A Compile-Time Memory-Reuse Scheme for Concurrent Logic Programs

A. V. S. Sastry, R. Sundararajan, E. Tick

CIS-TR-91-24a
July 1992

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 run-time 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 version of a paper appearing in the *Joint International Conference and Symposium on Logic Programming*, Washington D.C., November 1992.

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

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 [8] surveys a range of traditional garbage collection techniques, classified as *mark and sweep* algorithms. Reference counting [22], efficient garbage collection algorithms such as Baker's algorithm [1, 17], and incremental garbage collection schemes [10], 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 [11] 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 [6] scheme is an example of binary reference counting. Much work has been done in compile-time detection and reuse of structures in functional programs [15]. 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 [25] and Bruynooghe [4]) 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 [29]. 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. Correctness and termination proofs are given in the Appendix.

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(:) [29], a guarded Horn clause has the form "H :- $Ask_1, \ldots, Ask_m$: $Tell_1, \ldots, Tell_n \mid B_1, \ldots, B_0$." 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

```
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

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|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 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[1] The precise semantics of this code is unimportant. Critically, instruction `put_list` allocates a new list cell on the heap for every call to

[1]The instruction set is for the FGHC abstract machine developed by Kimura and Chikayama [21], similar in some respects to Warren's Abstract Machine (WAM) for Prolog [33].

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 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 [28], Hudak's work on abstraction of reference count for a call-by-value language with a fixed order of evaluation [14], and update analysis for a first-order lazy functional language with flat aggregates using a non-standard semantics called path semantics [2]. All the above analyses are for sequential implementations. The analysis of single threading or storage reusability in logic languages is different from functional languages because of the power of unification and logical variables available in the former.

Another important implementation area is shape analysis, the static derivation of data structure composition. Recent work by Chase *et al* [5] 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 [14].

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.

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 [27].

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 [10]. 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 speed implementation.

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 [6]. 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 [18]. Several MRB-based optimizations are described in [18]. 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

```
foo([A|B]) :- true : 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

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 (among others) have been proposed for sequential Prolog (e.g., [24, 4, 9, 31]). But these techniques do *not* extend easily to concurrent logic languages, where no assumptions can be made about the order of execution of the goals or about the interleaving of their reduction. The only 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 [13]. They sketch a collecting semantics for Strand programs in which a program state is associated with a record of the program components that operated on it. The collecting semantics is then converted into an abstract interpretation framework by supplying an abstract domain in order to identify single consumers. The analysis details are in an unpublished

draft [12], hence it is premature to compare their scheme with ours.

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 abstract interpretation to detect threadedness in flat committed-choice logic programs and use this information to generate reuse instructions.

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 [32] defines 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. Saraswat [26] proposes a related language, Janus, which allows only *SS* 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, *SS* or *SM*, 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. 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 imply *incoming* data, by the semantics of concurrent logic languages.[2] 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:

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

[2] Computation will suspend until the input arguments are sufficiently instantiated.

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!

We also assume that *structure sharing analysis* has been done, and that the results of the analysis are available. We envision sharing analysis similar to [31, 19], with two modifications. First, the analysis must work for concurrent languages. Second, if it is determined that two variables may share, no subsequent grounding can undo this sharing. 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 sub-term of a structure affect another sub-term 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 the tell part of the guard and it contains at least one variable local to the clause (i.e., the variable does not appear in the head or the ask part or in the RHS of a tell equation of the form $X = f(\ldots, Y, \ldots)$ where X is a variable that appears in the head or the ask part) and that variable is multiple threaded. If the implementation allocates variables *inside* structures (as is usually the case), 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 conservatively assume that variables may be created within a structure. Relaxation of this assumption, whenever appropriate, can lead to more precise analysis. In order to be precise, however, the abstract domain must be expressive enough to represent that the top level of a structure may be single threaded and at the same time any of its sub-structures may be multiple threaded. The analysis presented in this paper conservatively treats a structure as multiple threaded when one of its sub-structures is 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.

In the rest of the paper, we will discuss mainly the threadedness of variables. The threadedness of structures can be derived, with the principles discussed above, from the threadedness of its components.

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 Concurrent Guarded Horn Clause (FGHC), Strand, and similar languages [29]. Proof of soundness of our analysis is given in the Appendix.

