≠ col &
                              abs(r-row) ≠ abs(col1-col) &
                              safe bd (col1-1)
} } } }
```

The symbols "^" and "^^" are infix operators for cons and append, respectively, and **hd** and **tl** are like car and cdr, respectively, in Lisp. A list may also be constructed using brackets, as in **[a,b,c]** (which is equivalent to **a^b^c^[]**). We will assume that the append operator in this example eagerly evaluates both of its arguments. In practice this can be accomplished either by providing such a strict operator as a primitive, or by annotating the source program. For example, one could write:

... newbds ^^ #(findboards (row+1)) ...

where the #-sign indicates that the second argument is to be evaluatd in parallel. Such annotations are beyond the scope of this paper, but are discussed thoroughly in [11].[7]

After partial evaluation we arrive at the following normalized equation group:

```
allboards [] 0   whererec

allboards board col = col=6 → board^[],
                      findboards board col 0
```

---

[7] As an aside, we should point out that in our experience "lazy lists" are one of the largest impediments to detecting parallelism in functional programs! There currently do not exist strong enough inferencing techniques to determine in non-trivial cases that an entire list will be evaluated. Until that time arrives, there appears to be a need for at least one of the following in any functional language being used for parallel computing: (1) a strict version of cons, append, and other list functions (such as provided in conventional Lisp), (2) a function that can "strictify" another function (such as the function **strict** described in [18]), or (3) an annotation that can "parallelize" subexpressions (such as the "eager expressions" described in [11] and [2]). Our preference is option (3), since the annotations can be designed so that they do not change the functional semantics of the program -- i.e., as a "black box" the lists still behave lazily, but one may eagerly evaluate the elements in parallel.

```
findboards board col row = row=6 → [],
                           (safe row col board (col-1) →
                                allboards (row^board) (col+1), [])
                           ^^ findboards board col (row+1)

safe row col bd col1 = (bd=[]) → true,
                       ((hd bd)<>row) & (col1<>col) &
                       (abs ((hd bd)-row)) <> (abs (col1-col)) &
                       (safe row col (tl bd) (col1-1))
```

In "conventional" super-combinators this becomes:

```
Result: (Y c12) nil 0
c12 allboards = c11 (c10 (c8 allboards))
c11 v28 board = v28 (^ board nil) board
c10 v26 v25 v27 col = c2 (= col 6) v25
                      (Y (v26 (c4 (- col 1)) v27 (+ col 1) (c6 (c5 col))))
c8 v21 v18 v20 v22 v23 v19 row = v18 (c3 (= row 6) (v19 (+ row 1)))
                                 v20 (v21 (^ row v20) v22) (Y (v23 row))
c6 v14 v15 v16 board = v14 (= board nil) (<> (hd board) v15)
                       (abs (- (hd board) v15)) (v16 (tl board))
c5 v11 v9 v10 v12 v13 col1 = if v9 true (& v10 (& (<> col1 v11)
                                  (& (<> v12 (abs (- col1 v11)))
                                   (v13 (- col1 1)))))
c4 v7 v5 v6 v8 safe = v5 (if (safe v6 v7) v8 nil)
c3 v3 v4 newboards = if v3 nil (^^ newboards v4)
c2 v1 v2 findboards = if v1 v2 (findboards 0)
```

In serial combinators we arrive at:

```
allboards board col =
   (if (= v1 6) (^ board nil) (findboards board col 0)))

findboards board col row =
   (if (= row 6) nil (spawn ( (v2 (α board col row))
                              (v3 (findboards board col (+ row 1))) )
                        (^^ v1 v2) ) )

α board col row =
   (if (safe board (- col 1) row col) (allboards (^ row board) (+ col 1)) nil)

