SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 23(4), 419–459 (APRIL 1993)

Generic Abstract Interpretation Algorithms for Prolog: Two Optimization Techniques and their Experimental Evaluation*

VINCENT ENGLEBERT, BAUDOUIN LE CHARLIER AND DIDIER ROLAND
University of Namur, 21 rue Grandgagnage, B-5000 Namur, Belgium

AND

PASCAL VAN HENTENRYCK†
Brown University, Box 1910, Providence, RI 02912, U.S.A.

SUMMARY

The efficient implementation of generic abstract interpretation algorithms for Prolog is reconsidered after References 1 and 2. Two new optimization techniques are proposed and applied to the original algorithm of Reference 1: dependency on clause prefixes and caching of operations. The first improvement avoids re-evaluating a clause prefix when no abstract value which it depends on has been updated. The second improvement consists of caching all operations on substitutions and reusing the results whenever possible. The algorithm and the two optimization techniques have been implemented in C (about 8000 lines of code each), tested on a large number of Prolog programs, and compared with the original implementation on an abstract domain containing modes, types and sharing. In conjunction with refinements of the domain algorithms, they produce an average reduction of more than 58 per cent in computation time. Extensive experimental results on the programs are given, including computation times, memory consumption, hit ratios for the caches, the number of operations performed, and the time distribution. As a main result, the improved algorithms exhibit the same efficiency as the specific tools of References 3 and 4, despite the fact that our abstract domain is more sophisticated and accurate. The abstract operations also take 90 per cent of the computation time, indicating that the overhead of the control is very limited. Results on a simpler domain are also given and show that even extremely basic domains can benefit from the optimizations. The general-purpose character of the optimizations is also discussed.

KEY WORDS: Abstract interpretation PROLOG Fixpoint algorithm Experimentation Languages Performance

1. INTRODUCTION

Abstract interpretation[5] is a general methodology to obtain, in a systematic way, tools to analyse programs statically (at compile time). The basic idea behind abstract interpretation is to approximate (usually undecidable) properties by using an abstract

* This work was done when Vincent Englebert and Didier Roland were visiting Brown University.
† Author to whom correspondence must be directed.

0038–0644/93/040419–41$25.50

Received 19 December 1991
Revised 3 December 1992

domain instead of the actual domain of computation. As a consequence, the program as a whole can be given an approximated meaning, hopefully capturing interesting properties while leaving out irrelevant details as much as possible.

Abstract interpretations of Prolog have attracted many researchers in recent years. The motivation behind these works stems from the need for optimization in Prolog compilers (given the very high level of these languages) and the large potential for optimization since the semantic features of logic languages make them more amenable to optimization. Mellish[6] was probably the first to define an abstract interpretation framework for Prolog motivated by early analysis tools for Prolog programs. Subsequently, many frameworks have been developed[6–13] and a variety of abstract domains have been proposed to cater for various program analysis tools involving modes,[14,15] types,[16] occur-check,[17] garbage collection,[18] static detection of parallelism[19] and program specialization,[20] to name a few.

However, relatively little attention has been devoted to generic abstract interpretation algorithms, although several authors have proposed (more or less precise) sketches of some algorithms.[6,8,12,21–23] Moreover, few experimental results are available to assess the practicability of the overall abstract interpretation approach for Prolog. One of the rare papers on the topic[3,4] describes the efficiency and accuracy of two analysis tools and shows that these tools are in fact practical both in terms of efficiency and in terms of usefulness of the results. However, it would be unreasonable to infer from References 3 and 4 the viability of generic abstract interpretation of Prolog programs, as the tools analysed were rather specialized and targeted towards specific applications.

Part of our research has been devoted to demonstrating the practicality of this area of research. Our starting point was the design of a generic abstract interpretation algorithm and its complexity analysis.[1] The algorithm, initially motivated by Reference 21, is a top-down algorithm focusing on relevant parts of the least fixpoint necessary to answer the user query. It is polynomial in the worst-case and linear in the sizes of the abstract domain and of the Prolog program in many interesting cases. The algorithm, together with its instantiation to a sophisticated abstract domain (derived from Reference 16) containing modes, types, sharing and aliasing, has been implemented in Pascal and run on a large number of programs. The experimental results have shown the practical value of the algorithm and have indicated that abstract interpretation can be competitive with specialized algorithms such as those reported in References 3 and 4.

In this paper, we reconsider the problem of implementing efficiently generic abstract interpretation algorithms for Prolog. We propose two new optimization techniques to the original algorithm: dependency on clause prefixes and caching of operations. The first improvement avoids re-evaluating a clause prefix when no abstract value on which it depends has been updated. As a consequence, it generalizes the dependency graph proposed in Reference 1 and avoids re-evaluating clauses and part of clauses unnecessarily. The second improvement consists of caching all operations on substitutions and reusing the results whenever possible. Garbage collection is performed on substitutions which are no longer in use. This improvement subsumes (in some sense) the first improvement because most of the computation time is consumed by the abstract operations. Therefore, executing clause prefixes using cached operations only requires a negligible amount of time. Caching also allows the sharing of results between independent computations, albeit at a higher

cost in memory. Moreover, in the case of finite abstract domains, it implies that the number of operations performed by the algorithm is bounded by the number of program points times the number of times their associated operations can be executed (in the worst case the square of the abstract domain size). The two optimization techniques are, in fact, general-purpose and can be used for other languages and approaches as well.

The algorithms (the original algorithm together with each improvement) have been implemented in C (about 8000 lines of code each). They have been tested on a large number of Prolog programs and compared with the original implementation. In conjunction with refinements of the abstract operations implementation, they produce an average reduction of more than 50 per cent in computation time. Extensive experimental results on the programs are given, including computation times, memory consumption, hit ratios for the caches, the number of operations, and the time distribution. As a main result, the improved algorithms exhibit the same efficiency as the specific tools of Warren, Debray and Hermenegildo,[3,4] despite the fact that our abstract domain (including types and sharing) is more sophisticated and accurate. The abstract operations also take 90 per cent of the computation time, indicating that the overhead of the control is small compared to the abstract operations. This means that the algorithm is close to optimality for computing the given abstract semantics.* Results on a simpler domain are also given and show that even extremely basic domains can benefit from the optimizations.

The rest of the paper is organized in the following way. Section 2 gives an overview of the original abstract interpretation algorithm and of the main abstract domain used in our experiments. Section 3 presents the clause prefix improvement and Section 4 presents the caching improvement. Section 5 is devoted to the experimental results of the algorithms. Section 6 discusses how to apply the optimizations to other contexts. Section 7 contains the conclusions of this research and directions for further work.

2. PRELIMINARIES

2.1. Normalized logic programs

Our original generic abstract interpretation algorithm (as well as the underlying abstract semantics) is defined on normalized logic programs. The use of normalized logic programs, suggested first in Reference 21, greatly simplifies the semantics, the algorithm and its implementation. Figure 1 presents a normalized version of the classical list concatenation program as generated by our implementation.

Normalized programs are built from an ordered set of variables $\{X_1, \ldots, X_n, \ldots\}$. The variables are called *program variables*. A normalized program is a set of clauses

$$H : - B_1, \ldots, B_n$$

where H is called the head, and $B_1, \ldots, B_n$ the body. If a clause contains m variables, these variables are necessarily $X_1, \ldots, X_m$. Moreover, the head of the clause is an

* Of course, other (less demanding) abstract semantics can be imagined.

```
append( X1 , X2 , X3 ) :-
   X1 = [],
   X3 = X2.
append( X1 , X2 , X3 ) :-
   X1 = [ X4 | X5 ],
   X3 = [ X4 | X6 ],
   append( X5 , X2 , X6 ).
```

Figure 1. An example of a normalized program: append/3

atom $p(X_1, \ldots, X_i)$ where p is a predicate symbol of arity i. The subgoals in the body of the clause are of the form:

(a) $q(X_{i_1}, \ldots, X_{i_m})$ where $i_1, \ldots, i_m$ are distinct indices
(b) $X_{i_1} = X_{i_2}$ with $i_1 \neq i_2$
(c) $X_{i_1} = f(X_{i_2}, \ldots, X_{i_m})$ where f is a function of arity $m-1$ and $i_1, \ldots, i_m$ are distinct indices.

The first form is called a procedure call. The second and third forms, called built-ins, enable unification to be achieved. Additional built-in predicates (e.g. arithmetic primitives) can be accommodated in the framework but are not discussed in the paper for simplicity. It is not a difficult matter to translate any Prolog program into its normalized version. The advantage of normalized programs comes from the fact that an (input or output) substitution for a goal p/n is always expressed in terms of variables $X_1, \ldots, X_n$. This greatly simplifies all the traditional problems encountered with renaming.

2.2. Operations on substitutions

To define the semantics, we will denote by UD the underlying domain of the program, i.e. the set of pairs (β_{in}, p) where p is a predicate symbol of arity n and β_{in} is an astract substitution on variables $\{X_1, \ldots, X_n\}$. The abstract substitutions on variables $D = \{X_1, \ldots, X_n\}$ are elements of a complete partial order (c.p.o.) $(AS_D, \leq)$. There is one c.p.o. for each set of variables. We will also need a number of operations on substitutions. These operations are defined informally here. Reference 1 contains a precise specification of the abstract operations in terms of the concretization function (see also Reference 24). In the following, abstract substitutions will be denoted by greek letters. We also call an *abstract couple* the association of an abstract substitution and a predicate symbol (e.g. (β, p)). We use *sat*, possibly subscripted or superscripted, to represent sets of abstract tuples. Note also that abstract substitutions represent sets of concrete substitutions.

The operations are as follows:

1. UNION $\{\beta_1, \ldots, \beta_n\}$ where $\beta_1, \ldots, \beta_n$ are abstract substitutions from the same c.p.o.: this operation returns an abstract substitution representing all the substitutions

satisfying at least one β_i. It is used to compute the output of a procedure given the outputs for its clauses.

2. AI_VAR (β) where β is an abstract substitution on $\{X_1, X_2\}$: this operation returns the abstract substitution obtained from β by unifying variables X_1, X_2. It is used for goals of the form $X_i = X_j$ in normalized programs.
3. AI_FUNC (β, f) where β is an abstract substitution on $\{X_1, \ldots, X_n\}$ and f is a function symbol of arity $n - 1$: this operation returns the abstract substitution obtained from β by unifying X_1 and $f(X_2, \ldots, X_n)$. It is used for goals $X_{i_1} = f(X_{i_2}, \ldots, X_{i_n})$ in normalized programs.
4. EXTC (c, β) where β is an abstract substitution on $\{X_1, \ldots, X_n\}$ and c is a clause containing variables $\{X_1, \ldots, X_m\}$ ($m \geq n$): this operation returns the abstract substitution obtained by extending β to accommodate the new free variables of the clause. It is used at the entry of a clause to include the variables in the body not present in the head. In logical terms, this operation, together with the next operation, achieves the role of the existential quantifier.
5. RESTRC (c, β) where β is an abstract substitution on the clause variables $\{X_1, \ldots, X_m\}$ and $\{X_1, \ldots, X_n\}$ are the head variables of clause c($n \leq m$): this operation returns the abstract substitution obtained by projecting β on variables $\{X_1, \ldots, X_n\}$. It is used at the exit of a clause to restrict the substitution to the head variables only.
6. RESTRG (g, β) where β is an abstract substitution on $D = \{X_1, \ldots, X_n\}$, and g is a goal $p(X_{i_1}, \ldots, X_{i_m})$ (or $X_{i_1} = X_{i_2}$ or $X_{i_1} = f(X_{i_2}, \ldots, X_{i_m})$): this operation returns the abstract substitution obtained by

   (a) projecting β on $\{X_{i_1}, \ldots, X_{i_m}\}$ obtaining β'
   (b) expressing β' in terms of $\{X_1, \ldots, X_m\}$ by mapping X_{i_k} to X_k.