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 [29]. The knowledgeable reader may wish to skip to the next sub-section.

A computation state is a triple $\langle G, \theta, i \rangle$ 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 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 $\langle G, \varepsilon, 0 \rangle$ consists of the initial goal G, the empty substitution ε, and the renaming index 0.

> *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 $\langle true, \theta, i \rangle$ is a terminal state that denotes successful computation and $\langle fail, \theta, i \rangle$ denotes finitely failed computation. If no transition is applicable to a state $S = \langle A_1, \ldots, A_n, \theta, i \rangle$ $(n \geq 1)$ 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

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

□

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

□

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

□

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 [23]. These conditions facilitate reasoning about structures that are definitely created at commit time (see Section 4.6).

1. Head arguments are distinct variables. This restriction simply moves the matching of head arguments with those of goal to the ask part of the guard.

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.
   - $X = f(...)$ where variable X must appear in the head.

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

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.
   - $X = f(...)$

   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 $\mathcal{P}(Computation)$. 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 two-point complete lattice L whose least element is SS (Single Producer/Single Consumer) and the top is SM (Single Producer/Multiple Consumers). Thus, our abstract domain $AbEnv$ is $PVar \rightarrow \{SS, SM\}$ and the partial ordering on $AbEnv$ is the usual point-wise ordering.

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(Term, State)* which is true whenever the term T occurs *more than twice* in state *State* (counting each occurrence of a term T in each atom in the state).

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

$$\alpha'(S) = \{ Z \mapsto SM \quad \text{if } Z \in \{ rename^{-1}(X) \mid multi_occurs(X,S) \vee (S = \langle G, \theta, i \rangle \wedge T \in sub\text{-}terms(X\theta) \wedge multi_occurs(T,S)) \} \\ Z \mapsto SS \quad otherwise \}$$

□

A variable X (representing some data structure) can be multiple-threaded in one of two ways.

- X is multiple-threaded if it appears more than twice in the current state.
- Suppose X is bound to a term $f(...T...)$ in the current substitution θ and a sub-term, say T, occurs more than twice in the current state. Then X may or may not be multiple-threaded. Consider:

$$:- p(X).$$

$$p(X_1) :- true : X_1 = f(a) \mid q(X_1), r(X_1), s(X_1).$$

Let the initial state be $\langle p(X_1), \{\}, 0 \rangle$. The state $S1$ resulting from reducing the goal $p(X)$ with respect to the above clause is $\langle q(f(a)), r(f(a)), s(f(a)), \theta = \{X \mapsto f(a), X_1 \mapsto f(a)\}, 1 \rangle$. The sub-terms $f(a)$ of the literals q, r, and s represent the current binding of the variable X_1 (and X) shared by the literals q, r and s. Hence, in this case, both X and X_1 are multiple-threaded. On the other hand, if the goal $p(X)$ is reduced with respect to the following clause, we will obtain a similar state $S2$ but in which variables X, X_1 and X_2 will not be multiple-threaded.

$$p(X_1) :- true : X_1 = f(a), X_2 = f(a) \mid q(X_1), r(X_2), s(X_2).$$

There is not enough information in the states $S1$ and $S2$ to distinguish between them and hence we safely approximate X, X_1 (and X_2) to be multiple-threaded in both states.

Note that our abstract domain does not keep track of the threadedness of the top level of a structure and its sub-structures separately. This can be done by a more expressive abstract domain and corresponding abstract domain operations. We have developed such an expressive domain and are experimenting with its usefulness, the results of which will be reported elsewhere.

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

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

□

In the following subsections, we describe the abstract interpretation algorithm in detail. The algorithm consists of (i) abstract reduction which is composed of 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., [9, 3]).

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. All occurrences of the same variable in the head and the guards are counted as a single occurrence and each occurrence of a variable in the body is counted individually.

