Stony Brook Parallel Programming Project

R.C. Sekar, Shaunak Pawagi and I.V. Ramakrishnan
Department of Computer Science
SUNY at Stony Brook, NY 11794

1 Introduction

Numerous parallel computing systems are currently available and many more are still under development. There are two broad approaches to programming them. In one approach (as exemplified by CCS) the programmer explicitly specifies the concurrent processes and the coordination among them. In the alternative approach, the programmer specifies the computation at a high level and the task of parallelizing is delegated to a smart compiler. This approach removes a lot of burden from the programmer and hence makes parallel programming considerably simpler. The high level specification can be given in an imperative language such as FORTRAN or in a declarative language such as ML. In contrast to declarative languages, an imperative language admits side effects that render the task of parallelization (as well as several other compile time optimizations) very complex. Hence a lot of research has focussed on parallel programming using declarative languages [Flag88, Gold88]. In this paper we study the parallelization of a declarative language based on equations.

Equational programming is a style of declarative programming in which systems of oriented equations of the form $l \to r$ where l and r are terms (with variables in them universally quantified implicitly) constitute the set of logical assertions describing a program. Languages based on equations include O'Donnell's equational logic [OD85] and OBJ2 [OBJ2]. Functional programming languages such as ML, Miranda, HOPE etc. are also equational, when function definitions are viewed as rewrite rules.

An equational program is evaluated by repeatedly replacing *redices* (a redex is an instance of a left-hand side) in the input term by the corresponding right-hand sides. This process is known as *normalization* and the *normal form* of a term contains no redices. Parallelism in an equational program arises due to the presence of several redices in different parts of the term being normalized.

A *safe* or *conservative* approach to parallel evaluation of equational programs is to reduce in parallel only the *needed redices*. These redices alone need to be rewritten in order to reach the normal form. Alternatively, the *eager* approach reduces needed as well as some unneeded redices. While doing so could potentially increase parallelism, this does not seem to happen in practice. On the other hand, it has the drawback of tying up processors and memory in wasteful computations and to hold potentially large structures (terms) that may be created by these wasteful computations. Furthermore, the eager approach has to preempt the resources tied up by wasteful computations when they are in short supply, and hence is much more complex than the safe approach. For these reasons, our implementation is based on the safe approach. Such an implementation requires us to develop algorithms to compute needed redices. This has been studied in [HL79, OD85, ?]. However, all these algorithms are essentially sequential, since they pick only one needed redex at a time. The development of algorithms to detect more parallelism and utilize it effectively has remained an open problem. In this paper we propose a solution to this problem. The following is a summary of our results.

1.1 Overview and Summary of Our Results

In this paper we present a new class of equational programs (in section 2) known as *path sequential systems* that are natural for parallel evaluation. In particular, searching for needed redices is done with minimal and "regular" inter-processor communication. Even if the input program is not path sequential, it can be so transformed easily [Sek89a]. We then describe the opportunities for parallelism in equational programs through examples (in section 3). The known sequential algorithms for normalization, when extended for parallel evaluation as suggested by Thatte [?], O'Donnell [OD85] et al, do not lead to any significant parallelism in practice since they detect very few needed redices at any time. Hence we develop a method for analyzing the program (in section 4) to discover many more needed redices. This information is then used at run time to extract almost all the parallelism in the program.

In order to effectively utilize the parallelism extracted by our analysis procedure, the parallel tasks (i.e., normalization of portions of input term) should be distributed among the processors in such a way as to maximize concurrency while minimizing inter-processor communication. Since the tasks are created dynamically, the distribution (also known as load balancing) algorithm must also be dynamic. We describe analysis techniques (in section 5) that yield information on *whether* a task is to be executed locally or on a remote processor, and in the latter case *how* to choose the remote processor. We also develop techniques to detect a class of functions called *pipeline functions* to effectively utilize the parallelism they offer. A description of our architectural is given in section 6. Our current work is briefly outlined in setion 7.

