Compilation of Concurrent Declarative Languages

Z. M. Ariola, M. Sami, B. C. Massey, and E. Tick
Dept. of Computer Science
University of Oregon
Eugene OR, USA 97403
{ariola, sami, bart, tick}@cs.uoregon.edu

Abstract

The plethora of concurrent declarative language families, each with subtly different semantics, makes the design and implementation of static analyses for these languages a demanding task. However, many of the languages share underlying structure. If this structure can be exploited, static analysis techniques can be shared across language families. These techniques can thus provide a common kernel for the implementation of quality compilers for this entire language class.

The purpose of this paper is to exploit the similarities of non-strict functional and concurrent logic languages in the design of a common intermediate language (CIL). The CIL is introduced incrementally, giving at each step the rationale for its extension. As an application, we have analyzed some state-of-the-art static partitioning algorithms from the literature in terms of their action upon CIL programs. This allowed us to "uncover" the relative advantages and disadvantages of the analyses, and determine promising directions for improving static partitioning.

1 Introduction

The compilation of concurrent declarative languages, such as Id and Strand, for high-performance multiprocessors depends fundamentally on increasing task granularity (partitioning) and hiding various latencies such as memory accesses. The plethora of language families, each with subtly different semantics, makes the design and implementation of static analyses, the backbone of a good compiler, a demanding task. However, many of the languages share underlying structure, and if this structure can be exploited, static analysis techniques can be shared across language families.

The process of identifying portions of a program that can be executed as threads, referred to as *partitioning*, has been a subject of growing interest to the implementors of concurrent logic programming languages and functional languages, here referred to collectively as concurrent declarative languages. In this paper, the important issues concerning partitioning for both concurrent logic programming languages and non-strict functional languages are presented, and some of the important partitioning techniques are discussed. The similarities between non-strict functional languages with single-assignment features and committed-choice languages with logic variables lead us to believe that partitioning techniques applicable in one language domain will be successful in the other. The partitioning algorithm and its associated static analyses are the kernel of any high-performance compiler for concurrent declarative languages. We believe that a good first step in understanding the state-of-the-art in partitioning and how to improve it is to define a common intermediate language (CIL) for both non-strict functional and concurrent logic languages, and evaluate partitioning algorithms with respect to the CIL.

This paper is organized as follows. Section 2 reviews the family of concurrent declarative languages. Section 3 defines a CIL covering a slightly restricted subset of this family. Section 4 reviews the partitioning problem and three algorithms proposed in the literature. This comparison reveals strong and weak points in each approach, and suggests directions for future exploration. A summary of the research, conclusions, and discussion of future work is given in Section 5.

2 Review of Concurrent Declarative Languages

This paper focuses on the compilation of concurrent declarative languages containing what we refer to as *dynamic-single-assignment (DSA) variables.* This family includes Id [21], pH (a parallel version of Haskell) [23, 5], Strand [11], and many others. DSA variables correspond to the notion of "I-structures" in Id [6], and to the "logic variables" of Strand. A DSA variable begins its life unbound, and may be bound only once during program execution. However, in contrast to static-single-assignment variables, the point at which a DSA variable is bound is not in general statically determinable: binding can occur at any time during program execution. Attempts to read a DSA variable before it is bound will typically cause the reader to suspend. Once bound, attempts to rebind a DSA variable will "fail," and will typically cause execution of the entire program to fail.[1]

In this section, we review two languages with DSA variables, chosen from the two language families of principle interest in this paper: the concurrent functional language Id, and the concurrent logic programming language Strand. We note some of the similarities between these languages, and thus motivate the introduction of our CIL.

Id is a higher-order, non-strict, strongly-typed functional language, augmented with I-structures [6] and M-structures [7]. I-structures are arrays whose elements get "refined" during the course of computation: that is, it is possible to create an I-structure without giving a definition for each of its elements. (In a purely functional language a variable is given exactly one binding or definition at creation time.) For our purposes, single-element I-structures are DSA variables [19]. M-structures add non-determinism to the languages; in this paper, however, we will not address this feature. Id has non-strict semantics, that is, a function is invoked even though its parameters may not be completely evaluated yet. However, unlike other modern functional languages, non-strictness is not implemented in terms of lazy evaluation. Id has a *parallel evaluation strategy*, that is, all computations are performed in parallel, not merely those whose results are sure to be needed. Thus, the parameters of a function are all evaluated in parallel with the evaluation of its body, even though the body might not need some parameter. For example, consider the following Id definitions:

$$\mathsf{Def}\ nth\ (x : xs)\ n = \mathsf{If}\ n == 1\ \mathsf{then}\ x\ \mathsf{else}\ nth\ xs\ (n-1)$$
$$\mathsf{Def}\ Ints\ x = x : (Ints\ (x+1))$$

and the main expression $(nth\ (Ints\ 1)\ 5)$. (Note: The expression $x : xs$ denotes a list whose head is x and tail is xs.) The function nth is invoked even though the evaluation of the first parameter does not terminate. The evaluation of the main expression will return 5 as a result. This is accomplished by evaluating the expression $Ints\ 1$ in parallel with the body of the nth function. Note that the main expression returns a value even though its evaluation does not terminate. In this specific example, we can safely assume that 5 is the final answer, however, in the general case, due to the presence of single-assignment variables, the result must be interpreted as a partial answer, that is, the answer could be refined further. In particular, as mentioned before, the partial answer could be refined to failure in case of multiple bindings; a condition that cannot be checked statically.[2]

All variables in a Strand program are DSA variables, although some may be bound immediately upon creation. Syntactically, a program consists of a sequence of procedures, each of which consists of a sequence of *clauses*. Each clause is of the form "$Head$:- $Guard$ | $Body$" where the head is a term and the guard and body are sequences of terms. Execution is defined by *goals* corresponding to procedure invocations, also called tasks and processes. Each goal specifies a procedure and its arguments: the head and body of each clause of the procedure are matched against the goal. If one or more clauses match the goal, one clause is selected for *commitment*. The committed clause's body will consist of a (possibly empty) sequence of new goals, which are all evaluated concurrently. If a goal fails to match because not enough information is available, the goal will suspend. If the goal fails to match for other reasons, the goal will cause failure of the program. A program may also fail by *deadlock* which occurs when all goals have suspended. Finally, as mentioned above, a program may fail if a DSA variable is assigned to a second time, thus informally there are three sources of program failure.