If a variable occurs two or fewer times, it is initialized to *SS*. If it occurs more than twice, it is initialized to *SM*, implying that variables that occur only in the guard (*Ask* or *Tell* part) are initialized to *SS*. The variables that occur only in the guard will inherit their threadedness from other structures with which they are matched/unified. The following example illustrates the computation of an initial approximation for the body occurrences.

$$f(X_1, X_2) \text{ :- } X_1 < X_2 \ : \ X_3 = t(X_1) \mid p(X_1, X_4), \ q(X_1, X_2, X_3, X_4).$$

Variable X_1 is initialized to *SM* since it occurs three times — once in the head and twice in the body of the clause. Variable X_2 starts as single-threaded because it occurs once in the head and once in the body. Similarly, $X_3 = X_4 = SS$. We refer to the initial environment of the clause, obtained with the above rules, as $AbEnv_{init} = \{X_1 \mapsto SM, X_2 \mapsto SS, X_3 \mapsto SS, X_4 \mapsto SS\}$ and 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:

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

Assume that the tell goal may cause X_2 and X_3 to share. Without considering sharing, the number of occurrences of X_2 and X_3 are each two. However, considering sharing, we count four occurrences of each. Thus we initialize each to *SM*.

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.

Definition: $AbsRed: Atom \times Clause \times AbEnv \rightarrow AbEnv$

$AbsRed(Goal, C, AbEnv_{call}) =$

$$\begin{array}{llcll}
\textbf{let} & H \text{ :- } A : T \mid B & = & rename(C) & (1)\\
& AbEnv_{init} & = & init(H \text{ :- } A : T \mid B) & (2)\\
& E' & = & Match(Goal, H, AbEnv_{call} \cup AbEnv_{init}) & (3)\\
& E'' & = & Punify(A, E') & (4)\\
& E''' & = & Aunify(T, E'') & (5)
\end{array}$$