2 Path Sequential Systems and Parallel Evaluation

We first introduce the notations and definitions that will be needed for the rest of this paper. Following this, we describe a parallel normalizing algorithm (suggested by Thatte and O'Donnell) and illustrate the problems that arise when it is used for parallel evaluation. We then define a class of systems known as *path sequential systems* where such problems are absent. (A detailed exposition to path sequential systems and complete and optimal algorithms for normalization in such systems appears in [Sek89c].)

A substitution β maps variables to terms. An instance $t\beta$ of a term is obtained by replacing x by $\beta(x)$ for each variable x in t. If s is an instance of t then we say $s \geq t$, and t is called a prefix of s. Thus a prefix of s corresponds to an "initial portion" of s. A *partial redex* is a prefix of an lhs. Now $s \rightarrow t$ means that t is obtained from s by replacing a redex $l\beta$ in s by $r\beta$ where $l \rightarrow r$ is rule. The reflexive and transitive closure of $\rightarrow$ is denoted by $\rightarrow^*$.

We use $\mathcal{L}$ to denote the set of lhs. The systems of equations permitted are restricted to be *regular* [OD85, HL79], i.e., the lhs are *linear* (no variable occurs more than once in an lhs) and *nonoverlapping* (two distinct lhs do not unify). They also follow a *constructor discipline*, i.e., the symbols in lhs can be partitioned into two disjoint sets, the functors which occur only as the outermost symbol and the constructors which occur only inside an lhs [HL79, **?**, OD85]. Such systems are referred to as *constructor systems*. In order that needed redices be computable, the input programs are restricted to be *strongly sequential*. This is the class admitted by Huet and Levy and Thatte [**?**, HL79]. Almost all equational programs follow all the above restrictions and so these restrictions do not appear to limit the expressiveness of the equational language.

The normalization of a term s in a constructor system proceeds as follows. Suppose that a prefix u of s has been inspected. The match set $\mathcal{L}_u$ of the prefix u is defined as $\{l \in \mathcal{L} \mid l \geq u\}$. Observe that only one of the lhs in $\mathcal{L}_u$ can ever match at the root of s. Suppose that there is a variable x in u that corresponds to a nonvariable subterm of every lhs in $\mathcal{L}_u$ (i.e., if $u = l\beta$, then $\beta(x)$ is a nonvariable subterm of l for every $l \in \mathcal{L}_u$). Such an x is known as an *index* and indices exist in every partial redex in strongly sequential systems [HL79]. Note that a match for any $l \in \mathcal{L}_u$ can arise in s only if the subterm t of s corresponding to x is reduced to a term with a constructor root (since root of t must match a symbol inside an lhs, which is a constructor). A parallel normalization algorithm will attempt to reduce all subterms of s that correspond to indices of u to their *head normal form*, i.e., a term with a constructor root[1]. Each such head normalization will proceed

[1]Strictly speaking head normal form implies that no redex can ever be uncovered at the root of the

analogously, in turn creating many more head normal form computations. This procedure continues until a redex is identified and rewritten at the root of s. If the root of s is now a constructor, then each of the subterms of s rooted at the children of its root are normalized using the same procedure. Otherwise, another partial redex u' is found at its root and the same procedure repeated.

We illustrate the problems with such a parallel algorithm with the following scenario. Suppose we wish to normalize a term $s = f(a(h(\cdots)), g(\cdots))$, where f, g and h are functors and the other symbols are constructors. Let the lhs with f as the outermost symbol be $f(a(c), b)$ and $f(a(x), c)$. Also assume that s is distributed, with f, $a(h(\cdots))$ and $g(\cdots)$ on processors p_1, p_2 and p_3 respectively. If p_1 initiates a match at the root of s, and attempts to complete the match locally, then it has to copy several nodes from p_2 and p_3. Apart from generating considerable amount of communication, such copying has the tendency to move the entire term towards the processor holding its root (thereby reducing parallelism), and hence should be avoided. This can be done by asking p_2 and p_3 to "continue the match" for both lhs. Now p_2 does not know whether to head normalize $h(\cdots)$ hoping to match c of the first lhs or consider it as a substitution for x in the second lhs. This can be decided only by p_3 after head normalizing $g(\cdots)$. So p_2 must *wait* for p_3 to *communicate* this result. Thus matching in this case leads to synchronized waiting and extra communication.