   It is used before the execution of a goal in the body of a clause. The resulting substitution is expressed in terms of $\{X_1, \ldots, X_m\}$, i.e. in the same way as the input and output substitutions of p in the abstract domain.
7. EXTG (g, β, β') where β is an abstract substitution on $D = \{X_1, \ldots, X_n\}$, the variables of the clause where g appears, g is a goal $p(X_{i_1}, \ldots, X_{i_m})$ (or $X_{i_1} = X_{i_2}$ or $X_{i_1} = f(X_{i_2}, \ldots, X_{i_m})$) with $\{X_{i_1}, \ldots, X_{i_m}\} \subseteq$ D and β' is an abstract substitution on $\{X_1, \ldots, X_m\}$ representing the result of $p(X_1, \ldots, X_m)$ β'' where β'' = RESTRG (g, β): this operation returns the abstract substitution obtained by extending β to take into account the result β' of the goal g. It is used after the execution of a goal to propagate the results of the goal on the substitution for all variables of the clause.
8. EXTEND (β, p, sat), given an abstract substitution β, a predicate symbol p, and a set of abstract tuples sat which does not contain (β, p) in its domain, returns a set of abstract tuples sat' containing (β, p) in its domain. Moreover the value $sat'(\beta, p)$ is defined as the lub (i.e. the least upper bound) of all $sat(\beta', p)$ such that $\beta' \leq \beta$.
9. ADJUST (β, p, β', sat) where β' represents a new result computed for the pair (β, p), returns a sat' which is sat updated with this new result. More precisely, the value of $sat'(\beta'', p)$ for all $\beta'' \geq \beta$ is equal to $lub\{\beta', sat(\beta'', p)\}$, and all other values are left unchanged. In the algorithm, we use a slightly more general version of ADJUST which, in addition to the new set of abstract tuples, returns the set of pairs (β, p) whose values have been updated.

2.3. Goal dependencies

We now have all operations necessary to design the algorithm. However, one of our main concerns in the design of the algorithm has been the detection of redundant computations. Redundant computations may occur in a variety of situations. For instance, the value of a pair (α, q) may have reached its definitive value (the value of (α, q) is in the least fixpoint) and hence subsequent considerations of (α, q) should only look up its value instead of starting a subcomputation. Another important case (especially in logic programming) is that of mutually recursive programs. For these programs, we would like the algorithm to reconsider a pair (α, q) only when some elements which it is depending upon have been updated. In other words, keeping track of the goal dependencies may substantially improve the efficiency on some classes of programs.

Our algorithm includes specific data structures to maintain goal dependencies. We only introduce the basic notions here.

Definition 1

A dependency graph is a set of tuples of the form $\langle(\beta, p), lt\rangle$ where lt is a set $\{(\alpha_1, q_1), \ldots, (\alpha_n, q_n)\}$ $(n \geq 0)$ such that, for each (β, p), there exists at most one lt such that $\langle(\beta, p), lt\rangle \in dp$.

We denote by $dp(\beta, p)$ the set lt such that $\langle(\beta, p), \mathrm{lt}\rangle \in dp$ if it exists. We also denote by $dom(dp)$ the set of all (β, p) such that $\langle(\beta, p), lt\rangle \in dp$ and by $codom(dp)$ the set of all (α, q) such that there exists a tuple $\langle(\beta, p), lt\rangle \in dp$ satisfying $(\alpha, q) \in lt$.

The basic intuition here is that $dp(\beta, p)$ represents at some point the set of pairs upon which (β, p) depends directly . To be complete, we need to define the transitive closure of the dependencies.

Definition 2

Let dp be a dependency graph and assume that $(\beta, p) \in dom(dp)$. The set $trans\_dp(\beta, p, dp)$ is the smallest subset of $codom(dp)$ closed by the following two rules:

1. If $(\alpha, q) \in dp(\beta, p)$ then $(\alpha, q) \in trans\_dp(\beta, p, dp)$.
2. If $(\alpha, q) \in dp(\beta, p)$, $(\alpha, q) \in dom(dp)$ and $(\alpha', q') \in trans\_dp(\alpha, q, dp)$ then $(\alpha', q') \in trans\_dp(\beta, p, dp)$.

Now $trans\_dp(\beta, p, dp)$ represents all the pairs which, if updated, would require reconsidering (β, p). (β, p) will not be reconsidered unless one of these pairs is updated.

We are now in position to specify the last three operations needed to present the algorithm:

1. REMOVE_DP (*modified*, *dp*), where *modified* is a list of pairs $(\alpha_1, p_1), \ldots, (\alpha_n, p_n)$ and dp is a dependency graph, removes from the dependency graph all elements $\langle(\alpha, q), lt\rangle$ for which there is an $(\alpha_i, p_i) \in trans\_dp(\alpha, q, dp)$.
2. EXT_DP (β, d, dp) inserts an element $\langle(\beta, p), \varnothing\rangle$ in dp.
3. ADD_DP $(\beta, p, \alpha, q, dp)$ simply updates dp to include the dependency of (β, p) w.r.t. (α, q) (after its execution $(\alpha, q) \in dp(\beta, p)$).

The algorithm makes sure that the elements (β, p) that need to be reconsidered are such that $(\beta, p) \not\in dom(dp)$.

2.4. The generic abstract interpretation algorithm

We are now in a position to present our generic abstract interpretation algorithm. The algorithm is composed of three procedures, and is shown in Figure 2.

The top-level procedure is the procedure solve which, given an input substitution β_{in} and a predicate symbol p, returns the set of abstract tuples *sat* containing $(\beta_{in}, p, \beta_{out})$ belonging to the least fixpoint and the final dependency graph. Given the results, it is straightforward to compute the set of pairs (α, q) used by (β, p), their values at the fixpoint, as well as the abstract substitutions at any program point.

The procedure solve_call receives as inputs an abstract substitution β_{in}, its associated predicate symbol, a set *suspended* of pairs (α, q), *sat*, and a dependency graph *dp*. The set *suspended* contains all pairs (α, q) for which a subcomputation has been initiated and not yet been completed. The procedure is responsible for considering (or reconsidering) the pair (β_{in}, p) and updating *sat* and *dp* accordingly. The core of the procedure is only executed when (β_{in}, p) is not suspended and not in the domain of the dependency graph. If (β_{in}, p) is suspended, no subcomputation should be initiated. If (β_{in}, p) is in the domain of the dependency graph, it means that none of the elements which it is depending upon have been updated. Otherwise, a new computation with (β_{in}, p) is initiated. This subcomputation may extend *sat* if it is the first time β_{in} is considered. The core of the procedure is a repeat loop which computes the best approximation of (β_{in}, p) given the elements of the suspended set. Local convergence is attained when (β_{in}, p) is in the domain of the dependency graph. One iteration of the loop amounts to executing each of the clauses defining p and computing the union of the results. If the result produced is greater or not comparable to the current value of (β_{in}, p), then the set of abstract tuples is updated and the dependency graph is also adjusted accordingly. Note that the call to the clauses is done with an extended suspended set since a subcomputation has been started with (β_{in}, p). Note also that, before executing the clauses, the dependency graph has been updated to include (β_{in}, p) (which is guaranteed not to be in the domain of the dependency graph at that point). (β_{in}, p) can be removed from the domain of the dependency graph during the execution of the loop if a pair which it is depending upon is updated.

The procedure solve_clause executes a single clause for an input pair and returns an abstract substitution representing the execution of the clause on that pair. It begins by extending the substitution with the variables of the clause, then executes the body of the clause, and terminates by restricting the substitution to the variables of the head. The execution of a goal requires three steps:

(a) restriction of the current substitution β_{ext} to its variables, giving β_{aux}
(b) execution of the goal itself on β_{aux} producing β_{int}
(c) propagation of its result β_{int} on β_{ext}.

If the goal is concerned with unification, the operations AI_VAR and AI_FUNC are used. Otherwise, procedure solve_call is called and the result is looked up in *sat*. Moreover, if (β_{in}, p) is in the domain of the dependency graph, it is necessary to add a new dependency. Otherwise, it means that (β_{in}, p) needs to be reconsidered anyway and no dependency must be recorded.

```
procedure solve(in β_in, p; out sat, dp)
begin
   sat := ∅;
   dp := ∅;
   solve_call(β_in, p, ∅, sat, dp)
end

procedure solve_call(in β_in, p, suspended; inout sat, dp)
begin
   if (β_in, p) ∉ (dom(dp) ∪ suspended) then
   begin
      if (β_in, p) ∉ dom(sat) then
         sat := EXTEND(β_in, p, sat);
      repeat
         β_out := ⊥;
         EXT_DP(β_in, p, dp);
         for i := 1 to m with c_1, ..., c_m clauses-of p do
         begin
            solve_clause(β_in, p, c_i, suspended ∪ {(β_in, p)}, β_aux, sat, dp);
            β_out := UNION(β_out, β_aux)
         end;
         (sat, modified) := ADJUST(β_in, p, β_out, sat);
         REMOVE_DP(modified, dp)
      until (β_in, p) ∈ dom(dp)
   end
end

procedure solve_clause(in β_in, p, c, suspended; out β_out; inout sat, dp)
begin
   β_ext := EXTC(c, β_in);
   for i := 1 to m with b_1, ..., b_m body-of c do
   begin
      β_aux := RESTRG(b_i, β_ext);
      switch (b_i) of
      case X_j = X_k:
         β_int := AI_VAR(β_aux)
      case X_j = f(...):
         β_int := AI_FUNC(β_aux, f)
      case q(...):
         solve_call(β_aux, q, suspended, sat, dp);
         β_int := sat(β_aux, q);
         if (β_in, p) ∈ dom(dp) then
            ADD_DP(β_in, p, β_aux, q, dp)
      end;
      β_ext := EXTG(b_i, β_ext, β_int)
   end;
   β_out := RESTRC(c, β_ext)
end
```

Figure 2. The generic abstract intepretation algorithm

2.5. Widening

In the case of infinite domains, the above algorithm may not terminate. To guarantee termination, the algorithm is extended with a widening technique.[5] The modification occurs at the beginning of the procedure solve_goal, where a new instruction

$$\beta_{in} := \text{WIDEN}\ (\beta_{in},\ p,\ \text{suspended});$$

must be inserted. The set suspended should also be viewed as a stack in this version, since the order of the elements is important for widening. The function WIDEN can be defined as follows:

```
function WIDEN(in β_new,p,suspended); AS
begin
  β_old := GET_PREVIOUS(p,suspended);
  WIDEN := β_old ∇ β_new
end;
```

The function GET_PREVIOUS returns the substitution β of the last pair (β, p) inserted in the stack suspended or $\perp$ if there is no such pair. The symbol ∇ is a widening operator.

Definition 3[5]

Let A be a poset. A widening operation on A is a function: $\nabla: A \times A \rightarrow A$ satisfying

1. For each infinite sequence $x_0, x_1, \ldots, x_i, \ldots$ $(x_k \in A)$, the sequence $y_0, y_1, \ldots, y_i, \ldots$ is increasing and stationary, where $y_0 = x_0$ & $y_{i+1} = y_i \nabla x_{i+1}$ $(i \geq 0)$.
2. $\forall x, y \in A$: $x, y \leq x \nabla y$.

The widening operator can be domain-independent or domain-dependent. In the experimental results described later on, we use a simple domain-independent operator: the least upper bound of the two abstract substitutions. Other, domain-dependent, widening operators have been investigated, and preliminary results are described in Reference 25.

It is important to realize that the algorithm, extended with the above widening technique, may not compute a fixpoint of the abstract transformation τ but may lead to a postfixpoint g (i.e. $g \geq \tau(g)$). This is of course perfectly acceptable in the context of abstract interpretation, since the objective is to compute an upper-approximation of the program meaning, and our framework guarantees that any postfixpoint is such an approximation.[26]

As the result of the algorithm is, in general, a postfixpoint, *narrowing* techniques[5] could be used as well. We did not investigate this point in depth, since narrowing does not seem very useful for the particular abstract domain. This is mainly because operation EXTG achieves an effect very similar to narrowing when execution returns from a procedure call which used widening.*

* In this case, EXTG implements the so-called *backward unification.*

Our use of widening is useful to limit, in an 'almost intelligent' way, the number of abstract inputs to be considered. On most of our benchmark programs, widening achieves the right generalizations and gives better results than *a priori* restrictions on the granularity of the analysis.

2.6. Overview of the abstract domain

In this section, we give a brief overview of the abstract domain used in our experiments. The abstract domain contains patterns (i.e. for each subterm, the main fuctor and a reference to its arguments are stored), sharing, same-value and mode components. The domain is more sophisticated than the sharing domains of, for instance, References 19 and 27 and than the mode domains of, for instance, References 3 and 4. It is best viewed as an abstraction of the domain of Bruynooghe and Janssens,[16] where a pattern component has been added. The domain is fully described in Reference 28 which also contains the proofs of monotonicity and consistency. Note also that the presentation in Reference 28 is generic and can be instantiated to a variety of applications. The presentation here is an instantiation to modes.