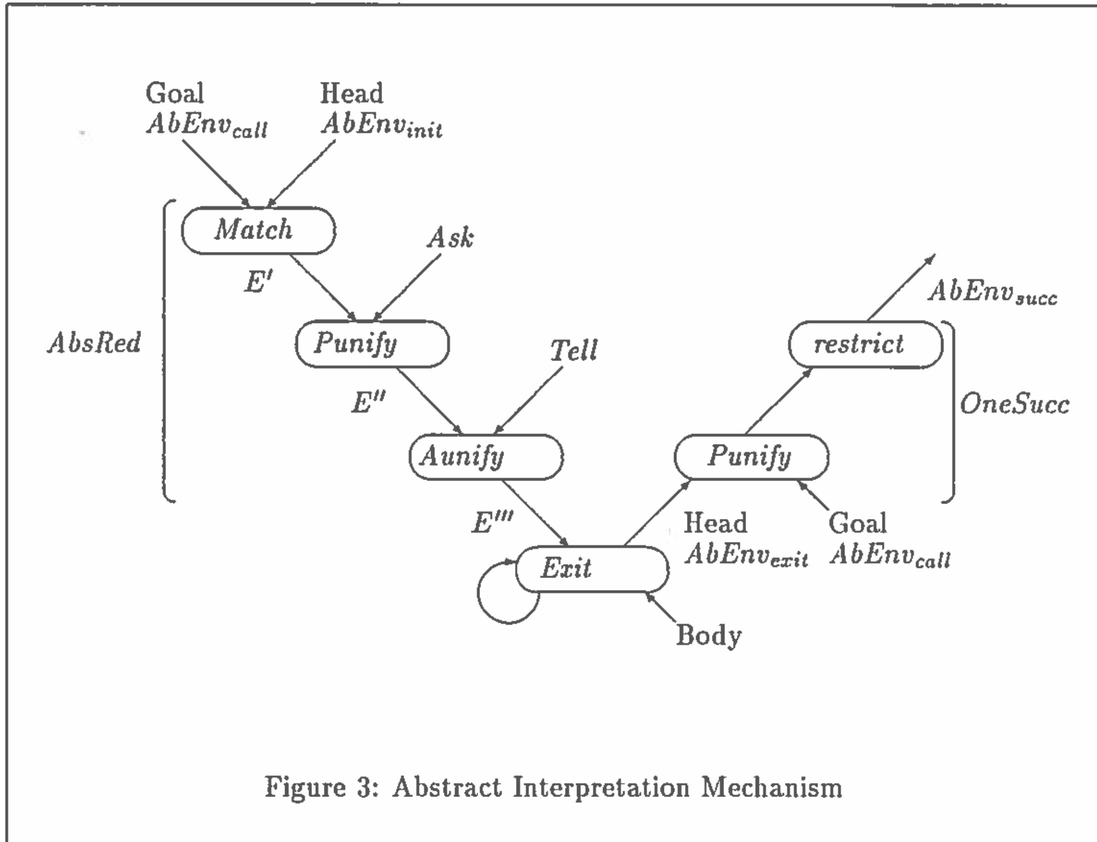

Figure 3: Abstract Interpretation Mechanism

in

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

□

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 using function *init* 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). 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 G 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 $AbEnv_{succ} = Succ(Query, AbEnv_{init}^{query})$, where *Query* is the top-level procedure invocation and $AbEnv_{init}^{query}$ 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

$$AbEnv^i_{entry} = AbsRed(Goal, p_i, AbEnv_{call}) \quad (1)$$

$$AbEnv^i_{exit} = Exit(b_i, AbEnv^i_{entry}) \quad (2)$$

in

$$\bigsqcup_{i=1}^{k} \left\{ OneSucc(Goal, p_i, AbEnv_{call}, AbEnv^i_{exit}) \right\} \quad (3)$$

□

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 G.

Definition: $OneSucc: Atom \times Clause \times AbEnv \times AbEnv \rightarrow AbEnv$

$OneSucc(Goal, C, AbEnv_{call}, AbEnv_{exit}) =$

let

$$(C', AbEnv_{exit'}) = rename((C, AbEnv_{exit})) \quad (1)$$

$$H :- A : T \mid B = C' \quad (2)$$

$$E = Aunify(\{Goal = H\}, AbEnv_{call} \cup AbEnv_{exit'}) \quad (3)$$

in

$$restrict(E, Vars(AbEnv_{call})) \quad (4)$$

□

In concurrent logic programs, body goals may execute in any order and their execution may also be interleaved. 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: $ExitIter : Body \times AbEnv \rightarrow AbEnv$

$$
\begin{array}{lll}
\multicolumn{3}{l}{ExitIter(Body,\ AbEnv_0) =} \\
\textbf{if} & empty(Body) \ \textbf{then} & \\
 & AbEnv_0 & \\
\textbf{else} & \textbf{let}\ Body = \{l_1, l_2, \ldots, l_n\} & \\
 & \textbf{in} & \\
 & \quad \bigsqcup_{i=1}^{n} Succ(l_i, AbEnv_0) &
\end{array}
$$

□

A different local fixed-point calculation was first used by Codognet *et al.* [7]. 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 [30].

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 $X_i \mapsto Y_i \in \theta$,

- if $\{X_i \mapsto SM, Y_i \mapsto SS\} \subseteq AbEnv$, then update $AbEnv$ with $Y_i \mapsto SM$. If Y_i is updated, for each Z that may share with Y_i, update $AbEnv$ with $Z \mapsto SM$.
- or, if $\{X_i \mapsto SS, Y_i \mapsto SM\} \subseteq AbEnv$, then update $AbEnv$ with $X_i \mapsto SM$. If X_i is updated, for each Z that may share with Y_i, update $AbEnv$ with $Z \mapsto SM$.

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.

- $X = f(\ldots Y\ldots)$ where X is a head variable. This goal does not create a binding for X in the actual execution. However, Y may be bound to a component of X, if Y doesn't appear in the head. If Y is multiple-threaded but X is not, then X (and all variables that may share with X) will become multiple-threaded. If Y is not multiple-threaded but X is, then Y (and all variables that may share with Y) will become multiple-threaded. If Y is a head variable, then Y cannot be bound to a sub-component of X but can merely be checked for equality and hence the threadedness of X and Y are not affected.

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. The set of tell equations $X_1 = t_1, \ldots, X_n = t_n$ in solved form represents an idempotent substitution [23] $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$. An idempotent substitution can be viewed as a bipartite graph [20]. The variables X_i form a vertex set S and the variables that occur in t_i form another vertex set T and these two are disjoint. There is an edge from a vertex $X \in S$ to a vertex $Y \in T$ whenever $X \mapsto f(\ldots Y\ldots)$ or $X \mapsto Y$ is in θ. We propagate the threadedness by repeating the following rules until there is no change in the abstract environment.