The problem illustrated above arises because p_2 is unable to decide (based only on the information received from its parent p_1) whether $h(\cdots)$ corresponds to a variable or a nonvariable of the lhs that will ultimately match at the root of s. Path sequential systems are exactly those for which each processor is able to make such a decision. Therefore in path sequential systems it is possible to search and reduce redices independently along each path without any need for synchronized waits or extra communication. They are thus appropriate for parallel evaluation.

Most programs in practice are path sequential. Even if they violate this restriction, they can be easily transformed into path sequential systems through the procedure described in [Sek89a], as long as they are strongly sequential. The transformation procedure is very efficient and in practice the transformed program is only slightly larger than the input program.

3 Parallelism in Equational Programs

Parallelism in equational program falls into two major categories:

---

term.

1. Horizontal parallelism: This kind of parallelism enables disjoint parts of a term to be reduced concurrently. For instance, it may be possible to reduce all the arguments of a function simultaneously before applying the function. More precisely, horizontal parallelism arises due to the presence of several disjoint redices. (redices $r_1, ..., r_n$ are said to be disjoint if no r_i is inside an r_j.)

2. Vertical Parallelism: This is also known as "coroutine" or "pipeline" parallelism and arises when a function and its argument can be evaluated in parallel. More precisely, this kind of parallelism arises when it is possible to reduce in parallel several redices $r_1, ..., r_n$ such that r_i is inside r_{i-1}.

These concepts are illustrated by the following examples. These examples also reveal the versatility of equational languages.

3.1 Example 1: Summing the Leaves in a Tree

This is a program to sum the *val* fields in the leaves of a tree, and is given by the following set of equations. The internal nodes in the tree are of the form $tree(left, right)$ where $left$ and $right$ are the two subtrees rooted at the children of that node. The leaves are of the form $leaf(val, x)$, where val is the field to be summed and x is whatever other information stored in the leaf.

$$\begin{aligned} sum(tree(left, right)) &\rightarrow sum(left) + sum(right) \\ sum(leaf(val, x)) &\rightarrow val \end{aligned}$$

Horizontal parallelism arises in the normalization of $sum(t)$ using these rules. This is because $sum(t)$ is immediately rewritten into $sum(t_1) + sum(t_2)$ (where t_1 and t_2 are the subtrees rooted at the children of root of t), which in turn gets rewritten into $sum(t_3) + sum(t_4) + sum(t_5) + sum(t_6)$ and so on. It is clear that the parallel execution time is proportional to the height of the tree, whereas sequential evaluation takes time linear in the number of nodes in the tree.

3.2 Example 2: Matrix Multiplication

The naive algorithm for matrix multiplication is modified so as to introduce a more "regular" flow of data, much like a systolic computation. Function *matmult* takes two matrices as its arguments: the first is input as a list whose elements are the rows of the first matrix m_1 and the second is another list whose elements are the columns of the second matrix m_2. Each row (or column) is in turn a list. The output is a list whose elements are reversed rows of the product matrix[2]. In the following, : is the cons operator.