The key concept in the representation of the substitutions in this domain is the notion of subterm. Given a substitution on a set of variables, an abstract substitution will associate the following information with each subterm:

(a) its *mode* (e.g. *Gro*, *Var*, *Ngv* (i.e. neither ground nor variable))
(b) its *pattern* which specifies the main functor as well as the subterms which are its arguments
(c) its possible *sharing* with other subterms.

Note that the *pattern* is optional. If it is omitted, the pattern is said to be undefined. In addition to the above information, each variable in the domain of the substitution is associated with one of the subterms. Note that this information enables us to express that two arguments have the same value (and hence that two variables are bound together). To identify the subterms in an unambiguous way, an index is associated with each of them. If there are n subterms, we make use of indices $1, \ldots, n$. For instance, the substitution

$$\{X_1 \leftarrow t * v, X_2 \leftarrow v, X_3 \leftarrow Y_1\backslash[\,]\}$$

will have seven subterms. The association of indices with them could be for instance

$$\{(1, t * v), (2, t), (3, v), (4, v), (5, Y_1\backslash[\,]), (6, Y_1), (7, [\,])\}$$

As mentioned previously, each index is associated with a mode taken from

$$\{\bot, Gro, Var, Ngv, Novar, Gv, Nogro, Any\}$$

In the above example, we have the following associations:

$$\{(1, \mathrm{Gro}), (2, \mathrm{Gro}), (3, \mathrm{Gro}), (4, \mathrm{Gro}), (5, \mathrm{Ngv}), (6, \mathrm{Var}), (7, \mathrm{Gro})\}$$

The pattern components (possibly) assigns to an index an expression $f(i_1, \ldots, i_n)$

where f is a function symbol of arity n and $i_1, \ldots, i_n$ are indices. In our example, the pattern component will make the following associations:

$$\{(1, 2 * 3), (2, t), (3, v), (4, v), (5, 6\backslash 7), (7, [\,])\}$$

Finally, the sharing component specifies which indices, not associated with a pattern, may possibly share variables. We only restrict our attention to indices with no pattern, since the other patterns already express some sharing information and we do not want to introduce inconsistencies between the components. The actual sharing relation can be derived from these two components. In our particular example, the only sharing is the couple (6, 6) which expresses that the variable Y_1 shares a variable with itself.

As the above representation may be difficult to visualize, we make use of a more appealing representation in the following. For instance, a predicate *factorize* (X_1, X_2, X_3) instantiated by the above substitution will be represented as

$$\begin{array}{ll} \textit{factorize} & (\mathit{Gro}(1) : * \ (\ \mathit{Gro}(2) : t,\ \mathit{Gro}(3) : v), \\ & \mathit{Gro}(4) : v, \\ & \mathit{Ngv}(5) : \backslash(\mathit{Var}(6),\ \mathit{Gro}(7) : [\,])) \end{array}$$

together with the sharing information {(6, 6)}. In the above representation, each argument is associated with a mode, with an index (between parenthesis), and with an abstract subterm after the colon. The subterm uses the same representation.

The execution of the algorithm on this domain for the append program is shown in Figure 3.

3. CLAUSE PREFIX DEPENDENCY

We now turn to the optimization techniques of the original algorithm. We start with the clause prefix improvement.

3.1. Motivation

To motivate the first improvement, consider the execution of the above algorithm on the append/3 program as depicted in Figure 3.

The first iteration considers both clauses. Only the first clause produces a result, since the second clause calls itself recursively with the same abstract substitution. The current approximation is the result of the first clause. The second iteration considers once again both clauses and updates the approximation to its final value. The third iteration does not produce any change to *sat* and hence the algorithm terminates. The key points to notice here are as follows:

(a) the first clause should not be considered more than once as it does not depend recursively on the call
(b) the second clause should only be reconsidered from the point where new information may be produced, i.e. CALL solve_goal
(c) the above two comments would remain valid even if the clause would first call a goal not calling recursively append/3 with the same abstract substitution.

The algorithm with the clause prefix improvement produces exactly the expected result,

```
CALL solve_clause 1
  EXIT EXTC (Var(1),Var(2),Gro(3))  ps: -(1,1)(2,2)"
  CALL UNIF-FUN (Var(1),Var(2),Gro(3))  ps: -(1,1)(2,2)"
  EXIT UNIF-FUN (Gro(1):[],Var(2),Gro(3))  ps: -(2,2)"
  CALL UNIF-VAR (Gro(1):[],Var(2),Gro(3))  ps: -(2,2)"
  EXIT UNIF-VAR (Gro(1):[],Gro(2),Gro(2))
  EXIT RESTRC (Gro(1):[],Gro(2),Gro(2))
  EXIT UNION (Gro(1):[],Gro(2),Gro(2))
EXIT solve_clause 1
CALL solve_clause 2
  EXIT EXTC (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6))  ps: -(1,1)(2,2)(4,4)(5,5)(6,6)"
  CALL UNIF-FUN (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6))  ps: -(1,1)(2,2)(4,4)(5,5)(6,6)"
  EXIT UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6)) ps: -(2,2)(3,3)(4,4)(6,6)"
  CALL UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6))  ps: -(2,2)(3,3)(4,4)(6,6)"
  EXIT UNIF-FUN (Ngv(1):.(Gro(2),Var(3)),Var(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Var(3),Gro(6))  ps: -(3,3)(4,4)"
  CALL solve_goal append(Var(1),Var(2),Gro(3))  ps: -(1,1)(2,2)"
  EXIT solve_goal append bottom
  EXIT EXTG bottom
  EXIT RESTRC bottom
  EXIT UNION (Gro(1):[],Gro(2),Gro(2))
EXIT solve_clause 2
ADJUST
CALL solve_clause 1
  EXIT EXTC (Var(1),Var(2),Gro(3))  ps: -(1,1)(2,2)"
  CALL UNIF-FUN (Var(1),Var(2),Gro(3))  ps: -(1,1)(2,2)"
  EXIT UNIF-FUN (Gro(1):[],Var(2),Gro(3))  ps: -(2,2)"
  CALL UNIF-VAR (Gro(1):[],Var(2),Gro(3))  ps: -(2,2)"
  EXIT UNIF-VAR (Gro(1):[],Gro(2),Gro(2))
  EXIT RESTRC (Gro(1):[],Gro(2),Gro(2))
  EXIT UNION (Gro(1):[],Gro(2),Gro(2))
EXIT solve_clause 1
CALL solve_clause 2
  EXIT EXTC (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6))  ps: -(1,1)(2,2)(4,4)(5,5)(6,6)"
  CALL UNIF-FUN (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6))  ps: -(1,1)(2,2)(4,4)(5,5)(6,6)"
  EXIT UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6))  ps: -(2,2)(3,3)(4,4)(6,6)"
  CALL UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6))  ps: -(2,2)(3,3)(4,4)(6,6)"
  EXIT UNIF-FUN (Ngv(1):.(Gro(2),Var(3)),Var(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Var(3),Gro(6))  ps: -(3,3)(4,4)"
  CALL solve_goal append(Var(1),Var(2),Gro(3))  ps: -(1,1)(2,2)"
  EXIT solve_goal append(Gro(1):[],Gro(2),Gro(2))
  EXIT EXTG (Gro(1):.(Gro(2),Gro(3):[]),Gro(4),Gro(5):.(Gro(2),Gro(4)),Gro(2),Gro(3):[],Gro(4))
  EXIT RESTRC (Gro(1):.(Gro(2),Gro(3):[]),Gro(4),Gro(5):.(Gro(2),Gro(4)))
  EXIT UNION (Gro(1),Gro(2),Gro(3))
EXIT solve_clause 2
ADJUST
CALL solve_clause 1
  EXIT EXTC (Var(1),Var(2),Gro(3))  ps: -(1,1)(2,2)"
  CALL UNIF-FUN (Var(1),Var(2),Gro(3))  ps: -(1,1)(2,2)"
  EXIT UNIF-FUN (Gro(1):[],Var(2),Gro(3))  ps: -(2,2)"
  CALL UNIF-VAR (Gro(1):[],Var(2),Gro(3))  ps: -(2,2)"
  EXIT UNIF-VAR (Gro(1):[],Gro(2),Gro(2))
  EXIT RESTRC (Gro(1):[],Gro(2),Gro(2))
  EXIT UNION (Gro(1):[],Gro(2),Gro(2))
EXIT solve_clause 1
CALL solve_clause 2
  EXIT EXTC (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6))  ps: -(1,1)(2,2)(4,4)(5,5)(6,6)"
  CALL UNIF-FUN (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6))  ps: -(1,1)(2,2)(4,4)(5,5)(6,6)"
  EXIT UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6))  ps: -(2,2)(3,3)(4,4)(6,6)"
  CALL UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6))  ps: -(2,2)(3,3)(4,4)(6,6)"
  EXIT UNIF-FUN (Ngv(1):.(Gro(2),Var(3)),Var(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Var(3),Gro(6))  ps: -(3,3)(4,4)"
  CALL solve_goal append(Var(1),Var(2),Gro(3))  ps: -(1,1)(2,2)"
  EXIT solve_goal append(Gro(1),Gro(2),Gro(3))
  EXIT EXTG (Gro(1):.(Gro(2),Gro(3)),Gro(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Gro(3),Gro(6))
  EXIT RESTRC (Gro(1):.(Gro(2),Gro(3)),Gro(4),Gro(5):.(Gro(2),Gro(6)))
  EXIT UNION (Gro(1),Gro(2),Gro(3))
EXIT solve_clause 2
```

Figure 3. The original algorithm on append/3

as depicted in Figure 4. Only the second clause is considered for the second and third iterations and re-execution starts only with the recursive call. The EXIT PREFIX line simply shows the substitution at this stage of the clause. All the operations performed by the algorithm are now strictly necessary.

More generally, the clause prefix improvement amounts to reconsidering only those clauses for which an element they depend upon has been updated. Moreover, execution in a clause restarts from the first goal for which an element it depends upon has been updated. The improvement thus avoids reconsidering any prefix of a clause which is known to give exactly the same result as its previous execution. Finally, execution of a goal restarts from the first clause for which an element it depends upon has been updated.

Note that the prefix optimization is independent of the program normalization process. Our original algorithm, adapted to work on the source programs instead of on normalized programs, would still need to perform the head unifications during subsequent iterations; these unifications are avoided in the prefix algorithm. In addition, the optimization does much more than simply avoiding the head unifications. It may entirely skip clauses and prefixes of clauses containing procedure calls. As a consequence, it avoids many costly operations, such as UNION, RESTRG and EXTG.

```
CALL solve'clause 1
  EXIT EXTC  (Var(1),Var(2),Gro(3))   ps: -(1,1)(2,2)"
  CALL UNIF-FUN  (Var(1),Var(2),Gro(3))   ps: -(1,1)(2,2)"
  EXIT UNIF-FUN  (Gro(1):[],Var(2),Gro(3))   ps: -(2,2)"
  CALL UNIF-VAR  (Gro(1):[],Var(2),Gro(3))   ps: -(2,2)"
  EXIT UNIF-VAR  (Gro(1):[],Gro(2),Gro(2))
  EXIT RESTRC  (Gro(1):[],Gro(2),Gro(2))
  EXIT UNION  (Gro(1):[],Gro(2),Gro(2))
EXIT solve'clause 1
CALL solve'clause 2
  EXIT EXTC  (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6))   ps: -(1,1)(2,2)(4,4)(5,5)(6,6)"
  CALL UNIF-FUN  (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6))   ps: -(1,1)(2,2)(4,4)(5,5)(6,6)"
  EXIT UNIF-FUN  (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6))   ps: -(2,2)(3,3)(4,4)(6,6)"
  CALL UNIF-FUN  (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6))   ps: -(2,2)(3,3)(4,4)(6,6)"
  EXIT UNIF-FUN  (Ngv(1):.(Gro(2),Var(3)),Var(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Var(3),Gro(6))   ps: -(3,3)(4,4)'
  CALL solve'goal  append (Var(1),Var(2),Gro(3))   ps: -(1,1)(2,2)"
  EXIT solve'goal  append  bottom
  EXIT EXTG   bottom
  EXIT RESTRC   bottom
  EXIT UNION  (Gro(1):[],Gro(2),Gro(2))
EXIT solve'clause 2
ADJUST
CALL solve'clause 2
  EXIT PREFIX  (Ngv(1):.(Gro(2),Var(3)),Var(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Var(3),Gro(6))   ps: -(3,3)(4,4)"
  CALL solve'goal  append (Var(1),Var(2),Gro(3))   ps: -(1,1)(2,2)"
  EXIT solve'goal  append (Gro(1):[],Gro(2),Gro(2))
  EXIT EXTG  (Gro(1):.(Gro(2),Gro(3):[]),Gro(4),Gro(5):.(Gro(2),Gro(4)),Gro(2),Gro(3):[],Gro(4))
  EXIT RESTRC  (Gro(1):.(Gro(2),Gro(3):[]),Gro(4),Gro(5):.(Gro(2),Gro(4)))
  EXIT UNION  (Gro(1),Gro(2),Gro(3))
EXIT solve'clause 2
ADJUST
CALL solve'clause 2
  EXIT PREFIX  (Ngv(1):.(Gro(2),Var(3)),Var(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Var(3),Gro(6))   ps: -(3,3)(4,4)"
  CALL solve'goal  append (Var(1),Var(2),Gro(3))   ps: -(1,1)(2,2)"
  EXIT solve'goal  append (Gro(1),Gro(2),Gro(3))
  EXIT EXTG  (Gro(1):.(Gro(2),Gro(3)),Gro(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Gro(3),Gro(6))
  EXIT RESTRC  (Gro(1):.(Gro(2),Gro(3)),Gro(4),Gro(5):.(Gro(2),Gro(6)))
  EXIT UNION  (Gro(1),Gro(2),Gro(3))
EXIT solve'clause 2
```