1. For each $X \in S$ such that there is an edge between X and Y, and $\{X \mapsto SM, Y \mapsto SS\} \subseteq AbEnv$, update *AbEnv* with $Y \mapsto SM$. Whenever Y is updated, for each Z that may share with Y, update *AbEnv* with $Z \mapsto SM$. We are simply propagating the multiple-threadedness of X to its components and their aliases.

2. The symmetric case of $\{X \mapsto SS, Y \mapsto SM\}$ is treated similarly. Abstraction of a tell unification equation thus involves propagating the lub of the abstractions of the two arguments.

A more general, and therefore more precise, formulation of abstract tell unification involves keeping track of local and output variables,[3] in conjunction with extending the

[3] For languages that restrict unification to assignment, such as Strand, this condition is known precisely.

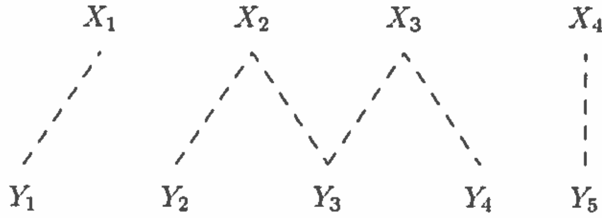

Figure 4: Approximating Tell Unifications

abstract domain to express information about the threadedness of sub-structures. The current formulation is imprecise in the following sense. Consider the structure created by the tell unification equation $X = f(Y_1, \ldots, Y_n)$ where X is a local variable, none of the Y_i is local and some Y_i is multiple-threaded. The structure X is reuseable if there are no more than two occurrences of X in the body of the clause, and those occurrences do not become multiple-threaded due to other reasons such as aliasing. However, our abstract domain cannot express that the top level of X is definitely single-threaded but other levels are not. This is a source of imprecision in our analysis.

As an example of the propagation steps, consider the set of tell unification equations $\{X_1 = f(Y_1, Y_2), X_2 = g(Y_2, Y_3), X_3 = h(Y_3, Y_4), X_4 = Y_5\}$ represented by the bipartite graph in Figure 4. Assume that initially only X_3 is multiple-threaded and let $\{(X_2, X_4), (Y_3, Y_5)\}$ be the set of pairs of variables that may share. Since X_3 is *SM*, Y_3, Y_4 also become *SM* and so does Y_5 since it shares with Y_3. X_4 becomes *SM* because of the equation $X_4 = Y_5$ and so does X_2 since it shares with X_4. We can now apply rule one again and this time Y_2 becomes *SM*. Since we cannot further apply the rules, the final result is $\{X_2, X_3, X_4, Y_2, Y_3, Y_4, Y_5\}$ are *SM* and the rest $\{X_1, Y_1\}$ are *SS*.

4.7 A Quick Example

In this section we illustrate the analysis for the quicksort program as listed in Figure 5, 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 *SS*. Let us assume that the input abstract environment is $\{Z_1 = SS, Z_2 = SS\}$ 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 *SS* and *SM*. In the example above, it was determined that all six potential applications of reuse are safe: variables `L1` and `L2` in *qsort/2*, `X1` and `X2` in *append/3*, and `List1`–`List3` in *split/4*. A compiler can generate the appropriate reuse instructions after the code for inspecting the head arguments.

Figure 6 illustrates a use of *qsort* that precludes reuse of the $f/3$ structure in

```
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 5: QuickSort Program in FCP(:)