$$\begin{aligned}
matmult(x : xs, y) &\rightarrow rowmult(x, matmult(xs, y), nil) \\
matmult(nil, y) &\rightarrow y \\[4pt]
rowmult(x, y : ys, z) &\rightarrow y : rowmult(x, ys, innerprod(x, y) : z) \\
rowmult(x, ans(w), z) &\rightarrow ans(z : w) \\
rowmult(x, nil, z) &\rightarrow ans(z : nil) \\[4pt]
innerprod(x : xs, y : ys) &\rightarrow x * y + innerprod(xs, ys) \\
innerprod(nil, nil) &\rightarrow 0
\end{aligned}$$

matmult gets rewritten into $rowmult(x_1, rowmult(x_2, rowmult(\cdots)\cdots)$, thereby creating a string of *rowmult*'s. The ith *rowmult* in this string computes the ith row of m_3, which is computed by taking the inner product of the ith row of m_1 with each of the columns of m_2. Thus the first argument to the ith *rowmult* is the ith row of m_1 and the the columns of m_2 are given to it by the $i+1$th *rowmult*. When it receives a column, it outputs the column immediately for consumption by the $i-1$th *rowmult*. It also uses this column to compute an element of the ith row of m_3. The successive elements of this row are accumulated in the third argument of *rowmult*. (The definition of *innerprod* is straight forward and is not explained here.) The innermost *rowmult* detects the end of m_2 when the third clause of *rowmult* matches. It then outputs its row of m_3, marking it with *ans* to indicate that it is *not* a part of m_2. An outer *rowmult* detects the end of m_2 when it sees the term $ans(t)$ (where t is the list of rows of m_3 computed by *rowmult*'s inner to it), whence it appends t with the row of m_3 it has computed. It then marks this term with *ans* to indicate that this is the output. Observe how the process of m_2 being "pumped through" all the processors has been captured elegantly.

Observe that the *rowmult*'s can be evaluated in parallel and thus vertical parallelism exists in this example. We call functions such as *rowmult* as pipeline functions. As result of vertical parallelism, *matmult* executes in $O(n^2)$ parallel time – a speedup of $O(n)$. We have also been able to write $O(n)$ parallel program for matrix multiplication.

3.3 Example 3: Quicksort

Quicksort exhibits both kinds of parallelisms and is described by the following equations.

$$\begin{aligned}
qs(x : xs) &\rightarrow qs1(split(xs, x, nil, nil)) \\
qs(nil) &\rightarrow nil
\end{aligned}$$

[2]Actually, the output produced is y followed by $ans(z).m_3$

$$
\begin{aligned}
split(x:xs,y,u,v) &\rightarrow if(x>y, split(xs,y,x:u,v), split(xs,y,u,x:v))\\
split(nil,y,u,v) &\rightarrow tuple(u,y,v)\\
qs1(tuple(x,y,z)) &\rightarrow append(qs(x), y:qs(z))\\
append(x:xs,y) &\rightarrow x:append(xs,y)\\
append(nil,y) &\rightarrow y
\end{aligned}
$$

qs takes an (unsorted) list as input, uses *split* to partition the list into two lists on the basis of the first element in the list and gives this to $qs1$. $qs1$ uses qs recursively on each of these sublists. The partitioning element is put at the beginning of the second list. The definition of *append* is obvious.

In this example, there is opportunity for horizontal parallelism as qs of the two halves of the list can be done simultaneously. There is scope for vertical parallelism as well, since *append* can be evaluated in parallel with qs in the term $append(qs(x), y : qs(ys))$. However, the speedup is only $O(\log n)$ on the average, since the *split* phase is essentially sequential.