Figure 4. The clause prefix algorithm on append/3

3.2. Formalization

The key idea behind the clause prefix improvement is to extend the dependency graph to clauses and clause prefixes. Since only procedure calls can produce different results from one execution to another, we only consider clause prefixes ending at atoms.

Definition 4

Let c be a normalized clause and $g_1, \ldots, g_m$ the successive goals in the body of c. Prefix i of clause c, say $c[i]$, is simply $g_1, \ldots, g_i$ $(0 \leq i \leq m)$. The position (an integer) of the last goal of $c[i]$ in clause c will be denoted by $last(c[i])$. To ease the presentation, we take the convention that prefix 0, say $c[0]$, is the empty prefix and $last(c[0])$ is the position of the first goal in c.

Note that the convention for prefix 0 allows us to have a uniform definition for the procedure solve_clause, which does not have to distinguish between the first and subsequent execution of a clause.

Definition 5

A dependency graph is a set of tuples of the form $\langle(\beta, e), lt\rangle$ where e is a goal, a clause or a clause prefix and lt is a set $\{(\alpha_1, q_1), \ldots, (\alpha_n, q_n)\}$ $(n \geq 0)$ such that, for each (β, e), there exists at most one lt such that $\langle(\beta, e), lt\rangle \in dp$.

All the definitions given previously can be generalized in a staightforward way to deal with clauses and clause prefixes. Several new operations and notations can now be defined to simplify the presentation of the algorithm.

The operation G_ADD_DP $(\beta, p, c, i, \alpha, q, dp)$ is a generalization of ADD_DP which takes into account clauses and clause prefixes. Informally speaking, this operation updates the dependency graph for a goal p, the clause c in which the goal appears, and the relevant clause prefix. We denote by $nbgoal(c)$ the number of procedure calls in a clause c. The operation is defined as follows:

```
procedure G_ADD_DP (in β, p, c, i, α, q, inout dp)
begin
   ADD_DP (β, p, α, q, dp);
   ADD_DP (β, c, α, q, dp);
   ADD_DP (β, c [i], α, q, dp)
end
```

The operation EXT_DP is replaced by two operations, G_EXT_DP and C_EXT_DP. The first operation updates the dependency for goals and clauses, whereas the second operation updates the dependency graph for prefixes. They are defined as follows:

```
procedure G_EXT_DP (in β, p, inout dp)
begin
   EXT_DP (β, p, dp);
```

```
  for i : = 1 to m with c1, ..., cm clauses-of p do
    EXT_DP (β, ci, p);
end

procedure C_EXT_DP (in β, p, c, inout dp)
begin
  for j := 0 to nbgoal(c) do
    EXT_DP (β, c[j], p);
end
```

Note, at this point, that the above definitions are conceptual. At the implementation level, the dependency set is replaced by pointers from abstract tuples to other abstract tuples together with additional information to distinguish between goal, clause and prefix dependencies. In addition, the information on which clauses and prefixes to execute is stored together with the abstract tuples.

During re-execution of a goal, only the clauses whose an element they depend upon has been updated need to be reconsidered. The set of these clauses is defined by

$$\text{MODIFIED\_CLAUSES}(\beta, p) = \{c \mid (\beta, c) \notin dom(dp) \text{ and } c \text{ is a clause of } p\}.$$

To avoid unnecessary UNION operations, two solutions are possible. One solution amounts to computing only the UNION of the re-executed clauses. This requires UNION to be 'accumulative' (i.e. the value of the function only depends on the values of its arguments, not on the order of computation, and does not change if the function is applied several times to the same argument. It is the case in the domains discussed here) and cannot be used in all circumstances (e.g. when the algorithm is applied to a greatest fixpoint computation). The second solution is general and amounts to memorizing the successive values of the variable β_{out}. Both solutions have been implemented and the difference in efficiency is negligible (i.e. cannot be captured with the precision of the clock function). Only the first solution is presented here for simplicity.

Similarly, execution of a clause starts after the first goal which has been updated. The index of the first such goal can be defined as

$$\text{FP}(\beta, c) = min\{i \mid (\beta, c[i]) \notin dom(dp)\ (0 \leq i \leq nbgoal(c))\}$$

To avoid unnecessary computation, the successive values of the local variable β_{ext} need to be saved. We use $logclause(\beta, c[i])$ to represent the value of β_{ext} before the execution of $last(c[i])$ $(1 \leq i \leq nbgoal(c))$.

The operation G_EXTEND generalizes the operation EXTEND to initialize the above data structures properly:

```
procedure G_EXTEND (in β, p, inout sat)
begin
  EXTEND (β, p, sat);
  logclause(β, c[0]) := EXTC(c,β);
end
```

Finally, to simplify the algorithm, we use the function FIRST_PREFIX (β, c) defined as

FIRST_PREFIX (β, c) = (last(c[FP(β, c)]), logclause(β, c[FP(β, c)]))

We are now in a position to define the algorithm with the clause prefix improvement. The new versions of the procedures solve_call and solve_clause are shown in Figure 5.

The procedure solve_call is modified to execute a clause only when the clause does not belong to *dom*(*dp*), i.e. some elements it depends upon have been updated. It also contains the generalized versions of EXTEND and EXT_DP.

The procedure solve_clause is modified to find the first prefix which has been updated together with its associated substitution. It executes all the goals in the prefix. When a procedure call is encountered, the current value of β_{ext} is stored in the log. In addition, the call to ADD_DP has been replaced by a call to G_ADD_DP which inserts not only the dependencies on goals but also on clauses and clause prefixes.

To be even more precise, we must add that our implementation also stores in the log the intermediate values of the variable β_{aux}. β_{aux} needs to be stored to avoid the RESTRG operation for the first goal of the first prefix to re-execute. This allows us to gain 2 to 3 per cent in execution time.

It is interesting at this point to reconsider the trace of append shown in Figure 4. The first iteration is exactly the same as in the standard algorithm. The second iteration is much more interesting, however. First note that the first clause is avoided entirely, since it contains no procedure call. The second clause restarts its execution immediately at the recursive call and none of the unifications are re-executed. The prefix just before the call is shown in the trace and the restriction operation is also avoided. Finally, the last iteration is essentially similar to the second iteration. Note also that, in this example, only the built-ins before the first goal are avoided. In general, many internal built-ins (i.e. built-ins in between two goals), restrictions and extensions are also avoided. This is the case even for simple programs such as qsort, where the second and subsequent iterations always start at the recursive calls. The calls to partition are not re-executed and, more importantly, there is no need to look up in an extension table to recognize that fact. The calls are simply skipped.

4. CACHING OF OPERATIONS

4.1. Motivation

The second improvement we propose is based on the recognition that the operations on substitutions are in fact the most time-consuming operations and should be avoided whenever possible. The improvement is extremely simple conceptually and amounts to caching the results of all operations on substitutions. Each time an operation is executed, the program first looks up to find out if the operation has been encountered already and makes use of the result whenever possible. The optimization subsumes the improvement presented in the previous section in the sense that it avoids computing all the operations for redundant clauses and clause prefixes.* In addition, the caching enables results to be shared between clauses,

* Practically it does not really subsume the first improvement in the sense that it is still necessary to reconsider these clauses although at a negligible cost.

```
procedure solve_call(in β_in, p, suspended; inout sat, dp)
begin
   if (β_in, p) ∉ (dom(dp) ∪ suspended) then
   begin
      if (β_in, p) ∉ dom(sat) then
         sat := G_EXTEND(β_in, p, sat);
      repeat
         β_out := ⊥;
         SC := MODIFIED_CLAUSES(β_in, p);
         G_EXT_DP(β_in, p, dp);
         forall c ∈ SC do
         begin
            solve_clause(β_in, p, c, suspended ∪ {(β_in, p)}, β_aux, sat, dp);
            β_out := UNION(β_out, β_aux)
         end;
         (sat, modified) := ADJUST(β_in, p, β_out, sat);
         REMOVE_DP(modified, dp)
      until (β_in, p) ∈ dom(dp)
   end
end

procedure solve_clause(in β_in, p, c, suspended; out β_out; inout sat, dp)
begin
   (f, β_ext) := FIRST_PREFIX(β_in, c);
   C_EXT_DP(β_in, p, c, dp);
   for i := 1 to m with b_1, ..., b_m body-of c do
   begin
      β_aux := RESTRG(b_i, β_ext);
      switch (b_i) of
      case X_j = X_k:
         β_int := AI_VAR(β_aux)
      case X_j = f(...):
         β_int := AI_FUNC(β_aux, f)
      case q(...):
         logclause(β_in, c[i]) := β_ext;
         solve_call(β_aux, q, suspended, sat, dp);
         β_int := sat(β_aux, q);
         if (β_in, p) ∈ dom(dp) then
            G_ADD_DP(β_in, p, c, i, β_aux, q, dp)
      end;
      β_ext := EXTG(b_i, β_ext, β_int)
   end;
   β_out := RESTRC(c, β_ext)
end
```

Figure 5. The algorithm with the clause prefix improvement

since substitutions are represented in a canonical way. An immediate consequence in the case of finite abstract domains is that the number of operations performed by the algorithm is bounded by the number of program points times the number of times their associated operations can be executed (in the worst case the square of the abstract domain size).*

* Of course, this is only true when all substitutions can fit in memory simultaneously. Otherwise, garbage collection will automatically recover those substitutions which are not strictly necessary (i.e. those not appearing in *sat*).

Figure 6 depicts the execution of the second improvement on the append/3 program. As can be noticed, all operations on the first clause as well as all operations up to the recursive call in the second clause are cached and therefore automatically reused by the algorithm. In this particular case, no further improvement is brought by the caching improvement compared to the clause prefix optimization. However, in other programs, other results will also be shared.

