An Improved Compile-Time Memory-Reuse Scheme for Flat Concurrent Logic Programs

R. Sundararajan, A. V. S. Sastry, E. Tick
Dept. of Computer Science
University of Oregon

CIS-TR-95-04
February 1995

Abstract

The single-assignment property of concurrent logic programming languages results in a large memory bandwidth requirement and low spatial locality in heap-based implementations. As large amounts of garbage are generated, it becomes necessary to salvage used memory frequently and efficiently, with a garbage collector. Another approach is to detect when a data structure becomes garbage and reuse it. In concurrent languages it is particularly difficult to determine when a data structure is garbage and suitable for destructive update. Dynamic schemes, such as reference counting, incur space and time overheads that can be unacceptable. In contrast, static-analysis techniques can be used to identify data objects whose storage may be reused. Information from static analysis can be used by the compiler in generating appropriate instructions for reuse, incurring little or no runtime overhead. In this paper we present a new method of reuse detection based on abstract interpretation. We present empirical performance measurements comparing the new scheme against binary reference counting (MRB). It is shown that our proposed static analysis can achieve most of the benefits of MRB and improve the execution time.

This report is an extended and corrected version of a paper appearing in the *Joint International Conference and Symposium on Logic Programming*, Washington D.C., November 1992.

Contents

1 Introduction

Logic and functional programming languages are examples of languages utilizing the *single-assignment property* of variables, i.e., a variable can be bound to a value, at most once. In logic programming languages, a logical variable starts its life as an undefined cell and may later hold a constant, a pointer to a structure, or a pointer to another variable. These programming languages do not allow in-place update of data structures. Abstractly, the effect of an update can be achieved by creating a new copy of the structure, with some new portion inserted into the copy.

The single-assignment property is elegant because it is possible to use the availability of data as a means of process synchronization, similar to data flow computation. However, this property has the undesirable effect of resulting in large memory turnover, due to excessive copying. Copying is wasteful in terms of both execution time and storage requirement. The lack of economy in memory usage results in prodigious memory requirements by programs that update aggregate data structures. Garbage collection needs to be invoked frequently as the heap space is limited. Large memory bandwidth requirements and poor cache utilization hinder construction of scalable and high-performance architectures for parallel logic languages.

Research in the area of runtime garbage collection in functional languages may be carried over to concurrent logic languages. Cohen [11] surveys a range of traditional garbage collection techniques, classified as *mark and sweep* algorithms. Reference counting [26], efficient garbage collection algorithms such as Baker's algorithm [2, 21], and incremental garbage collection schemes [13], may alleviate the problem to some extent by making the garbage collection operation cheaper to perform, but do not reduce the memory requirements. Other approaches, such as maintaining multi-version structures [14] have also been proposed to avoid excessive copying.

If it is known that there are no references to a data object (other than the process inspecting the object!), then the structure can be reclaimed and used in building other structures if necessary. The detection and reuse of such data objects can be done either at compile-time through static analysis, known as *compile-time garbage collection*, or at runtime with additional data structures and instructions or by a combination of both.

Runtime techniques include general reference counting and approximations to general reference counting, known as binary reference counting. The MRB [7] scheme is an example of binary reference counting. Much work has been done in compile-time detection and reuse of structures in functional programs [19]. However, these results do not carry over to logic programming because of the complexity of unification and the presence of the logical variable. Analysis techniques for sequential Prolog (e.g., Mulkers [30] and Bruynooghe [5]) do not extend easily to concurrent logic languages, where no assumptions can be made on the order of execution of the goals or about the interleaving of their execution. On the one hand, our approach is simpler than that for Prolog because committed-choice programs do not backtrack and therefore do not require trailing. However, the analysis is more complex because we cannot reason about when some goal will start or finish executing, and the related problem of concurrent interleaving.

In this paper, we present a new analysis technique based on abstract interpretation for compile-time garbage collection and compare it to other proposals. We restrict ourselves to committed-choice logic programming languages [36]. The purpose of this paper is not to provide a general framework for abstract interpretation of concurrent logic languages but rather derive a simple scheme for identifying instances of local reuse. The key contributions of this research are 1) a formal specification of a depth-one abstract domain for threadedness information,[1] and 2) empirical measurements showing the utility of this domain. This paper will benefit compiler writers and runtime system implementors by providing a clear, easy-to-understand specification of the abstract interpretation analysis and its expected performance.

The paper is organized as follows. In Section 2, we review the alternative reuse analysis proposals. In Section 3 we describe our scheme to identify structures that may potentially be reused with very little runtime overhead. Specifically, we identify some structures occurring as input arguments as potential candidates for reuse while constructing new structures in the clause. The runtime test for ensuring that these structures may actually be reused is simple. Section 5 reviews alternative instruction sets devised for exploiting reuse information. We present experimental results showing that our method reduces memory usage, and improves execution speed when compared to the MRB method. The conclusions of this study and a summary of future work are given in Section 6.

2 Motivation and Literature Review

We start with a brief introduction to committed-choice logic programs before discussing garbage collection methods. A committed-choice logic program is a set of guarded Horn clauses of the form: "H :- $G_1, \ldots, G_m \mid B_1, \ldots, B_n$" where $m \geq 0$ and $n \geq 0$. H is the clause head, G_i is a guard goal, and B_i is a body goal. The commit operator '|' divides the clause into a passive part (the *guard*) and active part (the *body*). When the guard is empty, the commit operator is omitted. "Flat" committed-choice languages have a further restriction that guard goals are simple builtin functions, such as $=, \leq, \geq, \neq$. In committed-choice languages such as FCP(:) [36, 42], a guarded Horn clause has the form "H :- $Ask_1, \ldots, Ask_m$: $Tell_1, \ldots, Tell_n \mid B_1, \ldots, B_p$." The guards are divided into two parts *Ask* and *Tell*, separated by a colon. The *Ask* part may contain any builtin predicates and the *Tell* part may contain only unification equations.

We say that a goal a commits to a clause i, if a successfully *matches* with the head of clause i (i.e., without causing any bindings to the variables of the goal) and the guards of clause i (the *ask* goals) succeed without binding any goal variable and the *tell* goals also succeed. When a goal can commit to more than one clause in a procedure, it commits to one of them non-deterministically (the others candidates are thrown away). Structures appearing in the head and guard of a clause cause *suspension* of execution if the corresponding argument of the goal is not sufficiently instantiated. For example, in order for a goal *foo(X)* to commit to the clause "*foo(X)* :- $X = [A \mid B]$: *true* | *bar(A,B)*," the argument X of the goal must already be bound to a list structure, whose head (car) and tail (cdr) may be any term, even unbound variables. The *Tell* part of the guard is empty in this case. In the rest

[1]This article corrects and extends an earlier paper [39] which introduced the theory.

```
append(X, In, Out) :- X = []    : Out = In    | true.
append(X, In, Out) :- X = [H|T] : Out = [H|Z] | append(T,In,Z).


      try_me_else         bot          ; set up continuation
      wait_list           R1           ; [H|T]
      read_car_variable   R1,R4        ; R4 = H
      read_cdr_variable   R1,R5        ; R5 = T
      put_list            R1           ; R1 = [
      write_car_value     R1,R4        ;       H
      write_cdr_variable  R1,R2        ;        |Z]
      get_list            R3,R1        ; Out = R1
      put_value           R1,R5        ; append(T,
                                       ;          In,
      put_value           R3,R2        ;             Z)
      proceed
bot:  suspend
```

Figure 1: List Concatenation Source and Compiled Programs

of the paper, structures appearing in the head or in the ask part of the guard are referred to as *incoming structures*.

A suspended invocation may be *resumed* later when the variable associated with the suspended invocation becomes sufficiently instantiated. A program successfully terminates when, starting from an initial user *query* (a conjunct of atoms), after some number of reduction steps, no goals remain to be executed, nor are suspended.

To motivate the analysis proposed in this article, we present an instance of local reuse, at the machine level. Consider an FCP(:) program that concatenates two lists, shown in Figure 1. The abstract machine code generated for the second clause of `append/3` in also listed in the figure.[2] The precise semantics of this code is unimportant. Critically, instruction `put_list` allocates a new list cell on the heap for every call to `append`. The first argument of the second clause is a list structure that can be inferred (by the semantics of committed-choice logic programming languages) to be occurring as an input argument. This clause also constructs a list structure in its tell part. If there is no other reference to the input list cell, and the components of the list cell are non-variables, then the cell can be reused while constructing the list `Out`. If local reuse were exploited by the `put_list` instruction, no memory would be wastefully allocated by the procedure.

Concurrent logic programs utilize *single producer/single consumer* communication quite extensively. The communication, in its simplest form, is performed with a shared variable. One process writes to the variable, and the other process reads the value. Since there is only one reader, the reader, after reading the value, may reuse the memory cells used for

[2]The instruction set is for the FGHC abstract machine developed by Kimura and Chikayama [23], similar in some respects to Warren's Abstract Machine (WAM) for Prolog [1].

storing the value of the variable. Reuse of data structures can have considerable impact because memory cells are allocated less frequently and the garbage collector is invoked less often. Furthermore, memory reuse improves spatial locality of memory references, thereby improving cache performance. Techniques to implement reuse, relevant to logic programs, are discussed below.