```
?- 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(f(D,E,F), Z) :- true :
   Z = f(F,D,E) |
   true.
```

Figure 6: Modified QuickSort Program in FCP(:)

funny_use/2 because of the imprecision caused by the coarseness of the abstract 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 $f/3$. Since L0 occurs three times, it is *SM* (outside consumers of L0 must be protected). However, $f/3$ itself is actually single threaded, although we conservatively set it to *SM*. Also, consider the query "? – *funny_qsort*(_, X)," which does not use the initial list, yet our analysis prevents reuse of L0 inside the procedure. In the next section we discuss the problem that arises if at runtime, unbound variables occur in the top level of the structure.

4.8 The Problem of Unbound Structure Arguments

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.

This problem may be 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 [13].

If outside allocation is not the storage management policy, then a variable check of the top-level arguments must be conducted at runtime. Usually when a structure is inspected by a consumer, the components of the structure are decomposed, and copies of the elements placed in machine registers, or in the environment of the consumer. During this decomposing, the runtime check is relatively cheap to perform.

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 [13] 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 [16] 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 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.

Program	Heap Usage (Words)		
	Naive	Dynamic	Static
insert	1,500,000	6,314	6,314
append	5,000,000	6,202	6,202
prime	323,786	12,128	12,158
qsort	8,000,000	61,725	61,725
pascal	167,070	127,072	147,270
triangle	543,809	539,523	—

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

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 32nd 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.

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 efficacy 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

Program	Execution Time (sec)			% Save ‡
	Naive	Dynamic	Static	
insert	52.2	53.6	50.5	5.6
append	111.0	114.3	106.5	6.7
prime	11.1	11.3	10.5	7.0
qsort	234.0	237.5	221.7	6.7
pascal	12.1	12.9	12.3	4.7
triangle	60.5	63.8	—	—

‡ Reuse compared to Collect Optimization

Table 2: Execution Speed: Comparison of No Optimization, Dynamic, and Static Analysis

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 presented in Table 2. 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 better 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 [6] in terms of amount of memory saved. In addition, execution speed is improved (4.7%–7.0%) in some cases.

Acknowledgements

E. Tick was supported by an NSF Presidential Young Investigator award, with generous funding from Sequent Computer Systems Inc. We thank Sreeram Duvvuru for his help in this research.

References

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

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

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

[4] M. Bruynooghe *et al.* Abstract Interpretation: Towards the Global Optimization of Prolog Programs. In *International Symposium on Logic Programming*, pages 192–204. San Francisco, IEEE Computer Society, August 1987.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

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

[11] 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, 1984.

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

[13] 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.

[14] 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.

[15] 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.

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

[17] A. Imai and E. Tick. Evaluation of Parallel Copying Garbage Collection on a Shared-Memory Multiprocessor. *IEEE Transactions on Parallel and Distributed Computing*, 1992.

[18] 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.

[19] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *North American Conference on Logic Programming*, pages 154–165. Cleveland, MIT Press, October 1989.

[20] S. B. Jones and D. L. Metayer. Compile-time Garbage Collection by Sharing Analysis. In *Conference on Functional Programming Languages and Computer Architecture*, pages 54–74. ACM Press, 1989.

[21] 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, August 1987.

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

[23] 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.

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

[25] 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.

[26] 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.

[27] A. V. S. Sastry and W. Clinger. Order-of-Evaluation Analysis for Destructive Updates in Strict Functional Languages with Flat Aggregates. Technical Report CIS-TR-92-14, University of Oregon, Computer Science Department, 1992.

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

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

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