It is also important to note that the caching technique remains useful even when the programs are not normalized, as was the case for the prefix optimization. The

```
CALL solve_clause 1
  EXIT EXTC (Var(1),Var(2),Gro(3))  ps: -(1,1)(2,2)"
  CALL UNIF-FUN (Var(1),Var(2),Gro(3))  ps: -(1,1)(2,2)"
  EXIT UNIF-FUN (Gro(1):[],Var(2),Gro(3))  ps: -(2,2)"
  CALL UNIF-VAR (Gro(1):[],Var(2),Gro(3))  ps: -(2,2)"
  EXIT UNIF-VAR (Gro(1):[],Gro(2),Gro(2))
  EXIT RESTRC (Gro(1):[],Gro(2),Gro(2))
  EXIT UNION (Gro(1):[],Gro(2),Gro(2))
EXIT solve_clause 1
CALL solve_clause 2
  EXIT EXTC (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6))  ps: -(1,1)(2,2)(4,4)(5,5)(6,6)"
  CALL UNIF-FUN (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6))  ps: -(1,1)(2,2)(4,4)(5,5)(6,6)"
  EXIT UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6))  ps: -(2,2)(3,3)(4,4)(6,6)"
  CALL UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6))  ps: -(2,2)(3,3)(4,4)(6,6)"
  EXIT UNIF-FUN (Ngv(1):.(Gro(2),Var(3)),Var(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Var(3),Gro(6))  ps: -(3,3)(4,4)"
  CALL solve_goal append(Var(1),Var(2),Gro(3))  ps: -(1,1)(2,2)"
  EXIT solve_goal append bottom
  EXIT EXTG bottom
  EXIT RESTRC bottom
  EXIT UNION (Gro(1):[],Gro(2),Gro(2))
EXIT solve_clause 2
ADJUST
CALL solve_clause 1
  EXIT EXTC **CACHED** (Var(1),Var(2),Gro(3))  ps: -(1,1)(2,2)"
  CALL UNIF-FUN (Var(1),Var(2),Gro(3))  ps: -(1,1)(2,2)"
  EXIT UNIF-FUN **CACHED** (Gro(1):[],Var(2),Gro(3))  ps: -(2,2)"
  CALL UNIF-VAR (Gro(1):[],Var(2),Gro(3))  ps: -(2,2)"
  EXIT UNIF-VAR **CACHED** (Gro(1):[],Gro(2),Gro(2))
  EXIT RESTRC **CACHED** (Gro(1):[],Gro(2),Gro(2))
  EXIT UNION **CACHED** (Gro(1):[],Gro(2),Gro(2))
EXIT solve_clause 1
CALL solve_clause 2
  EXIT EXTC **CACHED** (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6))  ps: -(1,1)(2,2)(4,4)(5,5)(6,6)"
  CALL UNIF-FUN (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6))  ps: -(1,1)(2,2)(4,4)(5,5)(6,6)"
  EXIT UNIF-FUN **CACHED** (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6))  ps: -(2,2)(3,3)(4,4)(6,6)"
  CALL UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6))  ps: -(2,2)(3,3)(4,4)(6,6)"
  EXIT UNIF-FUN **CACHED** (Ngv(1):.(Gro(2),Var(3)),Var(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Var(3),Gro(6))  ps: -(3,3)(4,4)"
  ** RESTRG CACHED **
  CALL solve_goal append(Var(1),Var(2),Gro(3))  ps: -(1,1)(2,2)"
  EXIT solve_goal append(Gro(1):[],Gro(2),Gro(2))
  EXIT EXTG (Gro(1):.(Gro(2),Gro(3):[]),Gro(4),Gro(5):.(Gro(2),Gro(4)),Gro(2),Gro(3):[],Gro(4))
  EXIT RESTRC (Gro(1):.(Gro(2),Gro(3):[]),Gro(4),Gro(5):.(Gro(2),Gro(4)))
  EXIT UNION (Gro(1),Gro(2),Gro(3))
EXIT solve_clause 2
ADJUST
CALL solve_clause 1
  EXIT EXTC **CACHED** (Var(1),Var(2),Gro(3))  ps: -(1,1)(2,2)"
  CALL UNIF-FUN (Var(1),Var(2),Gro(3))  ps: -(1,1)(2,2)"
  EXIT UNIF-FUN **CACHED** (Gro(1):[],Var(2),Gro(3))  ps: -(2,2)"
  CALL UNIF-VAR (Gro(1):[],Var(2),Gro(3))  ps: -(2,2)"
  EXIT UNIF-VAR **CACHED** (Gro(1):[],Gro(2),Gro(2))
  EXIT RESTRC **CACHED** (Gro(1):[],Gro(2),Gro(2))
  EXIT UNION **CACHED** (Gro(1):[],Gro(2),Gro(2))
EXIT solve_clause 1
CALL solve_clause 2
  EXIT EXTC **CACHED** (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6))  ps: -(1,1)(2,2)(4,4)(5,5)(6,6)"
  CALL UNIF-FUN (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6))  ps: -(1,1)(2,2)(4,4)(5,5)(6,6)"
  EXIT UNIF-FUN **CACHED** (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6))  ps: -(2,2)(3,3)(4,4)(6,6)"
  CALL UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6))  ps: -(2,2)(3,3)(4,4)(6,6)"
  EXIT UNIF-FUN **CACHED** (Ngv(1):.(Gro(2),Var(3)),Var(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Var(3),Gro(6))  ps: -(3,3)(4,4)"
  ** RESTRG CACHED **
  CALL solve_goal append(Var(1),Var(2),Gro(3))  ps: -(1,1)(2,2)"
  EXIT solve_goal append(Gro(1),Gro(2),Gro(3))
  EXIT EXTG (Gro(1):.(Gro(2),Gro(3)),Gro(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Gro(3),Gro(6))
  EXIT RESTRC (Gro(1):.(Gro(2),Gro(3)),Gro(4),Gro(5):.(Gro(2),Gro(6)))
  EXIT UNION (Gro(1),Gro(2),Gro(3))
EXIT solve_clause 2
```

Figure 6. The caching algorithm on append/3

head unifications should be cached, although the hit ratio is likely to be lower since they cluster together many operations which may be unrelated. In addition, operations like RESTRG, EXTG, UNION, EXTC and RESTRC need to be performed on unnormalized programs as well, and would be avoided by the caching optimization. This is especially important, since EXTG and UNION are two of the most costly operations, and the cost of RESTRG would increase in the absence of normalization.

4.2. Implementation

The implementation of the caching algorithm requires two main components

1. a memory manager for the substitutions
2. a number of hashtables, one for each operation, to store the results of the operations.

In addition, all abstract operations are modified to work with pointers to substitutions and to guarantee that the inputs are not modified.

The memory manager is responsible for allocating and deallocating the substitutions. It is accessed by a number of functions to create new substitutions and to copy, to modify, and to free them. Allocation of a substitution occurs mainly as the result of applying an abstract operation. When allocated, a substitution is stored in a hash table and subsequent requests for the same substitution will reuse it. Hence, when asked for allocation of a new substitution, the manager uses the hash table to find out if the substitution already exists, in which case it returns a pointer to the existing substitution. Note that testing if an entry in the hash table is equal to the substitution is extremely fast since the substitutions are represented in a unique way. "Syntactic" equality (instead of the abstract operation) can be used. Garbage collection is performed by associating a counter with each substitution and releasing the substitution when it is no longer referenced. The need for garbage collection comes from the fact that some operations create temporary substitutions which are released later.

The hashing function used for the substitution is of course domain-specific. In the current implementation, it includes most of the components, i.e. the mode component, the pattern component and the same value component. These components are combined in a simple way, using multiplication and exclusive or. The function does not take into account the shape of the terms but considers only the indices of the same-value component, the functors of the pattern component, and the modes of each subterm. Experimental results have shown that the function produces less than 20 per cent of collisions with relatively small tables (less than 1000 entries). Including the sharing component did not produce any improvement and was not included in the final implementation. Better hashing functions are certainly possible and would probably need to take into account the shape of the subterms. In the current implementation, collisions are handled by maintaining a linked list of substitutions for each entry.

Each operation (e.g. EXTG, UNIF-FUNC) is also modified to store its results into a hash table (after the operation is performed) and to look up in the hash table (before the operation is performed). The hashing function is much simpler, since it is performed directly on substitution pointers (instead of on the actual substitutions as is the case for the memory manager). It only uses multiplication of pointers and

possibly integers. Similarly only pointers to substitutions need to be stored since they identify uniquely a substitution. Almost all operations are worth caching. In addition, operations to compare substitutions, i.e. COMPARE and SMALLER, are also cached, although they are internal to other abstract operations. The operation ADJUST is not cached owing to the fact that some of its internal operations are already cached. The first task of ADJUST is to find out if the new result is greater than or non-comparable to the current result. This comparison is cached, and if the result is smaller or equal, nothing else needs to be done. Otherwise the set of abstract tuples is updated but no saving can occur. The system also caches the operation EXTEND because of the special role held in the implementation by this operation. In fact, this operation is responsible for testing membership of *dom*(*sat*), extending the set of abstract tuples if necessary and, in all cases, returning a pointer to the correct element in the set of abstract tuples. This of course implies a search in the set of abstract tuples (represented as Hasse diagrams). All other operations are expressed then in terms of this pointer (including the operation ADJUST). Hence, by caching the operation EXTEND, we make sure that the search is avoided most of the time (only the first time is not cached).

5. EXPERIMENTAL EVALUATION

In this section, we report our experimental results on the optimization techniques. In the following, we denote respectively by Original-P, Original-C, Prefix and Caching (OP, OC, Pr, Ca for short) the original algorithm coded in Pascal, the original algorithm coded in C with a number of optimizations on the domain implementation, the algorithm with the clause prefix improvement, and the algorithm with the caching improvement.

Section 5.1 describes some domain-dependent optimizations performed when moving from the Pascal to the C implementation. Section 5.2 describes the programs used in the experiments, Section 5.3 describes the computation times of the algorithms, Section 5.4 describes the number of operations on substitutions performed by each of the algorithms and the hit-ratios of the caches. Section 5.5 depicts the time distribution between control and abstract operations as well as the time distribution among the various abstract operations, and Section 5.6 reports the memory consumption of the algorithms. Finally, Section 5.7 gives the results on a simple abstract domain.

5.1. Domain optimizations

The C version of the algorithm contains a number of domain-dependent optimizations. The most important one is a lazy computation and representation of the sharing component, which produces about 30 per cent of improvement over the Pascal version. The key idea is to exploit the way sharing is represented in the domain. Only sharing between subterms whose patterns are undefined is represented explicitly. The actual sharing relation is computed by combining the sharing information and the pattern component which implicitly contains some sharing information as well. In the Pascal implementation, the actual sharing implementation was represented explicitly as well and computed at the end of each operation. As a consequence, the representation of the substitutions was heavier, increasing the cost of copying

and comparing substitutions and the cost of each abstract operation. In the C version, only the sharing between subterms whose patterns are undefined is represented. The actual sharing is computed by need when it is requested by an abstract operation. As a consequence, the representation is more compact and the operations cheaper.

Another improvement of the C version is the use of low-level C primitives to copy substitutions and compare them. These operations were performed by for loops in the Pascal version instead of simple memory copying and comparison in C.

Finally, some algorithms for the abstract operations have been revised to make them more data-driven instead of using straightforward top-down implementations.

All the optimizations were suggested by a profiling of the Pascal version of the program and indicate that special care should be taken in representing substitutions and implementing abstract unification.

5.2. The programs

The programs we use are hopefully representative of 'pure' logic programs (i.e. without the use of dynamic predicates such as assert and retract). They are taken from a number of authors and used for various purposes from compiler writing to equation-solvers, combinatorial problems and theorem-proving. Hence they should be representative of a large class of programs. In order to accommodate the many built-ins provided in Prolog implementations and not supported in our current implementation, some programs have been extended with some clauses achieving the effect of the built-ins. Examples are the predicates to achieve input/output and meta-predicates such as setof, bagof, arg and functor. The clauses contain assert and retract have been dropped in the one program containing them (i.e. Syntax error handling in the reader program).