2.1 Functional Language Research

One of the major implementation issues in functional languages is the efficient implementation of the update operator on array data structures. The straightforward implementation would take linear time in the size of the array, as opposed to constant time update in imperative languages. Through static analysis of liveness of the aggregates, the update operations can be optimized. The related research done in this area are detecting single-threadedness of the store argument of the standard semantics of imperative languages [35], Hudak's work on abstraction of reference count for a call-by-value language with a fixed order of evaluation [18], and update analysis for a first-order lazy functional language with flat aggregates using a non-standard semantics called path semantics [3]. All the above analyses are for sequential implementations.

Another important implementation area is shape analysis, the static derivation of data structure composition. Recent work by Chase *et al.* [6] describes a more accurate and efficient analysis technique than previous methods. They also describe how this storage shape graph (SSG) method can be followed by reference-count analysis [18]. Recently there has also been work on reordering the expressions in a strict functional language with the objective of making most of the updates destructive [33].

The analysis of single-threadedness or storage reusability in logic programming languages is significantly different from that of functional languages because of the power of unification and logical variables available in the former. We do not address the issue of shape analysis in this paper, only reference counting. One important use of our analysis is the reuse of "local" structures that do not require shape analysis to uncover.

2.2 General Reference Counting

One approach is to associate with each variable, a count of the number of references to that variable. The count is updated whenever new references to a variable are created or old ones discarded. If a *reader process* detects, upon reading, that the count is one (i.e., that there are no references to the variable other than its own), then the variable can be updated in place, or the variable's space can be reused. This method is known as *general reference counting.* The main disadvantages of this scheme are twofold. Firstly, memory requirements are doubled for each variable, due to the count field. Secondly, managing the count field to keep track of references may cause considerable runtime overhead. A garbage collector based on general reference counting is described in [13]. This scheme removes the need to store the reference count alongside the variable, by using tables to store addresses of variables and the number of references to each variable. The tables are stored in memory and need to be updated in the usual fashion. The tables reduce memory requirements, but still constitute a bottleneck in achieving a high-performance implementation.

```
foo( X ) :- X = [ A | B ] : true | bar( A, B ).

foo:    try_me_else           bot      ; set up continuation
        wait_list             R1       ; [ A | B ]
        read_car_variable     R1,R2    ; R2 = A
        read_cdr_variable     R1,R3    ; R3 = B
        collect_list          R1       ; collect list cell
        put_value             R1,R2    ; set up call to bar( A, B )
        put_value             R2,R3
        execute               R2,bar
```

Figure 2: List-Cell Reuse: Source and Compiled Programs

2.3 Binary Reference Counting

A binary reference count is a one bit tag associated with a data object that indicates if there are one or more references to that object. The method is an approximation of general reference counting in the sense that the condition when a multiply referenced data structure becomes singly referenced, cannot be detected. A prime example of this approach is the Multiple Reference Bit (MRB) garbage collection scheme [7]. One bit in each pointer, referred to as the MRB, is *set* if the referenced object has multiple references active, or *reset* if there is only one active reference. Advantages of the MRB are ease of hardware implementation and reasonable execution speed [22]. Several MRB-based optimizations are described by Inamura *et al.* [22]. In this scheme, memory is incrementally reclaimed with special `collect` instructions that are generated for each incoming structure. We illustrate this technique with a sample clause and its abstract machine code, shown in Figure 2. Instruction `collect_list` attempts to reclaim the list cell. The attempt succeeds if the MRB is *off*, in which case the cell is added to a free list. Separate free lists are maintained for structures of varying sizes. When a structure is created, instead of freshly allocating it from the heap, it is allocated from a free list. If the free lists are used in a LIFO (last in, first out) manner, it is likely that the reclaimed cell is still in the cache.

As mentioned above, `collect` instructions are generated for each clause-head structure. The MRB method *per se* does not involve compile-time determination of when `collect` instructions are needed, and when they are not. The `collect` instructions will succeed in reclamation if the structure is singly referenced, and fail to reclaim otherwise. All structures, regardless of their potential for reuse, will incur the overhead of `collect` instructions.

2.4 Compile-Time Analysis

The application of static program analysis to infer properties of programs, and the use of this information to generate specialized and efficient code, have proved to be quite successful in logic languages. Several static analyses of logic programs to infer groundness, and sharing information, structure sharing and liveness (among others) have been proposed for sequential Prolog (e.g., [5, 12, 29, 30, 31, 38]). The main differences between analysis of concurrent logic languages and pure logic programs are due to (i) synchronization entailed by goal–head matching and *Ask* unifications and (ii) interleaved execution of body goals.

In our proposed analysis, we consider synchronization to some extent. Since ask unifications cannot not bind goal variables, they are approximated differently than tell unifications which may bind goal variables. On the other hand, the synchronization implied by ask unifications may preclude some execution orderings of body goals. We assume that body goals may be executed in any order and hence our analysis is conservative.

The second difference is due to the interleaved execution of body goals in a concurrent logic program. Our analysis safely approximates all possible interleavings of body goals by a local computation which propagates the effect of reducing a body goal to all other body goals until a fixed-point is reached. This local fixed-point computation is an extra step in the analysis of concurrent logic programs as compared to parallel or sequential logic programs.

The only other work in applying static analysis techniques to detect possibility of reuse in concurrent logic languages, other than the research described in this paper, is by Foster and Winsborough [17]. Foster and Winsborough's paper [17] deals mainly with the ways to use information about single-threaded structures. They provide only a sketch of a collecting semantics for Strand programs [15] in which a program state is associated with a record of the program components that operated on it. The analysis details are in an unpublished, unfinished draft [16], and hence it is premature to compare their scheme with ours. In a recent private communication [41], one of the authors agrees that our approach is significantly different from theirs and the fact that their paper is unfinished makes it impossible for us to provide a direct comparison.

3 Overview of Proposed Static Analysis

The reference counting schemes previously reviewed have the main deficiency of excess runtime overheads. We are not aware of any successful (efficient) implementation of general reference counting for a parallel language. Binary reference counting, for instance MRB, adds runtime overheads to the abstract machine instruction set in which it is implemented. In this section, we propose a static analysis method based on an abstraction of transition system semantics to detect threadedness in flat committed-choice logic programs and use this information to generate reuse instructions. Suspension analysis of concurrent logic programs due to Codish *et al.* [8] is also based on an abstraction of transition system semantics.

There are four distinct ways in which a variable can be used for sharing information in concurrent logic programs. They are: Single producer–Single consumer (*SS*), Single producer–Multiple consumer (*SM*), Multiple producers–Single consumer (*MS*), and Multiple producers–Multiple consumer (*MM*). Since a variable may be bound at most once in logic languages, the notion of multiple producers implies that there are several potential producers but only one succeeds in *write-mode* unification. In a successful committed-choice program, all other potential producers perform *read-mode* unification. Ueda and Morita [40] define the class of *moded* FGHC programs to be those in which there are no competing producers. In legal moded FGHC programs, *MS* and *MM* variables do not exist. Strand is a similar language, with tell unification replaced by assignment only [15]. Saraswat [32] proposed a related language, Janus, which allows only S variables, each appearing only twice: as an "asker" and "teller," explicitly annotated by the programmer.

The purpose of our analysis is to determine which type of communication, S or M,[3] applies to each of the program variables. This information is used by the compiler to generate reuse instructions (see Section 5.1). The algorithm is safe for non-moded programs, but little reuse will be detected in programs where multiple producers and consumers abound. Some form of garbage collection will still be needed for programs with multiple producers and multiple consumers. We do not advocate our approach as an alternative to garbage collection but rather as supplementing garbage collection by identifying single-threaded structures at compile-time and reusing them. As our experiments show, in addition to reclaiming memory as efficiently as the MRB dynamic scheme, our method results in savings in execution time as well for programs with functional predicates. Since most (not all) programming paradigms can be implemented in moded programs, we expect accurate information to be produced from our simplified analysis, for a large class of programs.

Structures appearing in the head and ask part of a clause denote *incoming* data, since computation will suspend until the input arguments are sufficiently instantiated.[4] Thus if such an incoming structure is determined to be reusable, an attempt could be made to use its storage when constructing a structure in the body. Consider the following clause:

$$p(X,S,Y) :- S = t(L,C,R),\ X < C\ :\ Y = t(N,C,R) \mid p(X,L,N).$$

If the second argument in the head, S (which must be a structure $t(L,C,R)$), may be reused, it would be best to reuse it when constructing the structure $t(N,C,R)$. This is known as *instant* or *local* reuse. However, if no immediate use existed in the clause, the reclaimed storage could be stored away for future use (say, added to a free list). This is known as *deferred reuse.* Maintaining ordered free lists based on structure size, and using them in a LIFO manner, can result in more efficient program execution. Deferred reuse is equivalent to the `collect` operations defined in the context of MRB.