4 Automatic Extraction of Parallelism

The examples in the previous section illustrate how parallelism manifests itself in equational programs. Now we need a technique to detect and utilize this parallelism automatically. If the normalizing procedure outlined in section 2 (henceforth referred to as *normalize*) is used, very little parallelism results in practice. For instance, it detects no parallelism in examples 2 and 3, and in many other sorting and matrix computations. The reasons are not difficult to find. Observe that *normalize* reduces only outermost redices and hence it can never exploit vertical parallelism. In addition, in a term such as $append(t_1, t_2)$ it will try to head normalize only t_1 and will not attempt to evaluate t_2 at all, thereby ruling out any (horizontal) parallelism. For this reason *normalize* does not exploit any parallelism in qs. In fact, the situation is still worse – even in a term $merge(t_1, t_2)$ (the definition of *merge* is given in fig. 1), it will only head normalize t_1 and t_2 and this is often not sufficient (as noted in a *mergesort* program) to extract any significant parallelism. It appears that unless these subterms are evaluated to an extent more than their head normal form, no significant parallelism can be extracted. Therefore we need a method for finding the extent to which the subterms of a term need to be evaluated. This problem has been subject of *strictness analysis*. The first systematic treatment of strictness analysis was given by Mycroft [Myc80] who developed a procedure to infer the arguments of a function that are eventually evaluated. His work, however, was limited to *flat domains* such as integers, and the extension of strictness analysis to lists has been studied in [Wad87, Lin86, Hall85]. They developed a precise formulation of the problem and proposed solutions based on a domain theoretic approach. However, the development of simple, effective and efficient method that can handle function definitions by pattern match has remained an open problem.

Terms in non-flat domains can be in several stages of evaluation: head normal form, normal form or anywhere in between. Strictness analysis in this case means inferring to what extent the arguments of a function need to be evaluated in order to evaluate the function to a particular extent. Development of a simple, effective and efficient method requires us to identify what these "extents" are. We fixed this extent to be the normal form, and hence we analyze whether a function f requires its arguments $t_1, ..., t_n$ in normal form in order to normalize $f(t_1, ..., t_n)$. This appears to be the most appropriate extent since most functions f require their arguments in normal form if any term $f(\cdots)$ is to be normalized. Once we detect that certain subterms are evaluated eventually, all these terms can be evaluated as early as possible and a large amount of parallelism can result in their simultaneous evaluation. Observe that these subterms may be disjoint or may contain one inside another and hence both vertical and horizontal parallelism can be exploited.

We have developed a method of strictness analysis based on the above notion of appropriate extent. Our method is applicable to all data types and to polymorphic functions. It is intuitive and simple. We also have a linear (in program size) heuristic algorithm for inferring strictness in most practical cases. It also appears appears to be very effective in practice. For instance, it is able to detect all the parallelism in the examples of section 2. The efficacy of our procedure stems from the fact that our notion of strictness coincides with the concept of *nonerasing*. Intuitively, a function is nonerasing in its ith argument if it does not discard parts of this argument without even examining them. For instance, *append* is nonerasing in its first argument since it *examines* the outer structure of the list and *retains* all the members of the list in its output. In contrast, *length* (see fig.1) discards the elements of its input list without examining them and hence is erasing. Observe that all constructors are nonerasing. Most functions are also nonerasing since data items are seldom created and discarded without examining them. Even though there are functions such as *length* that are erasing in their arguments, we find that these arguments are invariably used by other erasing functions.

A more detailed exposition to our strictness analysis appears in [Sek89b].

$$
\begin{aligned}
merge(x:xs, y:ys) &\rightarrow if(x > y, y : merge(x:xs, ys), x : merge(xs, y:ys)) \\
merge(x, nil) &\rightarrow x \\
merge(nil, y:ys) &\rightarrow y : ys \\[6pt]
reverse(x:xs) &\rightarrow append(reverse(xs), x : nil) \\
reverse(nil) &\rightarrow nil \\[6pt]
length(x:xs) &\rightarrow 1 + length(xs) \\
length(nil) &\rightarrow 0
\end{aligned}
$$

Figure 1: Examples

5 Load Balancing and Load Estimation

Having developed a technique to detect which parts of the input term can be normalized in parallel, we address the problem of allocating processors to do these in parallel. First we study pipeline functions, since more care needs to be taken in order to effectively utilize the parallelism they offer. We then present methods for deciding *when* a task is to be delegated to a remote processor, and in that case *how* to select this processor.