[1] With regard to logic programs, we restrict ourselves to committed-choice languages with "eventual" tell unification [24] or assignment, rather than "atomic" tell unification, thus implying that binding failure causes program failure.

[2] The reader may refer to Ariola and Arvind [2] for further clarification. The parallel evaluation of Id has been adopted by the newly developed pH language, a parallel variant of the Haskell language [13]. Haskell is increasingly the standard language in the non-strict functional language research community.

There are marked similarities among the members of this language family. Execution is described in terms of *eager concurrency*: all program units will be executed concurrently as soon as they are reached, the only source of delay being suspension due to an attempt to read an unbound DSA variable. The concurrency in the languages is very fine-grained — many concurrent tasks will be spawned, and each of these tasks will be relatively small, such as a goal or function evaluation. Finally, the fact that tasks suspend when trying to read an unbound DSA variable implies that the languages are subject to the possibility of deadlock, and thus the semantics are truly concurrent. If an attempt is made at compile time to decide the order of execution of tasks (*e.g.,* for uniprocessor execution), care must be taken to ensure that some latter task will not provide a binding which some earlier task needs to proceed.

This problem of static scheduling to increase task granularity while avoiding deadlock is a main component of the work of compilation for the execution of these languages on high-performance multiprocessors [15, 12, 22, 18]. Comparative analysis of different partitioning techniques is best achieved by first developing an intermediate language common to the DSA language family. A common language allows us to accurately compare currently proposed analyses, as well as develop a single, unified analysis technique. The latter will help concurrent declarative language researchers to avoid unconsciously duplicating past research. The advantage of defining an intermediate *language* as opposed to an intermediate *form* (*i.e.,* a non-executable structure) is that the operational semantics of the language can assist us in formalizing and showing correctness of the static analyses.

The common intermediate language (CIL) or *kernel* language presented attempts to draw together the functional and logic programming communities. By sharing a CIL foundation, it will be easier to share ideas and implementations, exchange programs and reduce harmful diversity. However, we do not seek uniformity. There are clear differences between these languages. But having a common framework will allow us more easily to highlight and study these differences. In the next section, we develop the CIL incrementally, with a rationale for each extension.

3 Common Intermediate Language

In this section we introduce the main features of our kernel language through examples. In particular, we are interested in capturing the essential features needed to support DSA variables. At the outset we assume that a CIL program produces a unique result, independently of evaluation order. Thus, for instance, we disallow Prolog programs that produce multiple solutions, and concurrent logic procedures with non-mutually exclusive guards that are truly indeterminate, *e.g.,* stream mergers (which can easily be relegated to system-defined builtins).

Both concurrent logic languages and functional languages have gone through some evolution (and deevolution [27]) to support both parallelism and determinacy. Functional languages, on the one hand, restrict the handling of state, while concurrent logic languages restrict unification and backtracking. Consider the Prolog program in Figure 1.

Given a list of numbers, sum computes the sum of its elements. Two selection rules are in play: clauses are selected top-down, and body goals are selected in left-to-right order. Backtracking undoes bindings that lead to local and nonlocal failures. In this example, even if the body goals of sum' were reversed, or the clauses of plus were reordered, a query with a ground input list would be guaranteed by backtracking to produce a correct solution (although possibly very inefficiently!)

We can rewrite the above example program in a committed-choice language, removing backtracking and restricting the clause selection to matching. In Parlog [8] or Strand the program remains textually identical to the one in Prolog, with the inclusion of *mode declarations*: $\mathsf{sum}(?,?)$, $\mathsf{sum}'(?,?,\hat{\ })$, and $\mathsf{plus}(?,?,\hat{\ })$. The input mode '?' indicates that the passed parameter must be bound and the '^' indicates that the argument binding is produced by the procedure itself. Other committed-choice languages indicate these *ask* and *tell* unifications in syntactically different ways, but the idea is the same.

In functional language terminology, we say that plus is *strict* in both of its input arguments. We can translate the above clauses into our kernel language as shown in Figure 2.

A typical query is written as $\{Res = \mathsf{sum}([1,2]) \text{ in } Res\}$. The kernel language semantics allows a variable to appear on the left-hand side of only one equation (the output instance), and on the right-hand side of multiple equations (the input instances). The key point enabling this translation is the determination of

$$
\begin{array}{ll}
\mathsf{sum}(Xs, S) \leftarrow \mathsf{sum}'(Xs, 0, S). & \S 1 \\
\\
\mathsf{sum}'([\,], S, S). & \S 2 \\
\mathsf{sum}'([X \mid Xs], P, S) \leftarrow \mathsf{plus}(X, P, P'),\ \mathsf{sum}'(Xs, P', S). & \S 3 \\
\\
\mathsf{plus}(0, 0, 0). & \\
\mathsf{plus}(0, 1, 1). & \\
\mathsf{plus}(1, 0, 1). & \\
\mathsf{plus}(2, 0, 2). & \\
\mathsf{plus}(2, 1, 3). & \\
\quad\vdots &
\end{array}
$$

Figure 1: Sample Prolog Program

$$
\begin{array}{llll}
\mathsf{sum}(Xs) \rightarrow \mathsf{sum}'(Xs, 0) & & & \S 1 \\
\mathsf{sum}'([\,], P) \rightarrow P & & & \S 2 \\
\mathsf{sum}'([X \mid Xs], P) \rightarrow \{ & P' & = \ \mathsf{plus}(X, P) & \S 3 \\
 & S & = \ \mathsf{sum}'(Xs, P') & \\
 & \mathsf{in} & S\ \} & \\
\\
\mathsf{plus}(0, 0) \rightarrow 0 & & & \\
\mathsf{plus}(0, 1) \rightarrow 1 & & & \\
\mathsf{plus}(1, 0) \rightarrow 1 & & & \\
\mathsf{plus}(2, 0) \rightarrow 2 & & & \\
\mathsf{plus}(2, 1) \rightarrow 3 & & & \\
\quad\vdots & & &
\end{array}
$$

Figure 2: Sample Program in Kernel Language

variable modes. This is difficult to do in many cases, and impossible in general for an unrestricted language, as we discuss in the next section.