In the following presentation, data objects that have a single producer and single consumer are referred to as *single-threaded* and all other data objects are referred to as *multiple threaded.* We assume structure-copying implementations. Our analysis detects single-threaded structures at compile-time. These structures can be reused at runtime if the top-level components are nonvariables. The presence of uninstantiated variable(s) in the top-level of a structure renders the structure unsuitable for reuse (when variables are allocated inside structures) even if the structure is single-threaded. The reason is that a producer of the unbound variable may bind its value *after* the enclosing structure has been reused!

[3]We collapse *SM*, *MS* and *MM* into M in the analysis.

[4]A more formal explanation of semantics of flat concurrent logic programming languages can be found in Section 4.1.

We also assume that *structure sharing analysis* has been done, and that the results of the analysis are available. We envision structure sharing analysis similar to that proposed by King [24, 25], a research topic orthogonal to that of this paper. If sharing information is not available, then we can make worst-case assumptions about sharing and perform the analysis. This may produce fewer useful results.

3.1 Multiple Threadedness of Structures and Components

Compile-time detection of single-threaded data structures necessarily involves some representation issues and we now discuss these issues relevant to propagating threadedness information safely and precisely. Representation of compound structures has a direct bearing on how the threadedness of a structure affects the threadedness of its components (and vice versa) and raises the following three questions.

- Is a substructure of a multiple-threaded structure multiple threaded?
- Does a structure always become multiple threaded if one of its substructures is multiple threaded?
- How does the threadedness of a subterm of a structure affect another subterm of the same structure?

If a structure is multiple threaded, it means that there are (potentially) several consumers accessing the structure. Each consumer may access any substructure, implying that each substructure may also have multiple consumers. Thus multiple threadedness of a structure implies the multiple threadedness of its components.

Multiple threadedness of a component of a structure, however, does *not always* mean that the structure becomes multiple threaded. Suppose a structure is built in the body of a clause and it contains a head variable which is multiple threaded. A head variable is simply a reference to an incoming argument which has already been created. Only a *pointer* to that actual parameter resides in the structure built in the body. Because the variable is not created inside the current structure, the reuse of the structure does not affect the contents of the multiple-threaded component. Therefore the structure does not become multiple threaded.

Now suppose a structure is built in the tell part of the guard and it contains at least one variable Z local to the clause (i.e., the variable Z does not appear in the head, or the ask part, or in the RHS of a tell equation of the form $X = f(\ldots)$ where X is a variable that appears in the head or the ask part) and that variable Z is multiple threaded. If the implementation allocates variables *inside* structures, then a reuse of the structure will reclaim the space allocated for the multiple-threaded variable and is therefore unsafe. In this case, we have to make the structure multiple threaded. If the implementation creates variables *outside* structures (the structure arguments are linked to the variables by pointers), then multiple threadedness of a component would *never* make the structure multiple threaded.

For the analysis presented in this paper, we assume that variables are *never* created within a structure. Our abstract domain exploits this by being expressive enough to represent that the top level of a structure may be single threaded and at the same time any of its substructures may be multiple threaded.

The answer to the third question depends on the sharing of the components of the structures. If two subterms of a structure share, then multiple threadedness of one may make the other multiple threaded.

4 Abstract Interpretation

In an abstract interpretation framework for a language, it is customary to define a core semantics for the language leaving certain domains and functions unspecified. These domains and functions are instantiated by an interpretation. A standard interpretation defines the standard semantics of the languages and an abstract interpretation abstracts some property of interest. The abstract and the standard interpretations are related by a pair of adjoint functions, known as the abstraction and concretization functions. We first provide an operational semantics for the language Flat Concurrent Prolog, FCP(:), and then define our abstract interpretation method for reuse analysis. The proposed technique is also applicable to Flat Guarded Horn Clause (FGHC), Strand, and similar languages [36].

4.1 Operational Semantics for FCP(:)

The following operational semantics is a minor variation of the standard transition system semantics for concurrent logic programs and is derived from [36]. The knowledgeable reader may wish to skip to the next sub-section.

A computation state is a tuple $\langle G, \theta \rangle$ consisting of a goal G (a sequence of atoms), a current substitution θ, and a renaming index i. The index is used in renaming the variables of a clause (*PVar* for program variables) apart from the variables of the goal. Function *rename* : $PVar \times \mathcal{N} \rightarrow Var$ subscripts the program variables with a renaming index[5] and $rename^{-1}$: $Var \rightarrow PVar$ removes the subscript. Function *rename* can be homomorphically extended to *rename* : $Clause \times \mathcal{N} \rightarrow Clause$. The initial state $\langle G, \varepsilon \rangle$ consists of the initial goal G, the empty substitution ε, and an initial renaming index.

Definition: Computation
A computation of a goal G with respect to a program P is a finite or infinite sequence of states $S_0, \ldots S_i, \ldots$ such that S_0 is the initial state and each $S_{i+1} \in t(S_i)$ where t is a transition function from S to $\mathcal{P}(S)$ (defined below). □

A state S is a terminal state when no transition rule is applicable to it. The state $\langle true, \theta \rangle$ is a terminal state that denotes successful computation and $\langle fail, \theta \rangle$ denotes finitely failed computation. If no transition is applicable to a state $S = \langle A_1, \ldots, A_n, \theta \rangle$ $(n \geq 1)$ where $A_j \neq fail$, $1 \leq j \leq n$, then the state is dead-locked. We define the meaning of a program P as the set of all computations of a goal G with respect to P.

Definition: Transition Rules

[5]Throughout this paper, we do not show renaming indices in the formalism, to simplify the presentation.

- $\langle A_1, \ldots, A_j, \ldots, A_n, \theta \rangle \xrightarrow{reduce} \langle (A_1, \ldots, A_{j-1}, A_{j+1}, B_1, \ldots, B_k)\theta', \ \theta \circ \theta' \rangle$ if $\exists$ a clause C s.t. *rename(C,i)* $= H$:- *Ask* : *Tell* $|$ $B_1, \ldots, B_k$ and *try*$(A_j\theta, H, Ask, Tell) = \theta'$.
- $\langle A_1, \ldots, A_j, \ldots, A_n, \theta \rangle \xrightarrow{fail} \langle fail, \theta \rangle$ if for some j, and for all (renamed) clauses H :- *Ask* : *Tell* $|$ $B_1, \ldots, B_k$, *try*$(A_j\theta, H, Ask, Tell) = fail$.

$\square$

Function *try* is defined in terms of *match* which tests if the selected atom from a goal matches the head of the selected clause without binding any of the goal variables.

Definition:

$match(A_j, H) =$

fail	if $mgu(A_j, H) = fail$
θ	if θ is the most general substitution s.t. $A_j = H\theta$
suspend	otherwise

$\square$

Definition:

$try(A_j, H, Ask, Tell) =$

$\theta \circ \theta'$	if $match(A_j, H) = \theta \wedge test(Ask\theta) = success \wedge mgu(Tell\theta) = \theta'$
fail	if $match(A_j, H) = fail \vee (match(A_j, H) = \theta \wedge test(Ask\theta) = fail) \vee$ $(match(A_j, H) = \theta \wedge test(Ask\theta) = success \wedge mgu(Tell\theta) = fail)$
suspend	otherwise

$\square$

The definition of *test*$(Ask\theta)$ can be found in Shaprio [36]. This sufficiently summarizes the operational semantics of flat concurrent logic languages to define our abstract interpretation scheme.

4.2 Syntactic Assumptions

Without loss of generality, the following syntactic constraints are placed on logic programs to facilitate the analysis. These constraints can be satisfied by simple transformations at compile-time. The rationale behind the constraints (especially 2 and 3) is to ensure that *Ask* and *Tell* equations are in a solved form [27]. These conditions facilitate reasoning about structures that are definitely created at commit time (see Section 4.6). Similar syntactic assumptions are fairly standard in the analysis of sequential logic programs.

1. Head arguments are distinct variables. This restriction simply moves (to the ask part of the guard) the matching of head arguments with the goal arguments.
2. Equations of the ask part may be of two forms:
   - $X = Y$, where the variables X and Y must also appear in the head.
   - $X = f(\ldots)$ where variable X must appear in the head.

The left-hand-side variables may occur exactly once in the ask part.[6]

3. Equations of the tell part may be of two forms:
   - $X = Y$, where the variables X and Y must also appear in the head or the ask part of the guard.[7]
   - $X = f(...)$

   The left-hand-side variables may occur exactly once in the tell part.

4. All program variables have been renamed such that no variable occurs in more than one clause. This is because, when we define the abstraction function, we will merge information about various incarnations (renamed versions) of the same variable.