The program kalah is a program which plays the game of kalah. It is taken from Reference 29 and implements an alpha–beta search procedure. The program press is an equation-solver program taken from Reference 29 as well. We use two versions of this interesting program. The first version is the standard version (press1), whereas the second version (press2) has a goal repeated in the program (i.e. a goal is executed twice in a clause). The two versions illustrate a fact often neglected in abstract interpretation. A more precise domain, although requiring a higher cost for the basic operations, might in fact be much more efficient since fewer elements in the domain are explored. The repetition of some goals in the press program allows us to simulate a more precise domain (and hence to gain efficiency). The program cs is a cutting-stock program taken from Reference 30. It is a program used to generate a number of configurations representing various ways of cutting a wood board into small shelves. The program uses, in various ways, the non-determinism of Prolog. The program Disj is taken from Reference 31 and is the generate and test equivalent of a constraint program used to solve a disjunctive scheduling problem. This is also a program using the non-determinism of Prolog. The program Read is the tokenizer and reader written by R. O'Keefe and D.H.D. Warren for Prolog. It is mainly a deterministic program, with mutually recursive procedures. The program PG is a program written by W. Older to solve a specific mathematical problem. The program Gabriel is the Browse program taken from the Gabriel benchmark. The program Plan (PL for short) is a planning program taken from Reference 29. The program Queens is a simple program to solve the *n*-queens

Table I. Computation times of the algorithms and percentages: character version. OP, OC, Pr and Ca denote, respectively, the original algorithm in Pascal, the original algorithm in C, the prefix algorithm and the caching algorithm

Program	OP	OC	Pr	Ca	%OP-OC	%OP-Pr	%OC-Pr	%OP-Ca	%OC-Ca
Append	0·06	0·02	0·01	0·01					
Kalah	17·82	9·20	6·33	5·59	48·37	64·48	31·20	68·63	39·24
Queens	0·37	0·22	0·15	0·16	40·54	59·46	31·82	56·76	27·27
Press1	65·91	37·89	28·82	25·62	42·51	56·27	23·94	61·13	32·38
Press2	19·52	11·53	8·65	8·36	40·93	55·69	24·98	57·17	27·49
Peep	11·34	7·14	5·79	6·29	37·04	48·94	18·91	44·53	11·90
CS	16·02	7·93	5·70	5·92	50·50	64·42	28·12	63·05	25·35
Disj	12·26	6·75	3·03	3·25	44·94	75·29	55·11	73·49	51·85
PG	1·79	1·11	0·86	0·76	37·99	51·96	22·52	57·54	31·53
Read	50·41	30·26	24·42	25·37	39·97	51·56	19·30	49·67	16·16
Gabriel	4·88	2·89	1·95	2·06	40·78	60·04	32·53	57·79	28·72
Plan	1·26	0·71	0·59	0·60	43·65	53·17	16·90	52·38	15·49
QSort	0·56	0·34	0·22	0·23	39·39	60·71	35·29	58·93	32·35
Mean					42·21	58·49	28·38	58·42	28·31

problem. Peep is a program written by S. Debray to carry out the *peephole* optimization in the SB-Prolog compiler. It is a deterministic program. We also use the traditional concatenation and quicksort programs, say Append and Qsort (difference lists).

5.3. Computation times

We give two versions of the computation times. Table I depicts the results with the sharing represented by characters (i.e. bytes), whereas Table II depicts the results

Table II. Computation times of the algorithms and percentages: bit version

Program	OP	OC	Pr	Ca	%OP-OC	%OP-Pr	%OC-Pr	%OP-Ca	%OC-Ca
Append	0·06	0·03	0·02	0·02					
Kalah	17·82	13·52	9·30	7·95	24·13	47·81	31·21	55·39	41·20
Queens	0·37	0·30	0·18	0·18	18·92	51·35	40·00	51·35	40·00
Press1	65·91	53·03	40·52	34·68	19·54	38·52	23·59	47·38	34·60
Press2	19·52	16·06	12·23	11·32	17·73	37·35	23·85	42·01	29·51
Peep	11·34	9·98	8·08	8·62	11·99	28·75	19·04	23·99	13·63
CS	16·02	11·67	8·49	8·43	27·15	47·00	27·25	47·38	27·76
Disj	12·26	9·97	4·49	4·64	18·68	63·38	54·96	62·15	53·46
PG	1·79	1·53	1·19	1·09	14·53	33·52	22·22	39·11	28·76
Read	50·41	43·36	35·51	35·82	13·99	29·56	18·10	28·94	17·39
Gabriel	4·88	4·13	2·74	2·73	15·37	43·85	33·66	44·06	33·90
Plan	1·26	0·99	0·80	0·78	21·43	36·51	19·19	38·10	21·21
QSort	0·56	0·47	0·32	0·28	16·07	42·86	31·91	50·00	40·43
Mean					18·29	41·70	28·75	44·15	31·82

with the sharing represented by bits. The first four columns present the computation times in seconds, whereas the last five columns present the improvements as percentages (i.e. %P1–P2 and denotes (P1–P2)/P1).

As far as the character version is concerned, Caching produces an improvement of about 58 per cent compared to the original version in Pascal. Caching also produces an improvement of about 28 per cent compared to the original version in C. The programs Read and Peep are those producing the least improvement (about 47 per cent), whereas Disj and Kalah produces the best improvement (about 70 per cent). All the times are below 10 seconds except those for Press1 and Read, which take about 25 seconds. Prefix is marginally faster than Caching. It produces an average improvement of about 58 per cent over the original implementation and 28 per cent over the improved implementation in C. All programs are still under 28 seconds, and Prefix loses around three seconds on one of the big programs.

As far as the bit version is concerned, Caching produces an improvement of about 44 per cent over the Pascal implementation (Booleans are not coded as bits by the Pascal compiler) and 31 per cent over the improved C implementation. All programs still run below 12 seconds except Press1 and Read which take about 35 seconds. Prefix is slower, with an average improvement of about 41 per cent over the Pascal implementation and an average of about 28 per cent over the improved C implementation.

The results seem to indicate that the more costly the abstract operations, the more attractive Caching will be. On our domain, the character implementation of sharing (which is the fastest) produces a gain of 0·07 per cent in favour of Prefix, whereas the bit implementation produces a gain of 2·45 per cent in favour of Caching. We discuss this result later in the paper in light of other results.

The above results compare well with the specialized algorithms of References 3 and 4. On Peep, Read and PG, their best programs achieve 22·52, 60·18 and 3·25, respectively, on a SUN 3/50. This means that our algorithm is respectively 3·89, 2·46 and 4·27 times faster on a SPARC-I (4/60) (which is around 2–4 times faster). Moreover, our algorithms execute on a more sophisticated and accurate domain than the one used in References 3 and 4. In particular, our domain also includes sharing and pattern information omitted in References 3 and 4.

Table III indicates the gain of using characters instead of bits on Original-C, Prefix and Caching for the sharing components. The improvement obtained is fairly consistent among the algorithms and is, in general, about 26 per cent.

In summary, the two improvements produce substantial gains in efficiency. Even after a gain of around 40 per cent obtained by the C implementation by refining the abstract domain algorithms, they still produce an improvement of around 30 per cent. Depending upon the implementation of the sharing component (to favour memory or speed), Caching is slower (character version) or faster (bit version) than Prefix. Finally the algorithm efficiency is at least as good as the best specialized tools available for these tasks, although it uses a more sophisticated domain and provides more accurate results (see Reference 2 for details).

5.4. Number of abstract operations

To avoid considering the specifics of our implementation, we now give a more abstract view of the efficiency of the algorithms: the number of operations on

Table III. Percentage gained by using characters on the algorithms

Program	Original-C	Prefix	Caching
Kalah	31·95	31·94	29·69
Queens	26·67	16·67	11·11
Press1	28·55	28·87	26·12
Press2	28·21	29·27	26·15
Peep	28·46	28·34	27·03
CS	32·05	32·86	29·77
Disj	32·30	32·52	29·96
PG	27·45	27·73	30·28
Read	30·21	31·23	29·17
Gabriel	30·02	28·83	24·54
Plan	28·28	26·25	23·08
QSort	27·66	31·25	17·86
Mean	29·32	28·81	25·40

abstract substitutions performed by the various algorithms. The results are summarized in Table IV and depicted in detail in Tables XIII–XXIV given in the Appendix.

Table IV contains, for each abstract operation on all benchmark programs, the number of calls in the algorithms Original-C, Prefix and Caching. CA eval also gives the number of calls in Caching which are really evaluated (all the others being cached). Finally, it gives the percentage of operations saved for each of the improvements. Besides the traditional operations such as RESTRG and EXTG, results are also given for COMPARE (i.e. comparing two substitutions and returning equal, smaller, greater or not comparable), SMALLER (i.e. testing if a substitution is smaller than another substitution), AI_TEST (i.e. the built-in arithmetic comparisons) and AI_IS (i.e. the function is of Prolog). Note also that operation EXTG is only performed for procedure calls and is integrated into the operations UNIF_FUNC and UNIF_VAR for built-ins.

Table IV. Number of abstract operations on all programs for all algorithms

Operation	OC	Pr	Ca	Ca eval	%OC-Pr	%OC-Ca	%OC-Ca eval
COMPARE	7294	5994	3493	1736	17·82	52·11	76·20
SMALLER	24,840	20,390	13,392	8428	17·91	46·34	66·07
EXTEND	3416	2370	3416	987	30·62	0·00	71·11
AI_TEST	934	565	934	462	39·51	0·00	50·54
AI_IS	513	303	513	240	40·94	0·00	53·22
AI_VAR	896	615	896	566	31·36	0·00	36·83
AI_FUNC	13,916	9086	13,916	8208	34·71	0·00	41·06
EXTG	4982	3879	4982	3334	22·14	0·00	33·08
RESTRG	4982	2942	4982	2442	40·95	0·00	50·98
EXTC	5170	3388	5170	3388	34·47	0·00	34·47
RESTRC	5170	4325	5170	2704	16·34	0·00	47·70
UNION	9068	8468	9068	5349	6·62	0·00	41·01

The ratio OC-PR indicates that the percentage of calls saved for each of the operations by Prefix over the original algorithm. Half of the operations have a ratio of over 30 per cent, reaching peaks of about 40 per cent for AI_TEST and AI_IS. The time-consuming operations UNIF_VAR, UNIF_FUNC and EXTG dealing with unification achieve improvements of about 31, 34 and 22 per cent.

The ratio OC-CA eval indicates that the percentage of executed calls saved by Caching. These ratios are much higher than in the case of Prefix, including peaks of about 75 per cent for COMPARE and EXTEND, and about 37 per cent on the unification operations. This seems to indicate and to confirm the results of our previous section that, the more costly the abstract operations, the more attractive will be Caching. When only unification instructions are concerned (i.e. UNIF_VAR, UNIF_FUNC, EXTG) are considered, Caching produces a 7 per cent improvement over Prefix and a 36 per cent improvement over Original-C. Given the overhead for handling the caches, this fits nicely with the results observed for computation times.

The ratio OC-CA gives the number of calls to the operations spared by Caching. Caching basically calls the same operations as Original-C (but many of them are trivially performed through caching) except in the case where some operations are called inside operations. This is true for SMALLER and COMPARE where the number of calls is substantially reduced.

The lowest improvement occurs for EXTG, which was to be expected since this is the instruction executed just after a goal. Each time the output of an abstract tuple has been updated EXTG has to be evaluated. On the other hand, EXTEND has the highest improvement, which is not surprising since this is the operation performed first when an abstract tuple is considered. The most important differences between Caching and Prefix appear in operations UNION and RESTRC, no difference occurring in EXTC. The last result is easily explained since different clauses have very often a different number of variables in their normalized versions. The former result is explained by the fact that Prefix has in fact little to offer for the above operations. For instance, RESTRC is only avoided when the whole clause is not reconsidered.

As far as the individual tables are concerned, a few facts deserve to be mentioned. Read seems to be very peculiar, mainly due to the fact that the program is highly mutually recursive and that the domain is not particularly adequate for the program (see Reference 2 for a discussion of this). As a consequence, it requires many iterations, and exhibits excellent ratios for EXTEND and RESTRC, but rather lower improvements in general. Disj, on the other hand, has excellent ratios almost everywhere owing to its tail-recursive nature (and its substitution-preserving property (see References 1 and 2)).

5.5. Time distribution

In this section, we investigate the distribution of the computation time in various categories, including the abstract time (the time spent in the abstract operation), the control time (the total time − the abstract time) and the cache time (the time taken in managing the caches).

Table V describes the time distribution for caching. TT is the total time, TA the abstract time, TC the control time, and TH the cache time. TA is in fact a lower bound on the abstract time since an abstract operation is never re-executed. Moreover, some of the operations (i.e. ADJUST and EXTEND) are not included. The reason is

Table V. Distribution of computation times for Caching. TT, TA, TC and TH are the total time, the time spent in the abstract operation, the time spent in control, and the time spent in hashing, respectively

Program	TT	TA	TC	TH	TA%TT	TC%TT	TH%TT
Kalah	5·59	5·03	0·56	0·23	90·00	10·00	4·00
Queens	0·16	0·14	0·02	0·00	87·00	13·00	0·00
Press1	25·62	22·48	3·14	2·37	88·00	12·00	9·00
Press2	8·36	7·36	1·00	0·89	88·00	12·00	10·00
Peep	6·29	5·70	0·59	0·57	90·00	10·00	9·00
CS	5·92	5·60	0·32	0·32	94·00	6·00	5·00
Disj	3·25	3·00	0·25	0·25	92·00	8·00	7·00
PG	0·76	0·73	0·03	0·03	96·00	4·00	4·00
Read	25·37	23·73	1·64	1·40	93·00	7·00	5·00
Gabriel	2·06	1·80	0·26	0·16	87·00	13·00	8·00
Plan	0·60	0·52	0·08	0·02	87·00	13·00	3·00
Qsort	0·23	0·17	0·06	0·04	74·00	26·00	12·00

that, on the one hand, these operations contain suboperations that are included, and, on the other hand, much of the remaining time is spent in the updating of the set of abstract tuples which is best considered as control. The ratios TA%TT, TC%TT and TH%TT give the percentage of the total time spent in the abstract time, the control time and the cache time. The results indicate that about 90 per cent of the time is spent in the abstract operations. PG and CS are the most demanding in terms of abstract time, which is easily explained as they manipulate large substitutions and make relatively few iterations (especially CS). The results also indicate that the cache time takes a significant part of the control time, including 10 per cent on Press2. However, assuming a no-cost implementation of the control part, only about 10 per cent can be saved on the computation times. This indicates that the room left for improvement is rather limited.