[31] R. Sundararajan. An Abstract Interpretation Scheme for Groundness, Freeness, and Sharing Analysis of Logic Programs. Technical Report CIS-TR-91-06, University of Oregon, Department of Computer Science, October 1991.

[32] K. Ueda and M. Morita. A New Implementation Technique for Flat GHC. In *International Conference on Logic Programming*, pages 3–17. Jerusalem, MIT Press, June 1990.

[33] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

Appendix

In this appendix we give the correctness and termination proofs of the reuse analysis algorithms previously described. For the purposes of analysis, we define our abstract interpretation by the following set of semantic equations. In these equations, the function abstract reduction is modeled by *AbsRed* defined in Sections 4.4–4.5. We define three functions called *rgc*, reducing a goal with respect to a particular clause, *rb*, reducing a set of goals, and *rgp*, reducing a goal by considering all the clauses (of a procedure) which potentially can match with it. In the following, we ignore details of variable renaming (which can be found in Section 4.5). The equations are:

$$
\begin{aligned}
rgc[\![g,\ (h \text{ :- } G \mid B)]\!]\ abenv_{call} &= \textbf{let}\quad abenv_0 = AbsRed(g,\ (h \text{ :- } G \mid B),\ abenv_{call}) \\
&\quad\ \textbf{in}\quad rb[\![B]\!]abenv_0 \\
&\quad\ \textbf{end} \\
rb[\![b_1, \ldots, b_k]\!]abenv_0 &= fix(\lambda\sigma.\ \bigsqcup_{i=1}^{k} rgp[\![b_i, p_i]\!](\sigma \sqcup abenv_0)) \\
rgp[\![g,\ (h_1 \text{ :- } G_1 \mid b_1),\ \ldots, & \\
(h_k \text{ :- } G_k \mid b_k)]\!]abenv &= \bigsqcup_{i=1}^{k} rgc[\![g,\ (h_i \text{ :- } G_i \mid b_i)]\!]abenv
\end{aligned}
$$

The three components of the algorithm, namely, abstract reduction, local fixpoint computation, and global fixpoint computation, are reflected in these equations. We first prove that the abstract reduction is correct.

Theorem 1

If S is a concrete state and S' is the state resulting from the single reduction of the goal g with respect to a clause h :- $G \mid B$, then $\alpha(S') \sqsubseteq AbsRed(g,\ (h \text{ :- } G \mid B), \alpha(S))$. □

Proof Outline: We prove the above theorem by cases. We have to consider the cases in which a variable becomes multiple threaded in a concrete reduction and show that the abstract reduction operation also makes these variables multiple threaded.

Case 1: A variable X occurs in state S before the reduction and was multiple threaded. Since the abstract reduction operator may only make a single threaded variable into multiple threaded variable but not vice versa, the variable is multiple threaded in $AbsRed(g,\ (h \text{ :- } G \mid B),\ \alpha(S))$. Thus $\alpha(S')[\![X]\!] \sqsubseteq AbsRed(g,\ (h \text{ :- } G \mid B),\ \alpha(S))[\![X]\!]$.

Case 2: A variable X occurs in state S before the reduction and was single threaded. The only way X can become multiple threaded after the reduction (in state S') is

by the tell unifications. This is because ask unifications and head–goal matching do not bind any goal variables. They pass the threadedness of the variables in the goal to the variables of the clause under consideration. There are three possible ways in which X may become multiple threaded due to tell unifications. We show that in each of these three cases, our initialization rules and the abstract unification algorithm safely conclude X to be multiple threaded.

- X became multiple threaded because of a tell unification $X = Y$ and because Y is multiple threaded in S, i.e., $\alpha(S)[\![Y]\!] = SM$. Since the abstract tell unification of $X = Y$ propagates the least upper bound of the threadedness of its arguments to both the arguments, (section 4.6) $\alpha(S')[\![X]\!] \sqsubseteq AbsRed(g,\ (h \text{ :- } G \mid B),\ \alpha(S))[\![X]\!]$.
- X became multiple threaded because of a tell unification $X = f(...Y_i...)$ and because Y_i is multiple threaded. If Y_i is non-local,[4] then either it must have been multiple threaded in the previous state S, i.e., $\alpha(S)[\![Y_i]\!] = SM$, or Y_i itself must have become multiple threaded (due to the reduction from state S to state S') because of another tell unification.

  If Y_i is local, then Y_i is multiple threaded either because it occurs more than twice in S' (in which case the function *Init* would have correctly classified Y_i as multiple threaded) or because of another tell unification equation $X' = f(...Y_i...)$ where X' is multiple threaded.

  In either case, function *Aunify* (section 4.6) correctly propagates the threadedness of Y_i to X, and thus $\alpha(S')[\![X]\!] \sqsubseteq AbsRed(g,\ (h \text{ :- } G \mid B),\ \alpha(S))[\![X]\!]$.
- X became multiple threaded because of a tell unification equation $Y = f(\ldots, X, \ldots)$ where Y occurs in state S and Y either was multiple threaded in state S or became multiple threaded in state S'. After unification, X refers to some subterm of Y. This is approximated by our abstract tell unification rule (section 4.6) for propagating threadedness of a structure to its components. Since Y is mulltiple threaded in state S', the abstract tell unification makes X multiple threaded. Thus