5.1 Close Look at Pipeline Functions

Pipeline functions appear to be communication intensive in general. For instance, the whole of the second input matrix and part of the product matrix are passed by a *rowmult* to its its parent *rowmult*. So it is crucial that these two *rowmult*'s be executed on processors that are adjacent (or very close to each other). Therefore such pipeline functions require special attention in order to efficiently exploit the parallelism they offer. Furthermore, we want to distinguish between pipeline functions that contribute vertical parallelism and functions such as *reverse* (see fig.1) that do not lead to any vertical parallelism (in most cases). For instance, in $f(reverse(\cdots))$, if f cannot be evaluated before *reverse* produces any output, then no parallelism can be obtained by executing f and *reverse* on different processors. This is because *reverse* produces no output at all before consuming all of its input, and hence the evaluation of *reverse* will have to be completed before that of f can begin. Knowledge about whether a function is pipelining is thus useful in deciding whether to execute a task locally or on a different processor. If f needs at least the root of its argument before it can be rewritten, and g is not pipelining, then in evaluating $f(g(\cdots))$, we can execute g and f on the same processor without losing any parallelism. So it is important to identify pipeline functions. They are characterized as follows.

Pipeline functions are recursive and consume input incrementally to produce incremental output. If f is pipelining, then a term $f(\cdots)$ can be rewritten into a t such that t contains exactly one occurrence of f in a position nonerased in t. We have an elegant technique similar to that described in section 4 to detect whether a function is pipelining or not. Our method is efficient and appears to be very effective in practice. For instance, it detects that $rowmult, append$, etc. are pipelining and that $split, reverse, sum$, etc. are not pipelining.

5.2 Criteria for Processor Selection

The decision to execute a task locally or on a different processor should be based on the "cost" of the task. An obvious way to estimate the cost of a function f is to find the maximum (or average) number of rewrites required to normalize $f(t_1, ..., t_n)$, where $t_1, ..., t_n$ are in normal form. While this computation is straight forward for functions that are not recursive (and do not make use of any recursive function in their definition), it is uncomputable in general for recursive functions. However, in keeping with the assumption made in compiler theory that "loops are executed many times", we can assume that recursive functions are very costly. Since the weights of recursive and nonrecursive functions are not comparable, we use the following simple criterion. Only recursive functions (and functions defined in terms of recursive functions) can be delegated to a remote processor and all other functions will be computed locally.

Observe that the term being normalized is a dynamically changing DAG. Suppose that a new task is created by a processor and the processors in its neighborhood are already busy. If the new task is known to spawn off several more tasks in turn, it is preferable to execute it in a neighborhood where there are a large number of free processors; if not, it should be performed within the neighborhood of the processor that created the task in order to avoid excessive communication. To aid us in the decision process, we define the concept of "parallelism a function yields". If a function yields a high degree of parallelism, then that function is performed in a neighborhood with large number of free processors; otherwise it is executed in the neighborhood of the processor that created it.

We identify "large degree of parallelism" with potentially unbounded parallelism, in the same manner as "costly functions" are those that may lead to an unbounded sequence of rewrites. A function can yield unbounded parallelism in one of the following ways. Suppose a term $f(\cdots)$ can be rewritten into a function t which contains more than one occurrence of f in positions not erased in t. It is easy to see that each of these f's could in turn be rewritten to produce more f's and this number can grow without any bound. Examples of this type of functions include $qs, qs1$ and sum. The other possibility is when a function creates a string of pipeline functions, as in the case of *matmult* which creates a string of *rowmult*'s. Specifically, this will happen if a term $f(\cdots)$ can be rewritten into a t which contains

1. a pipelining function g in a position not erased in t

2. f in a position not erased in t and within a subterm corresponding to an argument of g that is pipelined by g.

We have a very simple technique to detect such functions. This technique appears to be very effective and detects almost all such functions (e.g., *matmult*) that yield unbounded parallelism.

6 Architectural Issues

The methods we have proposed are quite independent of the underlying architecture, and indeed all the techniques described are suited for parallel architectures based on message passing. We now turn our attention to choosing an architecture that appears especially suited for parallel normalization.