5. Arguments of procedure calls are variables. A goal such as $p(X,\ f(Z))$ can be replaced by a pair of goals $p(X,\ Y)$ and $Y = f(Z)$ where Y is a new variable not occurring in the clause and the unification goal $Y = f(Z)$ can be moved into the tell part of the guard. Unification equations of the form $Y = f(...)$ may appear in the tell part but not in the body. The reason for this transformation is to make explicit the structure creation operation and to simplify the abstraction of head–goal matching.

4.3 Abstract Domain

In the standard semantics, computation is defined as a (finite or infinite) sequence of states where two successive states are related by the transition function. Hence, the standard domain of interpretation is $\mathcal{P}(\mathit{Computation})$. We are interested in determining the set of program variables that will be bound only to single-threaded data structures in *any* computation. In our abstract domain a variable can take values from the complete lattice $L + L^2 + ... + L^k$ where L is the two-point lattice whose least element is S (Single Producer/Single Consumer) and the top is M (Single Producer/Multiple Consumers). Integer $k \geq 1$ is the maximum arity of any functor in the program. Thus, our abstract domain *AbEnv* is a finite function of type $PVar \rightarrow \{S, M, \langle S, S\rangle, \langle S, M\rangle, \langle S, S, S\rangle, \langle S, S, M\rangle, \langle S, M, S\rangle, \langle S, M, M\rangle, ...\}$. Note that the partial ordering on *AbEnv* is the usual point-wise ordering and the least upper bound of a set R of *AbEnv* is defined as follows.

<u>*Definition*:</u> $\bigsqcup R \;=\; f \in AbEnv \text{ s.t. } \forall x \in PVar\ f(x) = \bigsqcup \{y \mid f' \in R \wedge y = f'(x)\}$ $\square$

Our abstraction function α will map a set of computations to *AbEnv*. Function α is defined in terms of two other functions, α' and α''. Function α' maps a state to *AbEnv* and α'' maps a computation (which is a sequence of states) to *AbEnv*. We need an auxiliary predicate *multi_occurs*(*Var*, *State*) which is true whenever the variable *Var* occurs *more than once* in state *State* (counting each occurrence of a variable in each atom in the state). Procedure *subterm*(i, X) returns the set of subterms bound to the i^{th} argument of X.

[6]Local variables are rather a nusiance in our analysis so we remove them from the ask guards with rewriting rules.

[7]This is enforced to simplify the exposition of the abstract unification algorithm.

Definition: $\alpha' : State \rightarrow AbEnv$

$$\begin{array}{lcll} \alpha'(R) & = & \{Z \mapsto M & \textit{if } Z \in \{\ rename^{-1}(X) \mid multi\_occurs(X,R)\ \} \\ & & \ Z \mapsto \langle S, T_1, ..., T_n \rangle & \textit{if } Z \in \{\ rename^{-1}(X) \mid \neg multi\_occurs(X,R) \wedge \\ & & & \quad T_i = \begin{cases} M & \exists\, Y \text{ s.t. } Y \in subterm(i, X\theta) \\ & \quad \wedge\ multi\_occurs(Y,R) \\ S & \text{otherwise} \end{cases} \\ & & & \quad \}\ \} \end{array}$$

□

From the above definition, a multiple threaded variable is denoted as M. A single threaded variable is denoted as $\langle S, T_1, ..., T_n \rangle$, for $n \geq 0$ and $T_i \in \{S, M\}$. T_i is the threadedness of the i^{th} component of any term bound to the variable, where n is the maximum arity of all such terms. Note that threadedness of subterms can only be S or M because this domain does not contain information deeper than the top level of the term. For simplification, if $n = 0$ or if $T_i = S$ for all i, then the value is denoted as S. For example, consider the following tell unification: $X_1 = t(X_2, X_3)$ where X_2 is S, X_3 is M, and X_1 is known to be single threaded. Then the threadedness of X_1 is denoted as $\langle S, S, M \rangle$. If X_1 is known to be multiple threaded then its threadedness is simply M.

A variable X (representing some data structure) is assumed to be multiple threaded if it appears more than once in the current state. This has an inherent inaccuracy, as illustrated in the following example. Consider the query " ?– p" for the clause:

$$p \text{ :- } \mathit{true}\ :\ X = f(a) \mid q(X),\ r(X).$$

Let the initial state be $S_0 = \langle \{p\}, \{\} \rangle$. The state resulting from reducing the goal p with respect to the above clause is:

$$S_1 = \langle \{q(f(a)), r(f(a))\},\ \theta = \{X \mapsto f(a)\}\ \rangle.$$

The subterms $f(a)$ of the literals q and r represent the current binding of the variable X shared by the literals q and r. Hence, in this case, X really is multiple threaded.

On the other hand, consider what happens if the goal p is reduced with respect to the following clause:

$$p \text{ :- } \mathit{true}\ :\ X = f(a), Y = f(a) \mid q(X),\ r(Y).$$

We obtain the state

$$S_2 = \langle \{q(f(a)), r(f(a))\},\ \theta = \{X \mapsto f(a), Y \mapsto f(a)\}\ \rangle.$$

It is clear that neither X nor Y are multiple threaded; however, the subterms $f(a)$ of the literals q and r represent the current binding of the variable X and appear more than once in the state. Thus X (and Y) would be labeled multiple threaded.

Essentially, there is not enough information in the states S_1 and S_2 to distinguish between them and hence we safely approximate X to be multiple threaded in both states. Ideally a differentiation should be made concerning shared and copied versions of terms: in S_1, $f(a)$ is shared, whereas in S_2, there are two distinct copies of the structure $f(a)$;

one is created because of the equation $X = f(a)$ and another because of the equation $Y = f(a)$. For the purposes of threadedness analysis, we can tag all occurrences of structures in a program with unique tags. This will remove the imprecision caused by treating two occurrences of the structure $f(a)$ (in the second example) as the same.

Different (renamed) versions of the same variable may occur in a state but we merge their threadedness. We consider a variable to be multiple threaded in a state if any one of its renamed versions is multiple threaded. It is straightforward to extend the definition of α' from *State* $\rightarrow$ *AbEnv* to *Computation* $\rightarrow$ *AbEnv*, since a computation is just a sequence of states. In the following definition, $S \in Comp$ denotes each state S in the computation sequence *Comp* (by a slight abuse of notation).

Definition: $\alpha'' : Computation \rightarrow AbEnv$

$$\alpha''(Comp) = \bigsqcup_{S \in Comp} \alpha'(S)$$

□

The abstraction function α and the concretization function γ can now be defined as follows.

Definition: $\alpha : \mathcal{P}(Computation) \rightarrow AbEnv$

$$\alpha(CompSet) = \bigsqcup_{C \in CompSet} \alpha''(C)$$

□

Definition: $\gamma : AbEnv \rightarrow \mathcal{P}(Computation)$

$$\gamma(X) = \{C \mid \alpha''(C) \sqsubseteq X\}$$

□

In the following sub-sections, we describe the abstract interpretation algorithm in detail. The algorithm consists of (i) abstract reduction which includes initialization of the abstract environment of a clause (Section 4.4), head–goal matching and guard execution (Section 4.6), (ii) local fixpoint computation and abstract success environment computation (Section 4.5), and (iii) global fixpoint computation in the standard fashion (e.g., [12, 4]). In the future, it may be interesting to rephrase our simple abstract domain in terms of Marriott and Sondergaard's *Prop* domain [28], in order to give a more intuitive declarative meaning of abstract operations.

4.4 Initialization

The initial abstraction of the threadedness of variables is based on the number of occurrences of a variable (and the variables it shares with) in the head and the body. The general rule is that all occurrences of the same variable in the head and the guards (both ask and tell) are counted as a single occurrence and each occurrence of a variable in the body is counted individually. Initially, variables are assigned threadedness independent of the terms they may be bound to by ask and tell unifications. Thus A secondary refinement due to these guards is performed during threadedness propagation (Section 4.5).

If a variable occurs two or fewer times, it is initialized to S. If a variable occurs more than twice, it is initialized to M implying that variables that occur only in the guard (*Ask* or *Tell* part) are initialized to S. The variables that occur only in the guard will inherit their threadedness from other structures with which they are matched/unified.

There is one exception to the previous rules. A variable occurring in the head, *Tell* guard, and body is initialized to M. This is necessary because the guard produces a value which is exported through the head and shared by a body goal. In most practical programs, the exported variable will be shared by another (external) goal, thus it is multiple threaded.

The following example illustrates the computation of an initial approximation for the body occurrences.

$$\begin{array}{lll} f(X_1, X_2, X_5, X_6) & :- & X_1 < X_2 \ : \ X_3 = t(X_1),\ X_5 = a,\ X_6 = 3 \ \mid \\ & & p(X_1, X_4),\ q(X_1, X_2, X_3, X_4),\ r(X_5). \end{array}$$

Variable X_1 is initialized to M since it occurs three times — once in the head/guard and twice in the body of the clause. Variable X_2 is initialized to S because it occurs once in the head/guard and once in the body. Similarly, X_3 and X_4 are S. X_5 occurs in the head, *Tell* guard, and body, so it is M. X_6 occurs only once (head and guard), so it is S. We refer to the initial environment of the clause, obtained with the above rules, as $AbEnv_{init} = \{X_1 \mapsto M,\ X_2 \mapsto S,\ X_3 \mapsto S,\ X_4 \mapsto S,\ X_5 \mapsto M,\ X_6 \mapsto S\}$.