Note that the kernel language is essentially a *graph rewriting system* [1, 4]: each clause is translated into a rewrite rule. Note also that in order to respect the communication pattern expressed in a logic program via logical variables, the kernel language must allow assignment of a name to each computation. For example, in the atom $\mathsf{plus}(X, P, P')$, P' denotes the result of the operation. We make this explicit by saying $P' = \mathsf{plus}(X, P)$. Thus the right-hand side of our rewrite rules are not just simple terms, as in rule §1, but can be a set of equations of the form $\alpha_1 = t_1, \cdots, \alpha_n = t_n$, where each variable is distinct from each other. The in keyword denotes the result of the term: for example, the result of rule §3 in Figure 2 is S. Note that in the term $\{\alpha_1 = t_1, \cdots, \alpha_n = t_n \ \mathsf{in}\ \alpha_i\}$ the order of the equations is irrelevant. We discuss graph rewriting semantics in Section 3.3, after first dealing with mode analysis.

3.1 Mode Analysis

In logic programs, including committed-choice programs, no syntactic distinction is made between variables and their subterms which have *output mode* and are thus being bound in an occurrence, and those which have *input mode* and are thus being read in an occurrence. Thus, some form of mode analysis (*e.g.,* [28, 31]) or mode declarations [26] are required for translation into the kernel language. However, these methods are not perfect. For example, array elements can be aliased by indexing accesses, which means we might not be fully knowledgeable about all modes in an array. Furthermore, without good general aliasing analysis, these unknown array modes could "contaminate" the remainder of the program (to date, no mode analyzers do the sort of aliasing analysis required to effectively control this contamination).

Another point is that a certain class of automatic mode analyzers require language restrictions based on the concept of *permissible modes* [31, 25]. Sundararajan [25] defines a static analysis technique for determining a minimal set of permissible modes for a Prolog program. This is complicated by the lack of restrictions on Prolog unification. As discussed earlier, committed-choice languages restrict unification to ask and tell, making mode analysis considerably less complex. To further simplify the analysis, Ueda and Morita [31] restrict the language family to a "fully-moded" subset wherein each logic variable can have only a single producer. Furthermore, the subset requires equal modes in corresponding argument positions of a procedure's defining clauses (and all corresponding subterms *within* those arguments). These restrictions enable efficient, practical mode analysis [28].

Note that in the previous example, while in the concurrent logic program the processes spawned in clause §3 communicate through the logical variable P', in our setting the communication is rendered via a data dependency. This provides a hint for later phases of compilation that it is safe to sequentialize plus and sum'. However, if the implementation desires to execute $\mathsf{plus}(X, P)$ and the call to sum' in parallel then P' will have to be implemented as a DSA variable. The key point is that instead of restricting programs to use only DSA variables (as in concurrent languages), programs written in the kernel language may explicitly indicate points where synchronization is not needed to respect the language semantics. The kernel language implementation may still eliminate some synchronization points through static analysis, as well as inserting synchronization points if desired.

There are programs that cannot be consistently moded, and thus cannot be translated into the kernel language as it now stands. Furthermore, even fully-moded programs can have variable occurrences in which some portion of the occurrence's term structure is read, and another portion written. This condition also precludes translation. We discuss an extension to deal with this in Section 3.5, after first discussing the basic rewriting semantics.

3.2 Feedback Analysis

An additional level of analysis needs to be done on the Strand program before it can be automatically translated into the CIL in a useful way. As things stand, it is impossible to replace the DSA variables of a Strand program with "ordinary" CIL variables, as critical aliasing information is lacking. Without this information, pathological invocations of a procedure might deadlock, unless DSA variables are used. However, since the translator has no way of knowing which DSA variables can safely be replaced, none can, and thus partitioning analysis would produce poor results.

The M.S. thesis of Massey [20] suggests a way around these difficulties. The mode information obtained by the analysis of Section 3.1 can be used to check whether any procedure invocation "produces some of its own arguments," by inspecting the mode dependency graph for cycles. If a cycle exists, we refer to this phenomenon as "feedback." For the purposes of this paper we will restrict ourselves to "Feedback-Free Strand" (FFStrand) programs, as it is possible to translate *every* DSA variable in such programs into an ordinary CIL variable. This is clear from the fact that, in such programs, a static schedule can be found which identifies the binding sites of all variables at compile time.

3.3 Rewrite Semantics

Let us now present the execution of a query to the example of the previous section, following a rewrite semantics. We start out with the term $\{Res = \mathsf{sum}([1,2]) \text{ in } Res\}$, by applying rule §1 we obtain:

$$\{Res = \mathsf{sum}'([1,2],0) \text{ in } Res\}$$

by applying rule §3 we have:

$$\begin{array}{lcl} \{\ Res & = & \{\ P'' \ = \ \mathsf{plus}(1,0) \\ & & \quad\;\; S' \ = \ \mathsf{sum}'([2],P'') \\ & & \qquad\qquad \text{in } S'\} \\ \;\;\text{in } Res\} & & \end{array}$$

By pattern matching $\mathsf{sum}'([1,2],0)$ with the left-hand side of rule §3 we have created the bindings $X = 1$, $Xs = [2]$: we then create an instance of the right-hand side of the rule. This corresponds to the renaming done before matching in concurrent logic languages, and can also be seen as the allocation of a frame in the

$$
\begin{array}{lcl}
SE & \in & \text{Simple Expression} \\
\\
Variable & ::= & x \mid y \mid z \mid \cdots \mid a \mid b \mid \cdots \mid f \mid \cdots \mid x_1 \mid \cdots \\
Constant & ::= & Integer \mid Boolean \mid \cdots \\
SE & ::= & Variable \mid Constant \\
Term & ::= & SE \mid \mathsf{f}^n(SE_1, \cdots, SE_n) \\
Block & ::= & \{[Variable = Term]^* \ \mathsf{in}\ SE\} \\
Definitions & ::= & Term \rightarrow Block \mid Term \rightarrow Term \\
Program & ::= & Block
\end{array}
$$

Figure 3: The Grammar of Kernel_0

context of a procedure call. In the kernel language we want to eliminate all syntactic sugar, and thus we will consider the above program equivalent to:

$$
\begin{array}{lcl}
\{\ Res & = & S' \\
\ \ P'' & = & \mathsf{plus}(1,0) \\
\ \ S' & = & \mathsf{sum}'([2], P'') \\
\ \ \mathsf{in}\ Res\}
\end{array}
$$

That is, the internal nesting of blocks does not matter. In the above program we can now execute, for example, the third expression, obtaining:

$$
\begin{array}{lcl}
\{\ Res & = & S' \\
\ \ P'' & = & \mathsf{plus}(1,0) \\
\ \ S' & = & \mathsf{sum}'([\,], P''') \\
\ \ P''' & = & \mathsf{plus}(2, P'') \\
\ \ \mathsf{in}\ Res\}
\end{array}
$$

No conflict of names will arise, because at each step of the execution new names are introduced. Note also that $\mathsf{plus}(2, P'')$ cannot be executed until P'' gets a value, that is, until $\mathsf{plus}(1,0)$ has executed.