Table VI depicts the same results (except the cache time) for the original program. It indicates that the control time is very low, only reaching 8 per cent for Queens

Table VI. Distribution of computation time for Original-C

Program	TT	TA	TC	TA%TT	TC%TT
Kalah	9·22	8·89	0·33	96·42	3·58
Queens	0·24	0·22	0·02	91·67	8·33
Press1	38·21	37·44	0·77	97·98	2·02
Press2	11·58	11·47	0·11	99·05	0·95
Peep	7·17	7·15	0·02	99·72	0·28
CS	7·09	7·09	0·00	100·00	0·00
Disj	6·79	6·79	0·00	100·00	0·00
PG	1·07	1·07	0·00	100·00	0·00
Read	30·27	30·03	0·24	99·21	0·79
Gabriel	2·96	2·88	0·08	97·30	2·70
Plan	0·74	0·68	0·06	91·89	8·11
Qsort	0·34	0·32	0·02	94·12	5·88

Table VII. Distribution of computation time for Prefix

Program	TT	TA	TC	TA%TT	TC%TT
Kalah	6·46	6·21	0·25	96·13	3·87
Queens	0·15	0·12	0·03	80·00	20·00
Press1	29·4	28·82	0·43	98·03	1·97
Press2	8·85	8·45	0·40	95·49	4·51
Peep	5·86	5·74	0·12	97·95	2·05
CS	5·87	5·67	0·20	96·60	3·40
Disj	3·03	3·03	0·00	0·00	0·00
PG	0·86	0·81	0·05	94·19	5·81
Read	25·12	24·73	0·39	98·44	1·56
Gabriel	1·93	1·89	0·04	97·93	2·07
Plan	0·56	0·52	0·04	92·86	7·14
Qsort	0·23	0·20	0·03	86·96	13·04

and Plan, but being lower than 3 per cent in most cases. The negligible times for CS, Disj and PG may be explained by the fact that these programs are demanding in abstract time. Comparing those results with Caching, we observe that the control time in Caching has grown significantly due to the cache time (the rest of the control time being theoretically the same between Caching and Original-C).

Table VII depicts the same results (except the cache time) for Prefix. It indicates, as expected, that the control times are almost always smaller to those of Caching and greater than those of Original-C. Also the control times are much closer to Original-C than to Caching.

Table VIII depicts the distribution of the abstract time among the abstract operations for Caching. It clearly indicates that the most time-consuming operations are UNIF_FUNC and EXTG, confirming some of the results of the previous section. For Caching, the operations UNIF_FUNC and EXTG take more than 80 per cent of the time except for Queens (75 per cent). The operation UNION seems to be the next most demanding operation, but far behind the above two operations.

Table VIII. Percentage of time distribution among the abstract operations in Caching

Program	SMALLER	AI_TEST	AI_IS	AI_VAR	AI_FUNC	EXTG	RESTRG	EXTC	RESTRC	UNION
Kalah	0·38	2·17	3·31	0·57	38·72	45·23	0·85	1·79	1·98	5·00
Queens	0·00	9·76	4·07	0·00	45·53	30·08	0·81	2·44	2·44	4·88
Press1	1·33	1·11	1·60	0·53	42·72	41·44	1·33	2·26	1·86	5·81
Press2	0·68	0·68	1·50	0·41	46·92	38·99	1·50	2·19	2·05	5·06
Peep	0·18	0·26	0·09	7·50	53·09	25·31	1·23	2·65	3·70	6·00
CS	0·17	0·95	3·02	1·04	39·60	49·27	0·78	0·86	1·81	2·50
Disj	0·32	0·10	1·16	0·10	58·05	36·06	0·84	1·16	1·16	1·06
PG	1·37	0·14	4·93	1·37	41·37	40·14	1·51	2·47	2·33	4·38
Read	0·66	3·01	0·16	2·10	45·76	38·01	1·24	1·85	1·11	3·05
Gabriel	0·55	0·00	6·02	1·81	35·05	47·75	1·15	2·14	1·86	3·67
Plan	1·93	2·12	0·00	0·19	28·52	53·56	2·50	2·50	3·08	5·59
Qsort	0·00	1·23	0·00	6·17	24·69	56·17	1·85	2·47	2·47	4·94

Table IX. Percentage of time distribution among the abstract operations in Original-C

Program	SMALLER	AI_TEST	AI_IS	AI_VAR	AI_FUNC	EXTG	RESTRG	EXTC	RESTRC	UNION
Kalah	0·79	1·01	3·37	0·56	39·06	46·91	1·23	1·01	1·80	4·26
Queens	4·35	8·70	4·35	0·00	52·17	17·39	4·35	4·35	0·00	4·35
Press1	1·71	0·61	1·92	0·67	48·29	36·79	1·25	1·60	1·84	4·73
Press2	1·08	0·36	2·24	0·72	52·96	33·75	1·26	1·89	1·62	4·13
Peep	0·56	0·14	0·00	7·05	58·11	23·55	0·85	1·83	2·82	5·08
CS	0·26	0·51	3·32	0·77	48·59	41·05	0·64	1·28	1·41	2·17
Disj	0·29	0·00	1·75	0·15	67·45	27·45	1·02	0·73	0·44	0·73
PG	0·94	0·00	6·60	1·89	49·06	33·96	0·94	1·89	0·94	3·77
Read	0·87	2·11	0·20	1·91	49·41	37·76	1·51	1·98	1·21	3·05
Gabriel	0·70	0·00	7·37	3·16	41·75	39·30	1·05	1·75	2·11	2·81
Plan	1·52	1·52	0·00	1·52	36·36	46·97	3·03	1·52	3·03	4·55
Qsort	3·23	0·00	0·00	12·90	38·71	35·48	3·23	0·00	3·23	0·23

Table IX depicts the distribution of the abstract time among the abstract operations for Original-C. The results indicate once again that the most time-consuming operations are UNIF_FUNC and EXTG. The results are also almost similar to those of Caching. Other operations have somewhat different ratios due to the fact that the unification takes most of the time.

5.6. Memory consumption

Tables X and XI depict the memory consumption of the three programs when bits and characters are used for representing the sharing component, respectively. The before field gives the memory requirement before abstract interpretation, i.e. it includes the data structures necessary for parsing and compiling the Prolog programs

Table X. Memory consumption: results with the bit representation of sharing. Memory in Kb

Program	Original-C		Prefix	Caching		Pr/OC	Ca/OC	Ca/Pr
	before	max	max	before	max			
Append	2	4	5	148	152			
Kalah	56	125	244	204	723	1·95	5·78	2·96
Queens	6	12	18	152	182	1·50	15·10	10·11
Press1	82	279	1057	231	2952	3·79	10·58	2·79
Press2	84	176	450	233	1194	2·55	6·78	2·65
Peep	108	145	388	258	1197	2·67	8·25	3·08
CS	42	96	172	190	586	1·79	6·10	3·40
Disj	34	61	120	181	431	1·96	7·06	3·59
PG	13	32	61	159	272	1·90	8·50	4·45
Read	91	210	913	240	2834	4·35	13·49	3·10
Gabriel	26	57	123	173	412	2·16	7·22	3·34
Plan	16	35	66	163	247	1·88	7·05	3·74
Qsort	5	12	21	151	190	1·75	15·83	9·04
Mean						2·35	9·33	4·04

Table XI. Memory consumption: results with the character representation of sharing. Memory in Kb

Program	Original-C		Prefix	Caching		OC/Pr	OC/Ca	Pr/Ca
	before	max	max	before	max			
Append	2	13	14	148	162			
Kalah	56	149	309	204	983	2·07	6·54	3·18
Queens	6	22	28	152	195	1·27	8·86	6·96
Press1	82	324	1314	231	3831	4·05	11·82	2·91
Press2	84	201	547	233	1525	2·72	7·58	2·78
Peep	108	157	453	258	1457	2·88	9·88	3·21
CS	42	121	234	190	860	1·93	7·10	3·67
Disj	34	70	156	181	578	2·22	8·25	3·70
PG	13	45	80	159	315	1·77	7·00	3·93
Read	91	237	1144	240	3820	4·82	16·11	3·33
Gabriel	26	70	150	173	502	2·14	7·17	3·34
Plan	16	46	82	163	275	1·78	3·97	3·35
Qsort	5	21	32	151	207	1·52	9·85	6·46
Mean						2·46	8·84	3·85

as well as the sizes of the hash tables in the case of Caching. The max field gives the maximum memory requirement during the execution of the program. The most memory demanding program is Press1. It requires 279 kilobytes for Original-C, 1057 for Prefix and 2952 for Caching. On average, Prefix requires about twice as much memory as Original-C, but reaches peaks of about four times as much on Read and Press, which are the most time-consuming programs as well. Caching requires about nine times more memory than original on average and reaches a peak of about thirteen times as much on Read.* Caching requires around four times as much memory as Prefix but the ratios are lower on the most demanding programs.

When characters are used, Press1 requires 324, 1314 and 3831 kilobytes for Original-C, Prefix and Caching. On average, Prefix requires about 2·5 times more memory than Original-C but about 4–5 times more on Read and Press1. Caching requires about nine times as much memory as Original-C on average and reaches peaks of about 16 on Read.† Caching requires about four times more memory than Prefix and about three times more on Press1 and Read.

Note also that the percentage of memory saved by using bits instead of characters to represent the sharing component is about 20 per cent in all versions.

5.7. Results on a simpler domain

In this section, we report some experimental results on a simpler domain, i.e. the mode domain of Reference 28 which is a reformulation of the domain of Reference 8. The domain could be viewed as a simplification of the domain discussed so far where the pattern information has been omitted and the sharing has been simplified to an equivalence relation, although all operations are in fact significantly different.

* The high ratios on Qsort and Queens are not significant since the initialization takes most memory.
† Note that the average is only better because of the initialization effect.

The operations are much simpler but the loss of accuracy is significant. Nevertheless, the efficiency results illustrate the potential of the improvements even in unfavourable conditions.

Table XII depicts the efficiency results for the three programs with the bit and character representations of the sharing. For the bit version, Prefix reduces the computation by 28 per cent compared to Original-C, whereas Caching produces a 26 per cent improvement. The improvements still remain significant, given that the improvements of Prefix and Caching on the sophisticated domain were 28 and 31 per cent, respectively. For the character version, there is now a much larger difference in efficiency between Prefix and Caching. Prefix now brings around 29 per cent improvement, whereas Caching only improves Original-C by 6 per cent. Note also that the computation times are significantly reduced compared to the sophisticated domain, all times being less than 8 seconds.

These results indicate the potential of the improvements even on small and simple domains. It also gives us a first confirmation that the simpler the abstract domain, the more interesting Prefix becomes.

6. DISCUSSION AND FUTURE RESEARCH

In this section, we discuss various issues, possible extensions and directions for future research. The discussion is informal, preliminary and tentative, but it points out some open issues and possible solutions.

6.1. Caching

One of the main advantages of the caching technique is its generality: it can be added easily to any abstract interpretation algorithm based on a fixpoint semantics. Therefore it can be useful for the analysis of procedural, functional and parallel languages as well. As an example, caching has also been used in a re-execution-based abstract interpretation algorithm,[32] which interleaves two fixpoint computations and is much more dynamic. However, caching should not be taken as a panacea. Its effectiveness depends on several factors:

1. The control time of the algorithm should be negligible with respect to the time spent in the abstract operations.
2. The percentage of recomputed abstract operation calls must be sufficiently high.
3. The overhead due to the caching technique must be significantly lower than the gain due to non-recomputing abstract operations.