We use $AbEnv^i_{init}$ to mean the initial environment of clause i. Function *init: clause* $\rightarrow$ *AbEnv* returns the initial abstract environment for a clause using the above rules. As an example of sharing, consider the following:

$$p(X_1, X_2, X_3) :- true \ : \ true \ \mid \ q(X_1, X_2),\ r(X_3).$$

Without considering sharing, the number of occurrences of X_2 and X_3 are each two. Suppose we are given information that X_2 and X_3 may share. Considering this sharing information, we count three occurrences of each. Thus we initialize each to M.

4.5 Threadedness Propagation

Propagation of information across procedure calls involves modeling the reduction of a goal into a set of goals by head matching, including the successful execution of the guards and the interleaved execution of body goals. The overall mechanism is summarized in Figure 3, which is described in the following sub-sections. Given a goal *Goal*, a caller's environment $AbEnv_{call}$, and a clause C we abstract the reduction process by first computing the initial environment $AbEnv_{init}$ of C, and then by safely approximating the effects of head-goal matching, *Ask* testing, and *Tell* unifications.

$$\begin{array}{llcll}
\multicolumn{5}{l}{\underline{\textit{Definition:}}\ \ AbsRed: Atom \times Clause \times AbEnv \rightarrow AbEnv} \\
\multicolumn{5}{l}{\overline{AbsRed(}Goal, C, AbEnv_{call}) =} \\
\textbf{let} & H :- A : T \mid B & = & rename(C) & (1) \\
 & AbEnv_{init} & = & init(H :- A : T \mid B) & (2) \\
 & E' & = & Match(Goal, H, AbEnv_{call} \cup AbEnv_{init}) & (3) \\
 & E'' & = & Punify(A, E') & (4) \\
 & E''' & = & Aunify(T, E'') & (5)
\end{array}$$

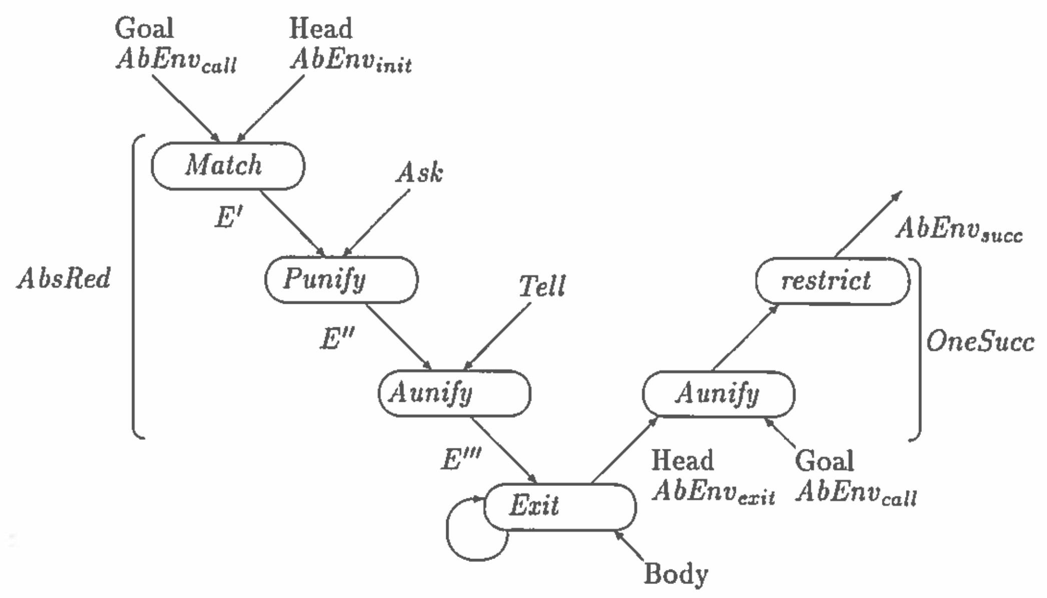

Figure 3: Abstract Interpretation Mechanism

in

$$rename^{-1}(restrict(E''', Vars(H, A, T, B))) \tag{6}$$

□

The variables of the clause C are consistently renamed (1) to avoid capturing the goal variables. The initial environment $AbEnv_{init}$ of the renamed clause is computed with the *init* function discussed in Section 4.4. *Match* approximates head–goal matching in the environment of the goal and the initial environment of the renamed clause (3). *Punify* abstracts the effect of *Ask* goals of the guard (4), and *Aunify* abstracts the effect of *Tell* unification goals (5). Functions *Match*, *Punify*, and *Aunify* are defined in Section 4.6. We restrict the resulting environment E''' to the variables of the renamed clause, and then apply the inverse of the renaming function (6). This gives us the abstract environment $AbEnv_{entry}$ for the variables of clause C on reducing goal g with respect to clause C.

The success environment of a user-defined goal *Goal* is obtained, in function *Succ*, by taking the lub (least upper bound) of success environments of all the matching clauses (3 below). Although the nondeterminism in committed choice languages is the *don't-care* type, at compile time we do not know which clause will commit, and hence take the lub. The program P (an implicit parameter to *Succ*) is analyzed by calculating $AbEnv_{succ}$ =

$Succ(Query, AbEnv^{query}_{init})$, where *Query* is the top-level procedure invocation and $AbEnv^{query}_{init}$ is the initial environment for the query variables.

```
Definition: $Succ: Atom \times AbEnv \rightarrow AbEnv$
$Succ(Goal, AbEnv_{call}) =$
let $\{p_1, p_2, \ldots, p_k\}$ be the clauses whose heads match $Goal$ and
      $\{b_1, b_2, \ldots, b_k\}$ be their respective bodies
    $AbEnv^i_{entry} = AbsRed(Goal, p_i, AbEnv_{call})$        (1)
    $AbEnv^i_{exit} = Exit(b_i, AbEnv^i_{entry})$        (2)
in
    $\bigsqcup_{i=1}^{k} \left\{ OneSucc(Goal, p_i, AbEnv_{call}, AbEnv^i_{exit}) \right\}$        (3)
```
□

The function *OneSucc* is similar to *AbsRed*, with two exceptions. First, we use the exit environment of a clause instead of its initial environment. Second, after simulating the head unifications, the result is restricted to the variables of the calling environment (4 below) and *not* to the variables of clause C whose head matched *Goal*.

```
Definition: $OneSucc: Atom \times Clause \times AbEnv \times AbEnv \rightarrow AbEnv$
$OneSucc(Goal, C, AbEnv_{call}, AbEnv_{exit}) =$
 let $(C', AbEnv_{exit'}) = rename((C, AbEnv_{exit}))$        (1)
     $H :- A : T \mid B = C'$        (2)
     $E = Aunify(\{Goal = H\}, AbEnv_{call} \cup AbEnv_{exit'})$        (3)
 in
     $restrict(E, Vars(AbEnv_{call}))$        (4)
```
□

In concurrent logic programs, body goals may execute in any order and their execution may also be interleaved. Codognet *et al.* [10], and Codish *et al.* [9, 8] describe cases for which it is sufficient to consider any one particular schedule of body goals during analysis, as opposed to all schedules. In our analysis, however, we *do* need to consider interleavings, as the following example shows.

```
       :-  p(X), q(X).
p(X)   :-  ... | r(X).
q(X)   :-  ... | a(X), b(X).
```

Consider two of many possible interleavings (each is a sequence of resolvents): $\{(p,q),(p,a,b),(p,b),(r,b),(b)\}$ and $\{(p,q),(r,q),(q),(a,b),(a)\}$. In the first schedule procedure a cannot reuse X because it executes concurrently with p and b. In the second schedule, procedure a can safely reuse X. Thus it is certainly safe to consider *all* schedules, which we do as follows.

We safely approximate this by iterating the computation of abstract exit environment (given the abstract entry environment) until the exit and the entry environments are the same. This function is performed in *Exit*, as follows.

```
Definition:  $Exit: Body \times AbEnv \rightarrow AbEnv$
$Exit(Body, AbEnv_{entry}) =$
 let  $AbEnv_{exit} = ExitIter(Body, AbEnv_{entry})$
 in   if    $(AbEnv_{entry} = AbEnv_{exit})$  then
            $AbEnv_{entry}$
      else
            $Exit(Body, AbEnv_{exit})$
                                                        □
```

```
Definition:  $ExitIter: Body \times AbEnv \rightarrow AbEnv$
$ExitIter(Body, AbEnv_0) =$
 if     $empty(Body)$  then
        $AbEnv_0$
 else   let $Body = \{l_1, l_2, \ldots, l_n\}$          □
        in  $\bigsqcup_{i=1}^{n} Succ(l_i, AbEnv_0)$
```

The collection of all interleavings was also done by Codognet *et al.* [10], although applied to another application. In that framework, a similar, but different, local fixed-point calculation was used. Since the functions *AbsRed*, *Exit*, *Succ*, *Match*, *Punify*, and *Aunify* are monotonic and the domain L is finite, the least fixed point exists by Kleene's fixed-point theorem [37].