There is a major difference between the computational model presented here and the one usually employed to give the operational semantics of concurrent languages. In our approach we do not keep a separate component describing the current value of each variable. Instead, these "substitutions" are kept directly in the term itself. Our preliminary kernel language is basically a system of recursive equations over a first-order signature Σ, as illustrated in Figure 3. Note that each functional symbol is applied to simple expressions only: that is, we assume that each expression has been given a name. It is possible to show that the term-rewriting system of Figure 3 is *confluent*, though this is outside the scope of this paper.

3.4 Multiple Values

Concurrent logic procedures can also return multiple values, as shown in the following example:

$$
\begin{array}{l}
\mathsf{distribute}([\mathsf{send}(1,X) \mid Xs], Out1, Out2) \leftarrow Out1 = [X \mid Out1'], \mathsf{distribute}(Xs, Out1', Out2). \\
\mathsf{distribute}([\mathsf{send}(2,X) \mid Xs], Out1, Out2) \leftarrow Out2 = [X \mid Out2'], \mathsf{distribute}(Xs, Out1, Out2'). \\
\mathsf{distribute}([\,], Out1, Out2) \leftarrow Out1 = [\,], Out2 = [\,].
\end{array}
$$

In order to translate the above clauses we need to extend Kernel_0 by allowing multiple return values. The need for multiple-return-value facilities in intermediate languages has been illustrated in the context of functional languages [3]. A new language, Kernel_1, is obtained by changing the production of the syntactic category *Block* as follows:

$$
Block ::= \{[Variable \cdots Variable = Term]^* \ \mathsf{in}\ SE \cdots SE\}
$$

Thus the translation of the above program will be:

$$
\begin{array}{lll}
\mathsf{distribute}([\mathsf{send}(1,X) \mid Xs]) \leftarrow \{ & Out1 & = \; [X \mid Out1'] \\
 & Out1' \; Out2 & = \; \mathsf{distribute}(Xs) \\
 & \quad \mathsf{in} \; Out1 \; Out2\} & \\
\mathsf{distribute}([\mathsf{send}(2,X) \mid Xs]) \rightarrow \{ & Out2 & = \; [X \mid Out2'] \\
 & Out1 \; Out2 & = \; \mathsf{distribute}(Xs) \\
 & \quad \mathsf{in} \; Out1 \; Out2\} & \\
\mathsf{distribute}([\,]) \rightarrow [\,]\,[\,]. & &
\end{array}
$$

If a concurrent program is fully moded and furthermore, if no occurrence of a variable corresponds to multiple modes, then the program can be translated into a Kernel_1 program. In the next section, we tackle the problem of translating programs that do not meet these constraints.

3.5 Dynamic-Single-Assignment (DSA) Variables

Processes synchronize via DSA variables. However, they cannot make non-deterministic choices based on the availability of data. Consider the following program:

$$
\begin{array}{l}
\mathsf{counter}(X) \leftarrow \mathsf{counter}'(X, 0). \\
\\
\mathsf{counter}'([\mathsf{clear} \mid Xs], C) \leftarrow \mathsf{counter}'(Xs, 0). \\
\mathsf{counter}'([\mathsf{add} \mid Xs], C) \leftarrow C' := C + 1, \mathsf{counter}'(Xs, C'). \\
\mathsf{counter}'([\mathsf{read}(X) \mid Xs], C) \leftarrow X = C, \mathsf{counter}'(Xs, C). \\
\mathsf{counter}'([\,], C).
\end{array}
$$

Analysis can detect that the (first argument) stream is *input* and also the variable C is *input*; however, the stream contains further variables that need to be instantiated. Kernel_1 does not yet allow this, since we do not have the concept of an unbound variable. Instead, a variable is always associated with a particular value, and we do not have the possibility of allocating some variable and later initializing it. Thus, we now extend the kernel language with the concept of DSA variables. A DSA variable defines an unbound variable, it is allocated via the Mk-Var primitive. Associated with a DSA variable there are two operations: *read* and *write*. The *read* operation fetches the value of a DSA variable: however, if no value has yet been given to the DSA variable, the *read* is suspended until a *write* operation gives the variable a value. Moreover, if an attempt is made to *write* a DSA variable that has already been written, an error will be raised and execution of the program will fail. We thus extend the set of predefined function symbols with two new operators: Store of arity two (a DSA variable and the value to be stored), and Select, of arity one (the DSA variable). We may write $A = v$ instead of $\mathsf{Store}(A, v)$. Analogously, we may write A instead of $\mathsf{Select}(A)$. Finally, we will relax the restriction that a block must return a value, since some blocks may be invoked only for their side-effects. We use the special identifier "_" as indication that the expression does not return a meaningful result.

The translated program will be:

$$
\begin{array}{lll}
\mathsf{counter}(X) \rightarrow \mathsf{counter}'(X, 0). & & \\
 & & \\
\mathsf{counter}'([\mathsf{clear} \mid Xs], C) \rightarrow \mathsf{counter}'(Xs, 0). & & \\
\mathsf{counter}'([\mathsf{add} \mid Xs], C) \rightarrow \{ & C' = & C + 1 \\
 & \_ = & \mathsf{counter}'(Xs, C')\} \\
\mathsf{counter}'([\mathsf{read}(X) \mid Xs], C) \rightarrow \{ & X = & C \\
 & \_ = & \mathsf{counter}'(Xs, C)\} \\
\mathsf{counter}'([\,], C) \rightarrow \{\,\}. & &
\end{array}
$$