safe bd col1 row col =
   (if (= bd nil) true (AND (<> (hd bd) row)
                            (<> col1 col)
                            (spawn ( (v2 (abs (- (hd bd) row))) )
                              (<> v2 (- col1  col)) )
                            (safe v3 (- col1 1) row col) ) )
```

6. Simulation Experiments

6.1. The DAPS Computing Model

Our physical model of computation is a very familiar one: a large number of processing elements (PE's) having only local store are interconnected in a homogeneous communications network and communicate by messages. The model is not new, and is sometimes referred to as a *network computer* or *ensemble architecture.* It is especially attractive since it appears to

be well-suited for VLSI design, and there is a growing number of real machines that fit its description. We refer to our model as **DAPS**, or *Distributed Applicative Processing Systems* [6], because of special features relating to the evaluation mechanisms and work distribution strategies [4].

DAPS is, of course, based on graph reduction, and for that reason there is a single global address space representing a free-space of available cells with which the program graph is constructed. Each PE is responsible for one contiguous portion of this address space, and thus parallelism comes to bear when concurrent redexes reside in the address spaces of different PE's. A task queue is maintained on each PE that essentially contains pointers to vertices in the graph representing available redexes. Execution occurs by initially mapping the program graph onto the global address space, and as the evaluation unfolds, expanding portions of the graph are allocated on neighboring PE's for increased parallelism. The choice of neighboring PE is controlled by a dynamic load-balancing technique that we refer to as "diffusion scheduling," which takes into account such factors as processor load, memory utilization, and direction of global references. In effect tasks are "pushed away" from busy processors, and "drawn toward" those to which they have global references (thus maintaining locality of reference). In this way work "diffuses" through the network in the direction of least resistance.

Previous work by the authors and others [5, 14] have demonstrated via simulation the viability of this style of architecture. In this section we present new simulation results using serial combinators, and compare them to similar results using super-combinators. Our simulator allows one to easily change parameters such as the number of PE's, network topology, and diffusion heuristics. The diffusion heuristic used in the following examples is as described in [5], whereas the other parameters are adjusted depending on the experiment. The interprocessor communication time between neighboring PEs was fixed at 10 "cycles," where one cycle is the reduction time of a typical primitive operator.

6.2. Results

Figure 6-1a shows the results of running the "six queens" example given in Section 4.3 on varying numbers of processors interconnected as both a hypercube and torus. The degree of parallelism on p processors is measured as a "speedup factor," which is the ratio of the execution time on one processor to the execution time on p. It is interesting to note that the results are almost identical for the two different topologies. In both cases ample degrees of parallelism are attained, and the speedup seems to increase in direct proportion to the number of processors.

In Figure 6-1b we plot the total number of "time steps" for the program, and compare them with a "parallelized" super-combinator version of the same program. Note that even on *one* processor, serial combinators out-perform super-combinators by a factor of about four.

As another example, consider this program to compute factorial in a divide-and-conquer manner:

```
{ fac n == pfac 1 n;
  pfac l h == h=l → l,
              h=l+1 → l*h,
              { c == (l+h)/2;
                result (pfac l c) * (pfac (c+1) h)};
  result fac n }
```

The resulting super-combinator expression is:

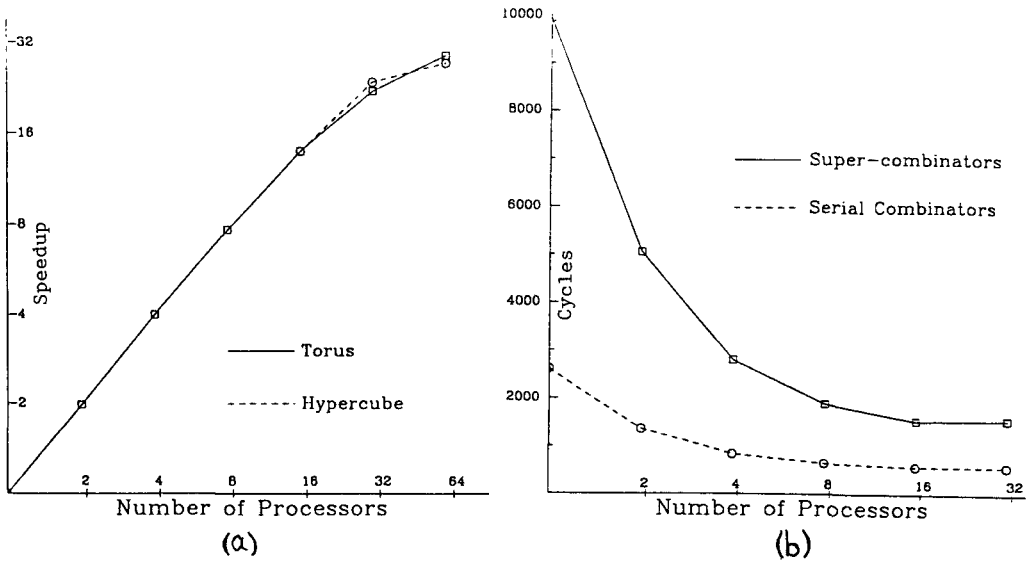


Figure 6-1: Six Queens

result: **Y** δ **1 n**
δ **pfac** = γ (α **pfac**) **pfac**
γ **v6 v7 l** = β **l (v6 (v7 l))**
β **v4 v5 h** = **(if (= v4 h) v4 (if (= v4 (+ h 1)) (* v4 h)**
(v5 h (/ (+ v4 h) 2))))
α **v2 v1 v3 c** = **(* (v1 c) (v2 (+ c 1) v3))**

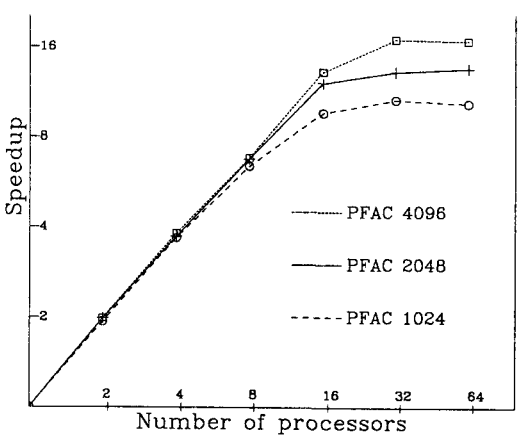
and the serial combinator expression is:

pfac l h = **if (= l h) l (if (= (+ l 1) h) (* l h)**
(spawn ((v1 (α l h))
(v2 (pfac l v1)))
(* v2 (pfac (+ v1 1) h))))
α **l h** = **(/ (+ l h) 2)**

The program **pfac n** was run on a simulated torus network for various values of **n**. The results are shown in Figure 6-2, and show that parallelism increases with **n**.

Figure 6-2:

Divide and Conquer Factorial



References

1. Barendregt, H.P.. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1984.

2. 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).

3. Curry, H.K., and Feys, R.. *Combinatoroy Logic.* Noth-Holland Pub. Co., Amsterdam, 1958.

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

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

6. Hudak, P. Distributed Applicative Processing Systems: Project Goals, Motivation, and Status Report. Technical Report YALEU/DCS/TR-317, Yale University, May, 1984.

7. Hudak, P. ALFL Reference Manual and Programmers Guide. Technical Report YALEU/DCS/TR-322, Second Edition, Yale University, Oct., 1984.

8. Hudak, P. and Kranz, D. A combinator-based compiler for a functional language. 11th ACM Sym. on Prin. of Prog. Lang., ACM, Jan., 1984, pp. 121-132.

9. Hudak, P. and Goldberg, B. Efficient distributed evaluation of functional programs using serial combinators. to appear in Proceedings of 1985 Int'l Conf. on Parallel Proc. and IEEE Trans. on Computers (October 1985), Aug., 1985.

10. Hudak, P. and Young, J. A set-theoretic characterization of function strictness in the Lambda Calculus. Technical Report YALEU/DCS/TR-391, Yale University, Jan., 1985.

11. Hudak, P. and Smith, L. Para-functional programming: a paradigm for programming multiprocessor systems. Technical Report YALEU/DCS/TR-390, Yale University, Jan., 1985.

12. Hughes, R.J.M. Super-combinators: A new implementation method for applicative languages. Sym. on Lisp and Functional Prog., ACM, Aug., 1982, pp. 1-10.

13. Johnsson, T. The G-machine: An Abstract Machine for Graph Reduction. PMG, Dept. of Computer Science, Chalmers Univ. of Tech., Feb., 1985.

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

15. Kieburtz, R.B. The G-machine: A Fast, Graph-Reduction Evaluator. CS/E-85-002, Dept. of Computer Science, Oregon Graduate Center, Jan., 1985.

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

17. Peyton Jones, S.L. with Clack, C. and Harris, N. GRIP -- A Parallel Graph Reduction Machine. Internal Note 1665, Dept. of Computer Science, Univ. College London, Feb., 1985.

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

19. Turner, D.A. "A new implementation technique for applicative languages". *Software -- Practice and Experience 9* (1979), 31-49.