Reiterating (Section 1), we are primarily interested in the local reuse application, not in developing a new framework. Codognet's application of suspension analysis raises a more critical question. If suspension information were known to the reuse analysis, data dependencies among body goals could be determined, and a more accurate estimation of threadedness could be obtained. We believe that the two applications could be pipelined: our local fixed-point calculation could be modified to accept a partial ordering of body goals produced by the suspension analysis. Alternatively, an attempt could be made to join the two applications within a more complex domain, to save analysis time with respect to the pipeline. These are topics of future research.

4.6 Abstracting Head–Goal Matching, Ask Tests and Tell Unifications

Head–Goal Matching

Function *Match*(*Goal*, *Head*, *AbEnv*) is now defined. Since we are dealing with canonical-form programs, head matching involves variable–variable unification only. We first unify the head H and the goal G obtaining an idempotent substitution $\theta = \{X_1 \mapsto Y_1, \ldots, X_n \mapsto Y_n\}$ such that $\theta H = G$. Next the threadedness is propagated by repeating the following rules until there is no change in the abstract environment. For each $X_i \mapsto Y_i \in \theta$,[8]

- if $\{X_i \mapsto M\} \subseteq AbEnv$, then there is no change in *AbEnv*: the formal parameter remains multiple threaded.

[8]Recall that X_i is the formal parameter and Y_i is the actual parameter. At this point, the formal parameter has been assigned an abstract value via initialization only (Section 4.4).

- if $\{X_i \mapsto S,\ Y_i \mapsto M\ \} \subseteq$ *AbEnv*, then update *AbEnv* with $X_i \mapsto M$. For each Z that may share with X_i, update *AbEnv* with $Z \mapsto M$.

- if $\{X_i \mapsto S,\ Y_i \mapsto \langle S, T_1, ..., T_n\rangle\} \subseteq$ *AbEnv*, for $n \geq 0$, then update *AbEnv* with $X_i \mapsto \langle S, T_1, ..., T_n\rangle$. For each Z that may share with X_i, update *AbEnv* with $Z \mapsto \langle S, ...\rangle$.

Ask Goals

The testing of *Ask* goals is simulated by function *Punify*. Recall that ask equations do not bind goal variables; they can at most bind the variables of the clause being matched with the current goal. Function *Punify*(*Ask*, *AbEnv*) is discussed below. A unification equation in the ask part can be in one of two forms (recall the constraints of Section 4.2):

- $X = Y$ in which both X and Y must also appear in the head. This goal simply tests for equality without creating any bindings and hence *Punify* ignores such ask goals.

- $X = f(...Y...)$ where X is a head variable and Y is the i^{th} argument of term f. This goal does not create a binding for X in the actual execution. If the term Y is a head variable or contains only head variables, then subterms of Y cannot be bound to subterms of X but can merely be checked for equality. Hence the threadedness of X and Y are not affected.

  If the term Y contains local variables, then there are a few cases to consider. Recall that each of local variables in Y could only have been initialized (Section 4.4) to either S or M.

  - if X is M then all local variables in Y become M.
  - if X is S then no change is made.
  - if X is $\langle S, ..., T_i, ...\rangle$, there are two possibilities. If T_i is S then no change. If T_i is M, then all local variables in Y become M.

Tell Unifications

Function *Aunify* approximates the effect of tell unifications. We assume that tell unification equations are in solved form, i.e., the LHS variables occur exactly once in the tell part (Section 4.2). Furthermore, in each equation of the form $X = Y$ (where X and Y are variables) both X and Y must appear in the head or the ask part of the guard.[9] We propagate the threadedness by repeating the following rules until there is no change in the abstract environment. First we consider tell unifications of the form $X = Y$.

1. For each equation $X = Y$ such that $\{X \mapsto M, Y \mapsto \langle S, T_1, ..., T_n\rangle\} \subseteq$ *AbEnv* for $n \geq 0$, update *AbEnv* with $Y \mapsto M$. Whenever Y is updated, for each Z that may share with Y, update *AbEnv* with $Z \mapsto M$. We are simply propagating the multiple threadedness of X to its components and their aliases.

[9]This requirement is enforced by normalization and serves only to simplify the exposition of the abstract unification algorithm.

The symmetric case of $\{X \mapsto \langle S, \dots \rangle, Y \mapsto M\}$ are treated similarly. Abstraction of a tell unification equation thus involves propagating the lub of the abstractions of the two arguments.

2. For each equation $X = Y$ such that either $\{X \mapsto M, Y \mapsto M\} \subseteq AbEnv$ or $\{X \mapsto S, Y \mapsto S\} \subseteq AbEnv$, there is no change in *AbEnv*.

3. For each equation $X = Y$ such that $\{X \mapsto S, Y \mapsto \langle S, T_1, ..., T_n \rangle\} \subseteq AbEnv$, update *AbEnv* with $X \mapsto \langle S, T_1, ..., T_n \rangle$. Whenever X is updated, for each Z that may share with X, update *AbEnv* with $Z \mapsto \langle S, T_1, ..., T_n \rangle$. The symmetric case of $\{X \mapsto \langle S, T_1, ..., T_n \rangle, Y \mapsto S\}$ is treated similarly.

4. For each equation $X = Y$ such that $\{X \mapsto \langle S, T_1, ..., T_m \rangle, Y \mapsto \langle S, U_1, ..., U_n \rangle\} \subseteq AbEnv$, for $n > m > 0$, update *AbEnv* with $X \mapsto \alpha$, where

$$\alpha = \langle S, T_1 \sqcup U_1, ..., T_m \sqcup U_m, U_{m+1}, ..., U_n \rangle.$$

Whenever X is updated, for each Z that may share with X, update *AbEnv* with $Z \mapsto \alpha$. The case of $n \leq m$ is treated similarly.

Next we consider tell unifications of the form $X = f(...Y...)$. For the purposes of exposition, this case is handled by collapsing it into the previous case. The right-hand-side can be represented, for the purposes of this unification, by a tuple $\langle S, T_1, ..., T_i, ..., T_n \rangle$ where n is the arity of structure f. Assuming Y is the ith component of f, then if Y is S or M, then T_i is assigned S or M respectively. If Y is $\langle S, U_1, U_2, ..., U_k \rangle$, we collapse this value into S if all U_j for $j \leq k$ are S. Otherwise, if any U_j is M, then the value for Y is collapsed into M. Once we summarize the right-hand side of the tell unification in this manner, we apply the previous propagation rules.

The following example illustrates how tell unification equations effect propagation of threadedness: $\{X_1 = f(X_2, X_3),\ X_2 = X_4\}$ and X_2 and X_3 are shared. Assume that initially X_1 is $\langle S, S, S \rangle$, X_2 is S, X_3 is S, and X_4 is M. Since X_4 is M, X_2 becomes M, and then because X_2 is the first argument of a structure assigned to X_1, X_1 becomes $\langle S, M, S \rangle$. Now X_2 and X_3 share, so X_3 becomes M. Since X_3 is the second argument of a structure that is assigned to X_1, X_1 is updated to $\langle S, M, M \rangle$.

4.7 A Quick Example

In this section we illustrate the analysis for the quicksort program as listed in Figure 4, which is in flattened canonical form. First, the initial abstract substitution is computed for each program variable in each clause. All program variables are initialized to S and there are no aliases. Let us assume that the input abstract environment is $\{Z_1 = S, Z_2 = S\}$ in the query " ?- *qsort*(Z_1, Z_2)." On completing the analysis, we obtain abstract substitutions for all program variables. In other words, we have identified which of the incoming arguments are of types S and M. In this example, it is determined that all 7 potential applications of reuse are safe: variables `L1` and `L2` in *qsort/2*, `X1` and `X2` in *append/3*, and `List1`, `List2` and

```
qsort(L1, Sorted1) :-
   L1 = [] :
   Sorted1 = [] |
      true.
qsort(L2, Sorted2) :-
   L2 = [Pivot|Rest] :
   Large = [Pivot|LS] |
      split(Rest, Pivot, S, L),
      qsort(S, SS),
      qsort(L, LS),
      append(SS, Large, Sorted2).

append(X1, Y1, Z1) :-
   X1 = [] :
   Y1 = Z1 |
      true.
append(X2, Y2, Z2) :-
   X2 = [H|T] :
   Z2 = [H|Temp] |
      append(T, Y2, Temp).
```

```
split(List1, P3, S3, L3) :-
   List1 = [] :
   S3 = [], L3 = [] |
      true.
split(List2, P4, S4, L4) :-
   List2 = [X|Xs], X =< P4 :
   S4 = [X|Rest] |
      split(Xs, P4, Rest, L4).
split(List3, P5, S5, L5) :-
   List3 = [X3|Xs3], X3 =< P5 :
   L5 = [X3|Rest] |
      split(Xs3, P5, S5, Rest).
```

Figure 4: QuickSort Program in FCP(:)

`List3` in *split/4*. A compiler can generate the appropriate reuse instructions (see Section 5.1) after generating the code for inspecting the head arguments.