Our architecture will consist of an ensemble of clusters of processors. Within each cluster processors are connected by a high-speed shared bus and the clusters in turn are connected by a low diameter message-passing interconnection network such as a hypercube or a mesh. In current and projected future technologies shared busses are much faster than processors. Processors on a shared bus can communicate very rapidly. However, traffic in shared busses grows in proportion to the number of processors on it. Current shared bus multiprocessors such as Sequent and Encore can support only upto a dozen processors on a single shared bus. Since we envision a large number processors, we need to connect each cluster (comprising a small number of processors) by a message-passing interconnection network. This network must have low diameter for a given number of links per node. Thus a hypercube with each node consisting of processors sharing a high-speed appears to be an appropriate architecture for parallel evaluation of equational programs.

7 Work in Progress

Currently, simulations are being carried out on the proposed architecture. In a few months we expect simulation results on the efficacy of the schemes we have proposed. In particular, we are studying the actual speed-ups achieved and how much of the available parallelism is exploited by our methods.

References

[Flag88] IAN WATSON ET AL, Flagship: A parallel architecture for declarative programming *International Conf. on Computer Architecture, June 1988.*

[Gog77] JOSEPH A. GOGUEN, Abstract errors for Abstract Data Types *IFIP Working Conference on Formal Description of Programming Concepts, 1977, pp. 491-522.*

[Gold88] BENJAMIN GOLDBERG, Multiprocessor execution of Functional Programs, *PhD thesis, Dept of Computer Science, Yale University.*

[Hall85] CORDELIA V. HALL AND DAVID S. WISE, Compiling Strictness into Streams, *14th ACM Symp. on Principles of Programming Languages, 1985, pp. 132-143.*

[HD82] C. HOFFMAN AND M.J. O'DONNELL, Programming with Equations, *ACM Transactions on Programming Languages and Systems (1982) pp. 83-112.*

[HL79] G. HUET AND J.J. LEVY, Computations in Nonambiguous Linear Term Rewriting Systems, *Tech. Rep. No. 359(1984), INRIA, Le Chesney, France.*

[Lin86] GARY LINDSTROM, Static Evaluation of Functional Programs, *Proc. of SIGPLAN'86 Symp. on Compiler Construction, SIGPLAN Notices* **21**, *7, 1986, pp. 196-206.*

[Myc80] A. MYCROFT, The Theory and Practice of Transforming call-by-need into call-by-value, *International Symposium on Programming, Springer, April 1980.*

[OBJ2] K. FUTATSUGI, JOSEPH A. GOGUEN, JEAN-PIERRE JOUANNAUD, AND JOSE MESEGUER, Principles of OBJ2, *Proc. 12th ACM Symposium on Principles of Programming Languages, 1985.*

[OD77] M.J. O'Donnell, Computing in systems described by Equations *Lecture Notes in Computer Science, v.58, Springer Verlag (1977).*

[OD85] M.J. O'Donnell, Equational Logic as a Programming Language, *MIT Press (1985).*

[Sek89a] R.C. Sekar, Shaunak Pawagi and I.V. Ramakrishnan, Transforming Strongly Sequential Rewrite Systems with Constructors for Efficient Parallel Execution, *International Conference on Rewriting Techniques and Applications, 1989.*

[Sek89b] R.C. Sekar, Shaunak Pawagi and I.V. Ramakrishnan, Small Domains Spell Fast Strictness Analysis, *Technical Report 89/09, Dept. of CS, SUNY at Stony Brook, New York, 1989.*

[Sek89c] R.C. Sekar and I.V. Ramakrishnan, Optimal Parallel Evaluation of Equational Programs, *Technical Report 89/15, Dept. of CS, SUNY at Stony Brook, New York, 1989.*

[That84] Satish Thatte, Demand driven Evaluation with Equations, *Technical Report CRL-TR-34-84, University of Michigan Computing Research Laboratory, Ann Arbor.*

[Wad87] Philip Wadler and R.J.M. Hughes, Projections for Strictness Analysis, *3rd Int. Conf. on Functional Programming and Computer Architecture, 1987.*