  $\alpha(S')[\![X]\!] \sqsubseteq AbsRed(g,\ (h \text{ :- } G \mid B),\ \alpha(S))[\![X]\!]$.

Case 3: X is a variable of the current clause, which implies X does not appear in state S but appears in state S'. Variable X can become multiple threaded one of the following ways.

[4] Those variables of a clause that do not appear in the head and the ask part of the guard nor in the RHS of a tell equation of the form $X = f(\ldots, Y, \ldots)$ where X is a variable that appears in the head or the ask part will be referred to as local variables of the clause. Non-local variables of the clause are just references to the terms or the sub-terms of the goal.

- Variable X is multiple threaded in state S' because it occurs more than twice in that state. X may occur more than twice in state S' because it occurs more than twice in the clause. If so, function *init* would classify X as multiple threaded and thus $\alpha(S')[\![X]\!] \sqsubseteq AbsRed(g,\ (h :- G \mid B),\ \alpha(S))[\![X]\!]$.
- Variable X is multiple threaded in state S' because it is bound to f(...,Y,...) in the current substitution θ and Y is mutiple-threaded. This situation may arise because of a tell unification equation $X = f(\ldots,Y,\ldots)$. Or it may arise because of another tell unification equation $Z = g(\ldots,X,\ldots)$ where Z is a multiple-threaded local variable. The abstract tell unification function would correctly classify X as *SM* in either case. It follows that $\alpha(S')[\![X]\!] \sqsubseteq AbsRed(g,\ (h :- G \mid B),\ \alpha(S))[\![X]\!]$.

We have thus shown that the abstract reduction safely approximates all possible cases in which a variable can become multiple threaded in a single reduction.

Lemma 1

The function *AbsRed* is monotonic. □

AbsRed is monotonic because it is a composition of monotonic functions *Aunify* and *Punify*.

Lemma 2

$\bigsqcup\{\alpha''(C) \mid C \in Comp$ is a computation of length k with intial state $S_0\} \sqsubseteq$ $\bigsqcup\{\alpha''(C) \mid C \in Comp$ is a computation of length $l \geq k$ with initial state $S_0\}$

□

From the definition of α'', it is obvious that $\alpha''(C) \sqsubseteq \alpha''(C')$ where C is a prefix of C'. Since every computation of length k is a prefix of some computation of length $l \geq k$, the above lemma holds.

Lemma 3

The local fixpoint computation gives a safe approximation of any state that could result from the reduction of the body goals of a clause. □

Proof Outline: The only way in which two body goals can interfere with each other is through common or shared variables by which they can propagate the multiple-threadedness of one to the other. Each body goal is solved abstractly with the current abstract substitution as the initial substitution, to gather the threadedness information of its variables. Since the reduction of goals may be interleaved, a goal a can be reduced

after some arbitrary k reductions of another goal b. By lemma 2, we know that the abstract environment after k reductions is less defined than the environment obtained by solving the goal completely. Given an environment that is a safe approximation of the current state, the least upper bound of the environments obtained by solving each goal individually, is also a safe approximation to the state at which any goal is reduced because it captures the worst case behavior of all other goals during reductions. In other words, the effect of any reduction of one goal on the other is taken care of by the least upper bound operation. The new environment if different from the initial environment, is used as the new initial environment for solving each body goal. This is formally achieved by computing the fixpoint of the equation:

$$abenv \;=\; \bigsqcup\{rgp[\![b_i, p_i]\!](abenv \sqcup abenv_0)\}$$

$abenv_0$ is the initial environment in which the goals are called and p_i is the procedure that defines the body goal b_i.

Lemma 4

The effect of a procedure is safely approximated by the global fixpoint computation. □

Proof Outline: Since the effect of each clause is safely approximated by the local fixpoint computation, the effect of a procedure is given by finding the effect of each clause and then taking the least upper bound. Given an environment which is a safe approximation to the state, the environment obtained by *rgp* safely approximates the state reached by the concrete execution of the procedure. This follows from the safety of abstract reduction (*AbsRed*), correctness of local fixpoint computation, and the monotonicity of the least upper bound operator. Since the procedures can be defined recursively, the global fixpoint is computed by successive approximation. The intial approximation of each predicate is an identity function (mapping an environment to itself). The ith approximation uses the $(i-1)^{th}$ approximation of the abstractions computed for each procedure.

Theorem 2

The analysis is sound. □

Proof: It follows from Theorem 1 and Lemmas 2, 3, and 4.

The termination of the analysis follows from the finiteness of the program (finite number of program variables), finiteness of the abstract domain, and the monotonicity of abstract reduction (lemma 1).