For example, we can invoke the counter with the following expression:

$$
\begin{array}{lll}
\{ \; a & = & \mathsf{Mk\text{-}Var} \\
\;\; b & = & \mathsf{Mk\text{-}Var} \\
\;\; \_ & = & \mathsf{counter}(\mathsf{add} : \mathsf{add} : \mathsf{read}\; a : \mathsf{add} : \mathsf{read}\; b : nil)\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{Variable} & ::= & x \mid y \mid z \mid \cdots \mid a \mid b \mid \cdots \mid f \mid \cdots \mid x_1 \mid \cdots \\
\mathit{Constant} & ::= & \mathit{Integer} \mid \mathit{Boolean} \mid \cdots \\
\mathit{SE} & ::= & \mathit{Variable} \mid \mathit{Constant} \\
\mathit{Term} & ::= & \mathit{SE} \mid \mathsf{f}^n((\mathit{SE})_1, \cdots, (\mathit{SE})_n) \mid \mathsf{Select}(\mathit{SE}) \mid \mathsf{Mk\text{-}Var} \\
\mathit{Block} & ::= & \{[\mathit{Statement}]^* \ \ \mathsf{in}\ \mathit{SE} \cdots \mathit{SE}\} \\
\mathit{Statement} & ::= & \mathit{Binding} \mid \mathit{Command} \\
\mathit{Binding} & ::= & \mathit{Variable} \cdots \mathit{Variable} = \mathit{Term} \\
\mathit{Commands} & ::= & \mathsf{Store}(\mathit{SE},\ \mathit{SE}) \\
\mathit{Definitions} & ::= & \mathit{Term} \rightarrow \mathit{Block}\ \ \mathit{Predicate} \mid \mathit{Term} \rightarrow \mathit{Term}\ \ \mathit{Predicate} \\
\mathit{Predicate} & ::= & <(\mathit{SE}, \mathit{SE}) \mid >(\mathit{SE}, \mathit{SE}) \mid \cdots \\
\mathit{Program} & ::= & \mathit{Block}
\end{array}
$$

Figure 4: The Grammar of Kernel_2

As with Kernel_1, it is possible to show that the extended term-rewriting system is *confluent.*

In order to express the notion of guards present in concurrent logic languages we need to further extend the kernel language with the concept of conditional rewrite rules. Thus for example, we will allow a rule to be expressed as: $l \rightarrow r\ \ P(x)$, where $P(x)$ is a "flat" predicate, *i.e.,* made of builtin predicates such as $\leq$. We will say that the above rule is applicable only if $P(x)$ is true. Note that all the free variables in $P(x)$ should be bound at the time $P(x)$ is evaluated, that is, no synchronization is assumed. The syntax of our final kernel language, Kernel_2, is given in Figure 4.

4 Partitioning Techniques

Compile-time partitioning of concurrent declarative languages is difficult, because any partitioning technique must be able to significantly increase the thread size while avoiding the accidental introduction of deadlock. Sometimes the correct order of execution of two processes can only be determined dynamically. Such processes have to be identified and placed in separate threads. This is especially intricate for concurrent logic programming languages, because of the prevalence of logic variables, which can cause hidden cycles through *aliasing.*[3] Non-strict functional languages have analogous problems with hidden cycles through I-structures.

Several researchers have explored the problem of thread partitioning of non-strict functional languages [12, 15, 22, 30]. While most of the techniques produce threads within a function and in isolation from the rest of the program, the analysis of Traub *et al.* [30], referred to in this paper as *Dependence-Demand (DD) analysis*, attempts to improve the thread size within a function by propagating global dependence information. The analysis we call *AB analysis* (after the sets A and B used in the analysis), proposed by Coorg [9], is an extension of DD analysis which achieves better partitioning across recursive calls. Partitioning techniques have also been proposed in the context of granularity analysis for concurrent logic programming languages [10, 17, 29]. The most significant of these is the analysis proposed by King and Soper [16], referred to in this paper as *coloring analysis*, which uses global dependence information to produce partitions.

The similarities between non-strict functional languages with I-structures and committed-choice languages with logic variables lead us to believe that partitioning techniques applicable in one language domain will be successful in the other. While the techniques developed for different language families differ in their methodology and input, there also exist some similarities. The translation of both non-strict functional languages and committed-choice languages into $Kernel_2$ (introduced in Section 3) provides a common framework for examining the different analysis techniques. This facilitates "uncovering" the relative advantages and disadvantages of the analyses, and determining future directions for developing improved partitioning analyses.

In Section 4.1, we present the definitions of thread which are used in the analyses. We present the three analyses discussed above in Sections 4.2, 4.3, and 4.4. While space limitations preclude a detailed explanation, we have formulated all of these analyses in some detail as algorithms operating upon the CIL,

[3] Aliasing occurs when two or more identifiers are simultaneously bound to the same variable

and considered their action upon example CIL programs. The similarities and differences between the three partitioning techniques are discussed in Section 4.5.

4.1 Thread Definitions

The proposed techniques for partitioning differ in their formalization of the notion of "thread." This difference has significant influence on the results produced and the complexity of the techniques. It is thus important that the notion of "thread" used in each technique be clearly defined.

- *Liberal Thread:* A thread is defined to be a totally ordered set of atomic actions of a program. Every function may have one or more threads. Thus a thread can itself include calls to other threads, and also include *split-phase*[4] memory accesses, which implies that a thread may have to suspend and wait for other threads to complete before it can continue. (The inclusion of function calls in a thread could lead to more than one thread being spawned concurrently in order to execute the callee. A stricter definition might require the callee to be completely sequentialized.)

  The liberal definition is commonly used in partitioning concurrent logic programming languages, because it corresponds well to their operational semantics: since the binding of logical variables is typically used for thread synchronization, threads will typically suspend until some logical variable is bound, and then resume execution.

- *Conservative Thread:* A thread is defined to be a subset of the atomic actions comprising a program, such that: 1) the thread is contained within a single function; 2) a compile-time ordering can be determined for the thread which is valid in all contexts in which the containing function can be invoked, and 3) once the first instruction in a thread is executed, it is always possible to execute each of the remaining instructions in the compile-time ordering without pause, interruption, or execution of instructions from other threads.

  The conservative definition is used in partitioning "dataflow" languages. Synchronization only occurs at the start of the thread, and a thread, once scheduled, runs to completion. Thus, function calls and split-phase memory accesses result in multiple threads. Also, multiple clauses result in separate threads for each clause.

4.2 DD Analysis

DD analysis produces partitions by grouping together terms which are dependent on the same set of inputs to a function or demanded by the same set of outputs produced by the function. The set of inputs which are required to compute a term is referred to as its *dependence set*, and the set of consumed output values of the term is referred to as its *demand set*. Putting terms with the same dependence set in the same partition is always safe, as it does not lead to any additional dependencies being introduced in the partition which can compromise the execution behavior of the program. Terms with the same demand set can also be placed in a single partition, as all of them must certainly be evaluated before execution can proceed.