Figure 5 illustrates the gain in precision afforded by the depth-one domain. In the body of *funny_qsort/2*, *gen/1* generates a list and *rev/2* reverses a list. The original list, sorted list, and reversed list are packaged together in $f/3$. Structure $f/3$ can be reused in *funny_use/2* because the output variable `Out` is given the abstract domain value $\langle S, M, M, S\rangle$ rather M which would result from a depth-zero scheme.

To illustrate one imprecision of our scheme, consider the query " *?- funny_qsort*$(\_, f(\_, S, T))$," which does not output the initial list of consecutive integers, `L0`. Variable `L0` is given the abstract value M during initialization, precluding its reuse in *qsort/2*. The problem is that initialization is performed local to the clause without regard for how the clause is invoked. We chose to avoid complicating initialization rather than optimizing such cases.

Another imprecision arises from abstracting together all functors bound to a variable. Greater accuracy could come from tracking each functor with its own threadedness.

4.8 Comments

The presence of uninstantiated variable(s) in the top-level of a structure renders the structure unsuitable for reuse, even if the structure is single threaded. The reason is because a producer of the unbound variable may bind its value *after* the enclosing structure has been reused. This might result in an erroneous unification failure.

```
?- funny_qsort(A,B),
   funny_use(B,C).

funny_qsort(L0, Out) :- true :
   Out = f(L0,L1,L2) |
   gen(L0),
   qsort(L0,L1),
   rev(L1,L2).

funny_use(Old, New) :-
   Old = f(D,E,F) :
   New = f(F,D,E) |
   true.
```

Figure 5: Modified QuickSort Program in FCP(:)

This problem is avoided by always allocating variable cells outside of structures and placing only pointers, to the variable cells, inside structures. While this permits reuse, it introduces extra dereference operations and may also increase memory consumption. These tradeoffs have been quantitatively analyzed by Foster and Winsborough [17].

The algorithm was presented specifically for flat concurrent logic programs. Any effort to extend the algorithm to more expressive languages will likely require modification of the rules for assigning and propagating threadedness, but not the overall control. For example, concurrent constraint logic programs will require that *Ask* guards be considered for calculating variable instances because their immediate reduction before commit is no longer guaranteed. Because such modifications will likely make the analysis less accurate, we cannot comment about the *practicality* of extending the algorithm.

5 Experimental Results

In this section we review two alternative committed-choice language instruction-set extensions for exploiting reuse information. The extensions are from the Strand abstract machine and the PDSS emulator. These extensions are similar, and deserve some explanation to put our empirical performance measurements in context. A performance comparison between our method and MRB is presented. The main purpose of this analysis is to illustrate that our analysis technique in fact works! Of course, further research is needed to present a full characterization of the utility of the scheme for real benchmarks.

5.1 Reuse Instruction Sets

Foster and Winsborough [17] describe a reuse instruction set for the Strand abstract machine. The extension includes:

- `test_list_r(L,H,T)` — If a register `L` references a list structure, then place a reference to the list in reuse register R, and place references, to the head and tail, in `H` and `T`, respectively.
- `assign_list_r(L)` — Place a reference to the list structure, referenced by reuse register R, into register `L`, and let the structure pointer point to the head of the list.
- `reuse_list_tail_r(L)` — Place a reference to the list structure, referenced by reuse register R, into register `L`, and let the structure pointer point to the tail of the list. Here we avoid the write mode unification of the head cell.

The reuse instructions use an implicit operand, the reuse register `R`, or a set of reuse registers. The reuse register is effectively a fast "free list" of currently reusable structures. This method is an efficient way of managing reusable dead structures for deferred reuse.

In contrast, the PDSS system [20] implements a reuse instruction set, designed around the MRB method, for the KL1 abstract machine. Deferred reuse is based on the `collect` operation which places reusable structures in free lists. When a new structure is required, it may be allocated from the free list. PDSS also includes instructions for instant reuse. The extensions for instant reuse include:

- `put_reused_func(Rvect,OldVect,Atom)` — Set the `Rvect` to point to the same location as `Oldvect`, set the name of functor to `Atom`.
- `put_reused_list(Rlist,OldList)` — Set `Rlist` to point to the same location as pointed by `OldList`.

Instant reuse is more efficient than deferred reuse since the intermediate move onto the free list is avoided. However, recall from Section 4.8 that a runtime variable check is needed for each structure argument. In our empirical experiments with reuse analysis, presented in the next section, the PDSS system was used. Since PDSS allocates unbound variables outside of structures, variable checks are not needed, so our comparison is fair. It is an open research question as to the performance tradeoff between allocating variables inside structures and doing this check, or allocating variables outside structures and thereby incurring additional dereferencing.

5.2 Performance

Measurements were made with the PDSS emulator running on a Sun SparcStation I. Six small benchmark programs were analyzed: append, insert, primes, qsort, pascal, and triangle. Insert constructs a binary tree of integers. Prime uses the Sieve of Eratosthenes to generate prime numbers. Qsort is the standard quicksort algorithm previously shown. Pascal generates the 32^{nd} row of Pascal's Triangle. Triangle solves the triangle puzzle of size 15. For each benchmark, three compiled versions were generated:

Naive — A version with no collect nor reuse instructions. This program is used as a basis for comparison.

Program	Heap Usage (Words)			Execution Time (sec)			
	Naive	Dynamic	Static	Naive	Dynamic	Static	% Save‡
insert	1,500,000	6,314	6,314	52.2	53.6	50.5	5.6
append	5,000,000	6,202	6,202	111.0	114.3	106.5	6.7
prime	323,786	12,128	12,158	11.1	11.3	10.5	7.0
qsort	8,000,000	61,725	61,725	234.0	237.5	221.7	6.7
pascal	167,070	127,072	147,270	12.1	12.9	12.3	4.7
triangle	543,809	539,523	—	60.5	63.8	—	—

‡ (Static − Dynamic)/Dynamic

Table 1: Heap Usage and Execution Time: Comparison of No Optimization, Dynamic, and Static Analysis

Dynamic — A version with collect instructions as generated by the existing PDSS compiler.

Static — A version with instant reuse instructions appropriately used and no collect operations are used.

Note that the Naive and Static systems still have MRB management overheads associated with individual instructions that make bindings, potentially requiring modifying the MRB.

Both static *and* dynamic methods can be used together in a hybrid scheme. However, in this study we wish to compare the effectiveness of reuse with that of the MRB scheme, and therefore we do not present results concerning the hybrid. The heap usage patterns in the benchmarks are presented in Table 1. The measurements in Table 1 reflect the behavior we expected from the benchmarks. The benchmarks illustrate classes of full, partial, and no-reuse programs. Insert, append, prime and qsort extensively use stream-based single producer/single consumer communication. Our algorithm predicted potential for full instant reuse, as confirmed in the table. In these benchmarks the heap memory requirements of the static and dynamic versions are nearly identical. This demonstrates that it is possible to achieve as much efficiency as collect in benchmarks where a large number of single-threaded structures are constructed.

In pascal, where only 50% $((167,070-147,270)/(167,070-127,072))$ of single-threaded structures were statically determined, the heap requirements of the static version are only slightly higher than that of the dynamic version. In triangle, the board structure is multiple threaded. The `collect` operations almost never succeed in reclaiming memory and the memory requirements of the dynamic and naive versions of the program are nearly the same. Since reuse is not possible, we did not generate a static version of the program.

The memory requirements of the naive versions of the benchmarks can be several times higher than the static and dynamic versions. The extent of memory reuse is highly program dependent however. These measurements are meant only to illustrate how our algorithm can exploit reuse when conditions are ripe. To further illustrate the effectiveness of static analysis, the execution times of the benchmarks are also presented in Table 1. By compiling reuse into the program, the execution speed is consistently better than that obtained through the MRB optimization. By implementing reuse more efficiently (than in PDSS), the savings may be even higher. The results are biased against the static system which still pays the overhead of MRB manipulation within abstract machine instructions that make bindings. Thus the savings appear lower than what could be achieved in a completely MRB-free system. Note that in PDSS, even after stop-and-copy garbage collection, the naive version of a benchmark runs as fast as, if not faster than, the dynamic version. This lends support to our claim that in the absence of special hardware to implement MRB, a good garbage collector is important. To improve the combination of reference counting and a good garbage collector, memory reuse is necessary.