When a partition is created by grouping terms with the same dependence set, the demand set of the partition is given by the union of the demand sets of the terms in the partition. The dependence set of a partition is created by grouping terms with the same demand set is obtained in a similar manner. It may happen that the modified dependence sets or demand sets of some of the partitions are the same, allowing the process to be repeated. Thus the steps of creating partitions by equality of dependence sets and demand sets are iteratively applied until a fixpoint is reached. This iterative process is referred to as *basic-block partitioning* because it is applied to the blocks of a function.

As a result of basic-block partitioning, terms which produce different input arguments to another function (callee) could be placed in the same partition. The terms in the callee using these input arguments can then be placed in the same partition. Also, basic-block partitioning of the callee can result in different terms in the callee which produce result values being placed in the same partition. This in turn allows terms in the caller which use these result values to be placed in the same partition. This information is propagated

[4] Split-phase memory accesses involve separate request and response messages, potentially long latencies, and potentially out-of-order responses.

across function call boundaries, allowing improvement in the partitioning of both the caller and callee. Since any change in partitioning changes the dependence and demand characteristics across the function call boundaries, partitioning and propagation across the boundaries are iteratively applied until a fixpoint is reached. This process is referred to as *global analysis*.

4.3 Coloring Analysis

Coloring analysis, developed for concurrent logic programming languages, sequentializes all terms of a clause, unless the data dependence between two terms is bidirectional or can only be determined at run-time. In addition, terms without such constraints may still be placed in different partitions, in order to control granularity of the resulting partitions. The main concern of the technique is to identify pairs of terms whose dependence cannot be determined at compile time. Dependence between terms is inferred by identifying the input and output arguments of the terms. A term t' is dependent on another term t if one or more of the inputs of t' are produced by t. If the dependence is bidirectional, or if the directionality can only be determined at run-time, the terms are placed in separate partitions to allow correct execution. The partitions are determined by k-coloring the dependence graph of the terms of a clause, and hence the name *coloring analysis*. DD analysis groups terms into partitions only when it can deduce dependence between them. In contrast, coloring analysis groups terms into partitions if it can conclude that placing them in the partition would not create cyclic dependencies. Thus unlike DD analysis, it would place two terms which do not have any interdependence in the same partition.

Higher granularity threads lead to lower process creation, management, and communication overheads. However, excessive sequentialization could be detrimental to performance, because useful parallelism may be lost. Unlike DD and AB analyses, coloring analysis attempts to control the granularity of the threads by performing a very simple granularity analysis [16]. This is done by classifying terms as constant, linear, or non-linear depending upon the estimated cumulative difference between the growth of computation and communication during the execution lifespan. The constant class corresponds to terms which are builtins, or are defined in terms of constant terms. The linear class corresponds to either non-recursive or linearly recursive functions. These are fine-grained, whereas the non-linear class (neither constant nor recursive) corresponds to functions which are typically coarse-grained. This classification gives a simple prescription for controlling granularity: two terms which are non-linear are allocated to different partitions, whereas any other pair of terms can potentially be allocated to the same partition.

4.4 AB Analysis

The DD partitioning technique does not handle recursion satisfactorily. The AB partitioning technique proposed by Coorg [9] attempts to alleviate this drawback by incorporating the recursion handling property of *strictness analysis* [32].[5] This technique uses *paths*[6] as a way of representing dependence information. Intuitively, a path of a term gives the set of variables of that term which are required to evaluate it.

The possible presence of cyclic dependencies in the program makes the task of propagating dependence information difficult. Instead of handling cyclic dependencies in the analysis phase, where dependence information is propagated, AB analysis performs an initial step which divides the function block into multiple acyclic blocks. Essentially, the cycles inside the block are moved outside, and then the cycles are completed using input and output variables. This step matches the graph-coloring step of coloring analysis. Since AB analysis breaks threads at function call boundaries, unlike coloring analysis, it propagates dependence information across the function call boundaries to produce better partitioning. It groups together terms demanded by the same set of outputs into partitions. It also attempts to group terms if doing so does not create any extra dependencies between inputs and outputs of the function.

4.5 Discussion

DD and AB partitioning techniques on one hand, and coloring analysis on the other, represent very different approaches developed for different programming language paradigms. In order to accurately evaluate these

[5] Strictness analysis identifies at compile time those arguments of a function which are needed for all possible computations.

[6] The "paths" of strictness analysis are unrelated to the "paths" of mode analysis. This confusing terminology is unfortunate but standard.

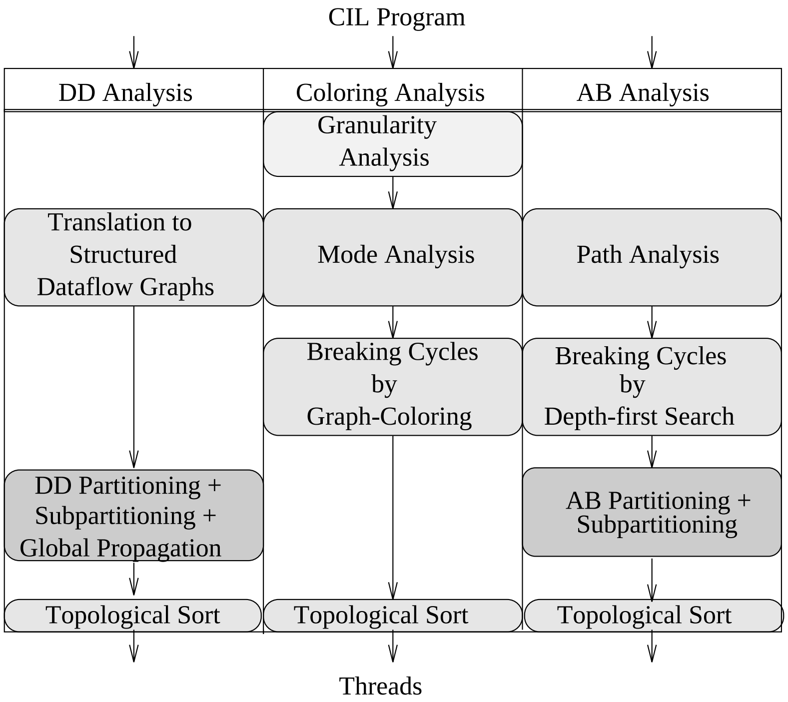

Figure 5: A Comparison of the Different Stages of DD, Coloring, and AB Analyses

approaches it is necessary to understand the issues they address. These issues include the number of threads produced for a given program, prevention of deadlock, simplicity, practicality, and the ability to tune the thread size to the underlying implementation.

The translation of programs from various language families into the CIL provided us with the same starting point for all the techniques discussed, facilitating a fair comparison of the three techniques. Unfortunately, space considerations preclude our exhibiting and explaining the translations. All three partitioning techniques share the same basic analysis steps: collecting the data dependencies between the terms of a function, grouping the terms into partitions, and then performing topological sort on the partitions to obtain threads.

The different stages in the three partitioning techniques are shown in Figure 5. Unlike DD and AB analyses, coloring analysis attempts to control the granularity of the threads (Section 4.3). DD analysis infer the data dependencies in a program through dataflow analysis. Coloring analysis can use dependence information derived from a variety of sources, such as mode analysis. AB analysis uses path analysis to derive dependence information. DD analysis breaks threads at all points which could be involved in a cyclic dependency. Coloring and AB analyses handle cyclic dependencies explicitly. Once the cycles have been broken, coloring analysis proceeds to produce liberal threads. However, DD and AB analyses, which produce conservative threads, attempt to improve the thread size by propagating dependence information across function call boundaries which break threads. Once partitions are produced, threads are obtained in all cases by topologically sorting the partitions.

Different definitions of “thread” used in the analysis lead to a large difference in the number of threads produced for the same program. In the coloring analysis, a liberal thread can include more than one function, can have an arbitrary number of synchronizing and long latency accesses (which may lead to suspensions), and can even contain references to other threads. In DD and AB analyses, a thread is a collection of instructions which does not contain procedure calls or synchronizing or long latency accesses — synchronization occurs only at the beginning of the thread. Thus, in DD and AB analyses, the requirement that split-phase accesses be kept in separate threads results in extra threads, and function calls split threads as well. Another important reason for differences in the number of threads is that in coloring analysis terms which are independent of each other are kept in the same thread in arbitrary order. DD and AB analyses will keep them in separate threads. It is likely, although not certain, that the reduction in the number of threads produced by coloring analysis in this way will be a useful one.

DD analysis is based on a worst-case scenario. Inclusion of more than one function call in a thread may cause deadlock, if the order of execution of the functions is determinable only dynamically. To prevent introduction of deadlock, DD analysis breaks threads at all function call boundaries. Similarly, inclusion of synchronizing accesses in threads may cause deadlock, if each of two threads blocks waiting for bindings which can only be provided by the other. In order to prevent any such possibility, DD analysis splits threads across all synchronizing accesses. In real programs, not all functions require their scheduling order to be determined at run time, not every synchronizing access results in deadlock, and not every conditional produces cyclic dependencies. Instead of splitting threads only when failing to split them might compromise the termination characteristics of the program, DD analysis applies the same strategy under both safe and unsafe situations. This keeps the analysis technique simple, but results in small thread size.

AB analysis attempts improve DD analysis while still retaining its simplicity. The propagation of dependence information across recursive calls is improved by use of path analysis, which identifies the presence of strict arguments in recursive calls using fixpoint iteration, and uses this information to improve partitioning. To keep the partitioning technique simple, DD analysis breaks threads at all function call boundaries and synchronizing accesses before applying the partitioning algorithm. AB analysis breaks threads only when cyclic dependencies are present. Though it breaks threads at function call boundaries and synchronizing accesses anyway during the main partitioning step, the delay facilitates better propagation of dependence information. This also allows the production of liberal threads by minimal modifications to the AB algorithm. AB analysis, like DD analysis, does not include mechanisms for controlling the granularity of threads.

Coloring analysis uses explicit safety mechanisms to prevent deadlock, by identifying terms which must be kept in separate threads. In an effort to conservatively derive safe information, the abstraction techniques used lose information, potentially resulting in less granularity than might theoretically be exploitable. The granularity analysis technique employed in coloring analysis to control the size of the threads (Section 4) is very imprecise. As King and Soper [16] themselves mention, a large collection of techniques are needed to analyze a small class of programs: thus, the difficulties of implementation in a compiler may outweigh the benefits achieved. The inclusion of multiple functions and synchronizing and long latency accesses in a thread does result in increased thread size. It can also result in inefficiency. Thread state has to be stored and context switching has to be performed on every suspension. If suspension is frequent, this overhead places a burden on the underlying implementation Coloring analysis provides no control over the number of suspension points in a thread, and thus cannot control it.

To summarize our results: DD analysis is simple and practical, avoiding safety checks by working under worst-case assumptions in all cases. The resulting thread size is small, especially given the global nature of the analysis. AB analysis offers improvements over DD analysis in the handling of cyclic dependencies and recursion, while keeping much of its simplicity. In contrast, coloring analysis obtains larger threads, and has a granularity control mechanism, but is complicated, and is not a good fit with some implementations.

5 Conclusions and Future Work

Non-strict functional and concurrent logic languages seem to be strongly similar, although this has not been widely recognized by researchers in these communities. Thus, solutions to problems in one domain may well transfer to the other. In particular, the static analysis problems encountered during compilation of these languages seem to be quite general: comparable solutions to these problems have been adopted independently in the two language communities.

We thus suggest that translation of programs from both language families into a common intermediate language has a number of attractive features. In particular, we note that 1) A clear description of CIL semantics will permit easy definition of the semantics of traditional languages in terms of the CIL; 2) optimizations performed on the CIL form will provide compilation improvements for both language families, and 3) comparison between static analysis algorithms for CIL programs is much easier than comparisons between algorithms for static algorithms of programs written in a variety of different languages.

We propose a particular CIL, Kernel$_2$, as a candidate. We note that its semantics are easy to define via term rewriting, and that it seems to contain the important features necessary for the straightforward translation of non-strict functional and concurrent logic languages. While this language seems useful for illustrative purposes, one important area for future work is in obtaining a final CIL design which satisfies all of our criteria, yet is simple to understand, analyze, and implement.

By way of evaluation of our technique, and because it is important in its own right, we consider a particular static analysis example, that of partitioning or threading concurrent programs. A good partitioning strategy has to address several sometimes conflicting goals [14]. The thread length has to be maximized in order to decrease thread switches, decrease explicit synchronization, and increase pipeline utilization and locality. This however should not be at the expense of exploitable parallelism, as the objective of partitioning is to create threads containing little or no easily exploitable parallelism. It is also necessary to identify portions of a program whose execution order is dynamically determined, so that they are not statically scheduled in a single thread. The thread size should be flexible: the compiler should be able to tune the size to the underlying hardware for efficiency.