In programs with a preponderance of multiple-consumer communication, the `collect` operations simply add runtime overhead without reclaiming memory. In such a case, the program performs better if the `collect` operations are simply removed. This is evident in triangle, for instance, where our analysis determined that there is no scope for reuse. The naive version outperforms the version with `collect` operations.

In programs where the majority of the structures have single consumers, `collect` operations are avoided wherever instant reuse is possible. This reduces the overhead of free-list management. The static version is 6–7% faster than the dynamic version in insert, append, prime, and qsort, where 100% instant reuse was possible. Even in pascal, where instant reuse was only 50% as memory-efficient as collect, the static version was 4.7% faster.

6 Summary and Conclusions

We have introduced a new compile-time analysis method for determining single-threadedness of data structures in concurrent logic programs. The analysis is formulated in the framework of an abstract interpreter for FCP(:). The information produced is the "threadedness" of each logical variable: either single or multiple threaded, referring to the number of consumer processes associated with the variable. To avoid over-conservative approximation, the analysis imposes simple syntactic constraints that can easily be achieved at compile time without loss of generality. Sharing information is required to ensure correctness of the analysis, and its use is integrated into the threadedness propagation algorithm.

Empirical results indicate that the analysis enables local memory reuse that is comparable to the multiple reference bit (MRB) scheme [7] in terms of amount of memory saved. The proposed method is not suggested as an alternative to GC, but as a supplement to GC. The improvements in execution speed of (4.7%–7.0%) in the examples we tested provide evidence to this claim. The modest savings in execution speed is due to the overheads of MRB which we did not completely remove due to the complexity of the KL1 run-time system.

It would be interesting to explore more complex reuse schemes. In recent work on reuse of arrays in functional programs, Sastry *et al.* [33] propose a powerset domain of variables for gathering liveness information of the program variables. Chase *et al.* [6] use more complex domains for structure shape analysis. Another idea is to maintain information about the threadedness sub-structures up to a certain depth. Depth-k abstractions of Sato and Tamaki [34] may be useful in this regard.

Another research direction worth exploring is the integration of sharing analysis and threadedness analysis. Using a partial order for reducing body literals (based on, say, suspension analysis), instead of all possible interleavings considered in this paper, is also a promising avenue.

Acknowledgements

E. Tick was supported by an NSF Presidential Young Investigator award, with funding from Sequent Computer Systems Inc, and by a research grant from the Institute of New Generation Computer Technology (ICOT). The original PDSS measurements were made by Sreeram Duvvuru.

References

[1] H. Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, Cambridge, MA, 1991.

[2] H. G. Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4):280–294, 1978.

[3] A. Bloss. *Path Analysis and Optimization of Non-Strict Functional Languages.* PhD thesis, Yale University, Dept. of Computer Science, New Haven, May 1989.

[4] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10(2):91–124, February 1991.

[5] M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen. Abstract Interpretation: Towards the Global Optimization of Prolog Programs. In *International Symposium on Logic Programming*, pages 192–204. San Francisco, IEEE Computer Society Press, August 1987.

[6] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of Pointers and Structures. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–309, White Plains, NY, June 1990. ACM Press.

[7] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *International Conference on Logic Programming*, pages 276–293. University of Melbourne, MIT Press, May 1987.

[8] M. Codish, M. Falaschi, and K. Marriott. Suspension Analysis for Concurrent Logic Programs. In *International Conference on Logic Programming*, pages 331–345. Paris, MIT Press, June 1991.

[9] M. Codish, M. Falaschi, K. Marriott, and W. Winsborough. Efficient Analysis of Concurrent Constraint Logic Programs. In *International Colloquium on Automata, Languages and Programming*, number 700 in Lecture Notes in Computer Science, pages 633–644, Lund, Sweden, July 1993. Springer-Verlag.

[10] C. Codognet, P. Codognet, and M. Corsini. Abstract Interpretation of Concurrent Logic Languages. In *North American Conference on Logic Programming*, pages 215–232. Austin, MIT Press, October 1990.

[11] J. Cohen. Garbage Collection of Linked Data Structures. *ACM Computing Surveys*, 13:341–367, September 1981.

[12] S. K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, July 1989.

[13] L. P. Deutsch and D. G. Bobrow. An Efficient Incremental, Automatic Garbage Collector. *Communications of the ACM*, 19:522–526, September 1976.

[14] L. H. Eriksson and M. Rayner. Incorporating Mutable Arrays into Logic Programming. In *Proceedings of the Second International Logic Programming Conference*, pages 76–82, Uppsala University, 1984.

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

[16] I. Foster and W. Winsborough. A Computational Collecting Semantics for Strand. Research report, Argonne National Laboratory, 1990. Unpublished.

[17] I. Foster and W. Winsborough. Copy Avoidance through Compile-Time Analysis and Local Reuse. In *International Symposium on Logic Programming*, pages 455–469. San Diego, MIT Press, November 1991.

[18] P. Hudak. A Semantic Model of Reference Counting and Its Abstraction. In *Conference on Lisp and Functional Programming*, pages 351–363, Cambridge, 1986. ACM Press.

[19] P. Hudak and A. Bloss. The Aggregate Update Problem in Functional Programming Languages. In *SIGPLAN Symposium on Principles of Programming Languages*, pages 300–314, New Orleans, January 1985. ACM Press.

[20] ICOT. *PDSS Manual (Version 2.52e).* 21F Mita Kokusai Bldg, 1-4-28 Mita, Minato-ku Tokyo 108, Japan, February 1989.

[21] A. Imai and E. Tick. Evaluation of Parallel Copying Garbage Collection on a Shared-Memory Multiprocessor. *IEEE Transactions on Parallel and Distributed Computing*, 4(9):1030–1040, September 1993.

[22] Y. Inamura, N. Ichiyoshi, K. Rokusawa, and K. Nakajima. Optimization Techniques Using the MRB and Their Evaluation on the Multi-PSI/V2. In *North American Conference on Logic Programming*, pages 907–921. Cleveland, MIT Press, October 1989.

[23] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and its Instruction Set. In *International Symposium on Logic Programming*, pages 468–477. San Francisco, IEEE Computer Society Press, August 1987.

[24] A. King. A Framework for Sharing Analysis. In E. Tick and G. Succi, editors, *Implementations of Logic Programming Systems.* Kluwer Academic Publishers, 1994.

[25] A. King. A Synergistic Analysis for Sharing and Groundness which Traces Linearity. In *European Symposium on Programming.* Springer Verlag, 1994.

[26] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms.* Addison-Wesley, Reading MA, 2nd edition, 1973.

[27] J-L. Lassez, M.J. Maher, and K. Marriott. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–626. Morgan Kaufmann Publishers Inc., 1988.

[28] K. Marriott and H. Sondergaard. Abstract Interpretation of Logic Programs. In *North American Conference on Logic Programming*. Cleveland, October 1989. tutorial notes.

[29] C. S. Mellish. Some Global Optimizations for a Prolog Compiler. *Journal of Logic Programming*, 2(1):43–66, April 1985.

[30] A. Mulkers, W. Winsborough, and M. Bruynooghe. Analysis of Shared Data Structures for Compile-Time Garbage Collection in Logic Programs. In *International Conference on Logic Programming*, pages 747–762. Jerusalem, MIT Press, June 1990.

[31] A. Mulkers, W. Winsborough, and M. Bruynooghe. Live-Structure Data-Flow Analysis for Prolog. *ACM Transactions on Programming Languages and Systems*, 1993. To Appear.

[32] V. A. Saraswat, K. Kahn, and J. Levy. Janus: A Step Towards Distributed Constraint Programming. In *North American Conference on Logic Programming*, pages 431–446. Austin, MIT Press, October 1990.

[33] A. V. S. Sastry, W. Clinger, and Z. Ariola. Order-of-Evaluation Analysis for Destructive Updates in Strict Functional Languages with Flat Aggregates. In *Conference on Functional Programming Languages and Computer Architecture*, pages 266–275. Copenhagen, ACM Press, June 1993.

[34] T. Sato and H. Tamaki. Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Science*, 34:227–240, 1984.

[35] D. Schmidt. Detecting Global Variables in Denotational Specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, 1985.

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

[37] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge MA., first edition, 1977.

[38] R. Sundararajan and J. Conery. An Abstract Interpretation Scheme for Groundness, Freeness and Sharing Analysis of Logic Programs. In *Conference on Foundations of Software Technology and Theoretical Computer Science*, number 652 in Lecture Notes in Computer Science, pages 203–216. New Delhi, Springer-Verlag, December 1992.

[39] R. Sundararajan, A. V. S. Sastry, and E. Tick. Variable Threadedness Analysis for Concurrent Logic Programs. In *Joint International Conference and Symposium on Logic Programming*, pages 493–508. Washington D.C., MIT Press, November 1992.

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

[41] W. Winsborough. personal communication, 1993.

[42] E. Yardeni, S. Kliger, and E. Shapiro. The Languages FCP(:) and FCP(:,?). *New Generation Computing*, 7(2–3):89–107, January 1990.