We discuss three important thread-building algorithms. Our comparison was made a great deal easier by the use of the CIL. We note some of the relative advantages and disadvantages of these techniques, and discuss their domains of applicability. The comparison emphasizes that different thread definitions, motivated by different needs, lead to different (though comparable) static analysis performance under various metrics.

Future work in the development of partitioning algorithms will have to address the issue of obtaining optimal-sized threads based on a model that accounts for local *vs.* non-local execution and communication for a given architecture. The resulting threads can then be efficiently executed on architectures designed to execute threaded programs as well as on standard multiprocessors.

Acknowledgements

E. Tick was supported by an NSF Presidential Young Investigator award, with matching funds from Sequent Computer Systems Inc. M. Sami and B. Massey were supported by a grant from the Institute of New Generation Computer Technology (ICOT).

References

[1] Z. M. Ariola. *An Algebraic Approach to the Compilation and Operational Semantics of Functional Languages with I-Structures.* PhD thesis, Harvard University, June 1992.

[2] Z. M. Ariola and Arvind. P-TAC: A Parallel Intermediate Language. In *Conference on Functional Programming Languages and Computer Architecture*, pages 230–242. London, ACM Press, September 1989.

[3] Z. M. Ariola and Arvind. A Syntactic Approach to Program Transformation. In *ACM SIGPLAN Symposium on Parallel Evaluation and Semantics Based Program Manipulation*, New Haven, June 1991. Yale University, ACM Press.

[4] Z. M. Ariola and Arvind. Graph Rewriting Systems For Efficient Compilation. In *Term Graph Rewriting: Theory and Practice.* John Wiley and Sons, 1993.

[5] Arvind, L. Augusston, J. Hicks, R. S. Nikhil, S. Peyton-Jones, J. Stoy, and W. Williams. pH: a Parallel Haskell. Technical report, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, September 1993.

[6] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. In *Workshop on Graph Reduction*, number 279 in Lecture Notes in Computer Science, pages 336–369, Santa Fe, September/October 1986. Springer-Verlag.

[7] P. S. Barth, R. S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-Strict, Functional Language with State. In *Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, pages 538–568. Springer-Verlag, 1991.

[8] K. L. Clark and S. Gregory. PARLOG: Parallel Programming in Logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, January 1986.

[9] S. Coorg. Partitioning Non-strict Functional Languages, 1994. MIT M.S. Thesis. Unpublished.

[10] S. K. Debray, N.-W. Lin, and M. V. Hermenegildo. Task Granularity Analysis in Logic Programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–188, White Plains, NY, June 1990. ACM Press.

[11] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[12] J. E. Hoch, D. M. Davenport, V. G. Grafe, and K. M. Steele. Compile-Time Partitioning of a Non-Strict Language into Sequential Threads. In *Symposium on Parallel and Distributed Processing*, pages 180–189. Dallas, IEEE Computer Society Press, December 1991.

[13] P. Hudak, S. Peyton-Jones, and P. Wadler. Report on Programming Language Haskell: a Non-strict, Purely Functional Language, Version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[14] R. A. Iannucci. Toward a Dataflow/von Neumann Hybrid Architecture. In *International Symposium on Computer Architecture*, pages 131–140. Honolulu, May 1988.

[15] R. A. Iannucci. *Parallel Machine Languages.* Kluwer Academic Publishers, Boston, 1990.

[16] A. King and P. Soper. Heuristics, Thresholding and a New Technique for Controlling the Granularity of Concurrent Logic Programs. Technical Report CSTR 92–08, Department of Electronics and Computer Science, University of Southampton, 1992.

[17] A. King and P. Soper. Schedule Analysis: A Full Theory, A Pilot Implementation, And A Preliminary Assessment. Technical Report CSTR 92–06, Department of Electronics and Computer Science, University of Southampton, February 1992.

[18] A. King and P. Soper. Schedule Analysis of Concurrent Logic Programs. In *Joint International Conference and Symposium on Logic Programming*, pages 478–492. Washington D.C., MIT Press, November 1992.

[19] G. Lindstrom. Functional Programming and the Logic Variable. In *SIGPLAN Symposium on Principles of Programming Languages*, pages 266–280. New Orleans, ACM Press, 1985.

[20] B. C. Massey. Sequentialization of Parallel Logic Programs with Mode Analysis. Master's thesis, University of Oregon, September 1992. Also available as Technical report CIS-TR-92-18.

[21] R. S. Nikhil. Id (Version 90.0) Reference Manual. Technical Report CSG Memo 284-a, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, July 1990.

[22] R. S. Nikhil. A Multithreaded Implementation of Id Using P-RISC Graphs. In *Languages and Compilers for Parallel Computing*, 1993.

[23] R. S. Nikhil, Arvind, and J. Hicks. pH Language Proposal (Preliminary) — Draft. Technical report, MIT Laboratory for Computer Science, September 1993.

[24] E. Y. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[25] R. Sundararajan. *Data Flow and Control Flow Analysis of Logic Programs.* PhD thesis, Department of Computer Science, University of Oregon, 1994.

[26] Z. Symogyi. A System of Precise Modes for Logic Programs. In *International Conference on Logic Programming*, pages 769–787. University of Melbourne, MIT Press, May 1987.

[27] E. Tick. The Deevolution of Concurrent Logic Programming Languages. *Journal of Logic Programming*, 1994. Invited for Tenth Anniversary Issue. Also available as University of Oregon Technical Report CIS-TR-94-07.

[28] E. Tick, B. C. Massey, F. Rakoczi, and P. Tulayathun. Concurrent Logic Programs *a la Mode*. In E. Tick and G. Succi, editors, *Implementations of Logic Programming Systems*. Kluwer Academic Publishers, 1994.

[29] E. Tick and X. Zhong. A Compile-Time Granularity Analysis Algorithm and its Performance Evaluation. *New Generation Computing*, 11(3–4):271–295, June 1993.

[30] K. R. Traub, D. E. Culler, and K. E. Schauser. Global Analysis for Partitioning Non-Strict Programs into Sequential Threads. In *Conference on Lisp and Functional Programming*, pages 324–334. San Francisco, ACM Press, 1992.

[31] K. Ueda and M. Morita. Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, May 1994.

[32] P. Wadler. Strictness Analysis on Non-Flat Domains (by Abstract Interpretation Over Finite Domains). In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 181–198. Ellis Horwood Ltd, Chichester, 1987.