In the experimental results, we mentioned that caching is more appealing when the cost of the abstract operations is high. This is only true, however, when the hit ratio of the caches remains reasonably high (point 2). An interesting issue is to evaluate the impact of caching on the abstract domain prop,[33–35] which contains rather costly abstract operations and differs substantially from the two domains presented in this paper. When the cost of the operations is small, the overhead of caching may reduce its interest (point 3). Finally, caching does not reduce the control time (contrary to prefix) (point 1).

Caching is also more greedy in memory than the prefix optimization. This drawback can be overcome by designing a more sophisticated memory manager. The key idea

Table XII. Computation times and percentages on the small domain. The bit and character versions are distinguished by /B and /C respectively

Program	OC/B	PR/B	CA/B	%PR/B-OC/B	%CA/B-OC/B	OC/C	PR/C	CA/C	PR/C-OC/C	CA/C-OC/C
Append	0·02	0·02	0·04			0·01	0·02	0·02		
Kalah	2·60	1·81	1·88	30	28	1·91	1·41	1·94	26	–2
Queens	0·18	0·14	0·15	22	17	0·15	0·08	0·14	47	7
Press1	6·08	4·08	4·26	33	30	4·72	3·10	4·12	34	13
Press2	6·17	4·23	4·31	31	30	4·74	3·19	4·37	33	8
Peep	5·54	3·86	4·03	30	27	4·22	2·91	4·04	31	4
CS	9·92	7·76	7·29	22	27	7·40	5·83	6·95	21	6
Disj	3·68	2·09	2·20	43	40	2·72	1·57	2·27	42	17
PG	0·48	0·38	0·40	21	17	0·39	0·29	0·39	26	0
Read	5·92	4·25	4·45	28	25	4·52	3·28	4·42	27	2
Gabriel	1·26	0·88	0·94	30	25	0·96	0·69	0·93	28	3
Plan	0·37	0·30	0·34	19	8	0·29	0·25	0·29	14	0
Qsort	0·32	0·23	0·20	28	37	0·25	0·19	0·21	24	16
Mean				28·21	25·91				29·45	6·15

would be to release abstract substitutions when the memory requirement becomes too high. A simple solution is to empty the abstract operation caches. A more elaborate solution would be more selective and would remove the least used and/or the oldest abstract substitutions, following closely the ideas underlying virtual memory systems.

6.2. Prefix

This second technique may seem to be rather specific to Prolog at first glance since it is based on the clause and procedure concepts. However, the principle can be extended to many situations, especially if the abstract interpretation algorithm is designed manually (and not automatically).

Manual design

Suppose that the abstract semantics is defined by an arbitrary algorithm *T* computing some functional transformation. We show in Reference 26 that the algorithm of Reference 1 can be generalized to compute the least fixpoint of such a transformation. This algorithm can then be optimized by inserting some 'recovery points' inside algorithm *T*. Each time algorithm *T* is called with a previously encountered input, its execution will be restarted at the last recovery point whose information does not depend on the changes having occurred since the previous call. More clever and specific improvements can be obtained by analysing the internal structure of algorithm *T*. Independent parts can be identified and recomputed only if some value they depend upon has been improved.

Automization

Partial automization of the method is conceivable. Assuming that the abstract semantics be expressed in a very high level language (e.g. denotational style), a preliminary data flow analysis of the definition could provide a way to decompose the code of the transformation into several independent parts similar to the clauses of a Prolog procedure. Providing mathematical properties of some operations to the pre-analyser could be necessary.

Note, however, that highly dynamic algorithms and the use of widening techniques substantially complicate the use of the prefix optimization. In particular, the computation states that need to be saved may be much more complicated than those needed in the above algorithm. For instance, it is non-trivial, but clearly feasible, to generalize the prefix optimization for the re-execution algorithm of Reference 32. It is, however, an open issue to quantify the difference between caching and prefix on these fixpoint algorithms.

6.3. Combining prefix and caching

The combination of prefix and caching has also been studied, but the experimental results have been inconclusive, the combined algorithm being always worse than at least one of the two basic algorithms. The reason may be that the prefix optimization, when combined with the caching version, only reduces the control time which is

rather low already. In addition, it does so by complicating the dependency graph and copying substitutions, thus adding control time as well. However, the combination could be worth while when the control time of the algorithm is much higher.

6.4. Clause ordering

There is a variation of our algorithm which consists of updating the set of abstract tuples after each clause analysis instead of taking the union of the results and updating once. This variant is best combined with a *clause reordering*, so that non-recursive clauses would be handled first. Clause reordering can be performed using a simple analysis. The combination of these two ideas leads to a reduction of the number of iterations, of the control time of the algorithms, but does not really avoid any abstract operation when added to the prefix and caching algorithms. Our preliminary experiments with this idea have been inconclusive, the gain or the loss in efficiency being negligible. This result seems to be explained by the fact that the prefix optimization can also be viewed as an automatic and dynamic reordering of clauses, since the prefixes are never re-executed and a clause reordering would not remove any abstract operation.

6.5. Granularity

The algorithm presented here works at a fine granularity, i.e. it stores many input/output pairs with each predicate. Many other granularities have been proposed in the literature. For instance, Mellish[6] proposes to store a single input/output pair per predicate, whereas Nilsson's framework[12] preserves an input/output pair per program point (i.e. program goals). The impact of granularity on the efficiency and accuracy of abstract interpretation is an open issue, and the various alternatives needs to be evaluated experimentally. Note, however, that a coarser granularity does not necessarily lead to a better efficiency, since precision can be lost and convergence to the fixpoint (or the postfixpoint) may be slower. See, for instance, References 2 and 32 for more discussion on this topic.

7. CONCLUSION

In this paper, we have reconsidered the implementation of generic abstract interpretation algorithms for Prolog. We have presented two optimization techniques on the original algorithm:[1] clause prefix dependencies and caching. Clause prefix dependencies amount to generalizing the dependency graph to clauses and clause prefixes, whereas caching amounts to memorizing all operations on abstract substitutions. The optimization techniques have been evaluated experimentally and compared to the original implementation. Together with some optimization techniques on the abstract domain, they produce an average speed-up of about 60 per cent on the initial implementation. Experimental results on the computation times, the number of operations, the time distribution as well as the memory consumption have been presented for both algorithms and compared to the original algorithm. In addition, hit-ratios for the caches and comparisons between several implementations of the sharing component have been given. An interesting result is that the control part is reduced to 10 per cent in caching, indicating that there is not much room left for

improvement. Results on a simpler domain indicate that even basic domains can benefit from the optimization since the prefix improvement still reduces computation times by about 28 per cent.

It is not fully clear which of the two optimization techniques is most valuable in practice. The clause prefix improvement has the advantage of simplicity and memory consumption. The Caching algorithm can be reused more easily in other contexts, but it also requires more memory. Attempts to combine the two improvements turned out to be unsuccessful, the combined algorithm being always worse than at least one of the two basic algorithms. It is our belief that Caching is more attractive for sophisticated domains and Prefix for simpler domains. Future research and experimentation on other domains will help us answering this question and identify what were the peculiarities of our domains.

ACKNOWLEDGEMENTS

Saumya Debray and Leon Sterling provided us with some of the programs used in the experiments. This research was partly supported by the Belgian National Incentive-Program for fundamental Research in Artificial Intelligence (Baudouin Le Charlier) and by the U.S. National Science Foundation under grant number CCR-9108032 and the U.S. Office of Naval Research under grant N00014-91-J-4052 ARPA order 8225. (Pascal Van Hentenryck).

REFERENCES

1. B. Le Charlier, K. Musumbu and P. Van Hentenryck, 'A generic abstract interpretation algorithm and its complexity analysis (extended abstract)', *Eighth International Conference on Logic Programming (ICLP-91)*, Paris, France, June 1991.
2. B. Le Charlier and P. Van Hentenryck, 'Experimental evaluation of a generic abstract interpretation algorithm for Prolog', *Fourth IEEE International Conference on Computer Languages (ICCL '92)*, San Francisco, CA, April 1992.
3. M. Hermenegildo, R. Warren and S. Debray, 'Global flow analysis as a practical compilation tool', *Journal of Logic Programming*, **13**(4), 349–367 (1992).
4. R. Warren, M. Hermedegildo and S. Debray, 'On the practicality of global flow analysis of logic programs', *Proc. Fifth International Conference on Logic Programming*, Seattle, WA, August 1988, pp. 684–699.
5. P. Cousot and R. Cousot, 'Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints', *Conf. Record of Fourth ACM Symposium on POPL*, Los Angeles, CA, 1977, pp. 238–252.
6. C. Mellish, *Abstract Interpretation of Prolog Programs*, Ellis Horwood, Chichester, 1987, pp. 181–198.
7. R. Barbuti, R. Giacobazzi and G. Levi, 'A general framework for semantics-based bottom-up abstract interpretation of logic programs', to appear in *ACM Transactions on Programming Languages and Systems*.
8. M. Bruynooghe, 'A practical-framework for the abstract interpretation of logic programs', *Journal of Logic Programming*, **10**, 91–124 (1991).
9. N. D. Jones and H. Sondergaard, *A Semantics-Based Framework for the Abstract Interpretation of Prolog*, Ellis Horwood, Chichester, 1987, pp. 123–142.
10. K. Marriott and H. Sondergaard, 'Bottom-up abstract interpretation of logic programs', *Proc. Fifth International Conference on Logic Programming*, Seattle, WA, August 1988, pp. 733–748.
11. K. Marriott and H. Sondergaard, 'Notes for a tutorial on abstract interpretation of logic programs', *North American Conference on Logic Programming*, Cleveland, Ohio, 1989.
12. U. Nilsson, 'A systematic approach to abstract interpretation of logic programs', *Ph.D. Thesis*, Department of Computer and Information Science, Linkoping University, Linkoping (Sweden), December 1989.
13. W. H. Winsborough, 'A minimal function graph semantics for logic programs', *Technical Report TR-711*, Computer Science Department, University of Wisconsin at Madison, August 1987.

14. S. Debray and P. Mishra, 'Denotational and operational semantics for prolog', *Journal of Logic Programming*, **5**(1), 61–91 (1988).
15. C. Mellish, 'The automatic generation of mode declarations for Prolog programs', *Technical Report DAI Report 163*, Department of Artificial Intelligence, University of Edinburgh, 1981.
16. M. Bruynooghe and G. Janssens, 'An instance of abstract interpretation: integrating type and mode inferencing', *Proc. Fifth International Conference on Logic Programming*, Seattle, WA, August 1988, pp. 669–683.
17. H. Sondergaard, 'An application of abstract interpretation of logic programs: occur check reduction', *Proc. ESOP '86*, Sarrbruecken, Germany, 1986, pp. 327–338.
18. A. Mulkers, W. Winsborough and M. Bruynooghe, 'Analysis of shared data structures for compile-time garbage collection in logic programs', *Seventh International Conference on Logic Programming (ICLP-90)*, Jerusalem, Israel, June 1990.
19. K. Muthukumar and M. Hermenegildo, 'Determination of variable dependence information through abstract interpretation', *Proc. North American Conference on Logic Programming (NACLP-89)*, Cleveland, Ohio, October 1989.
20. W. Winsborough, 'Multiple specialization using minimal-function graph semantics', *Journal of Logic Programming*, **13**(2–3), 259–290 (1992).
21. M. Bruynooghe, G. Janssen, A. Callebaut and B. Demoen, 'Abstract interpretation: towards the global optimization of Prolog programs', *Proc. 1987 Symposium on Logic Programming*, San Francisco, CA, August 1987, pp. 192–204.
22. M. Codish, J. Gallagher and E. Shapiro, 'Using safe approximations at fixed points for analysis of logic programs', in *Meta-programming in Logic Programming*, Bristol, UK, 1989.
23. C. Codognet, P. Codognet and J. M. Corsini, 'Abstract interpretation of concurrent logic languages', *Proc. North American Conference on Logic Programming (NACLP-90)*, Austin, TX, October 1990.
24. B. Le Charlier, K. Musumbu and P. Van Hentenryck, 'Efficient and accurate algorithms for the abstract interpretation of Prolog programs', *Research Paper RP-90/9*, University of Namur, August 1990.
25. V. Englebert and D. Roland, 'Abstract interpretation of prolog programs: optimizations of an implementation', *Mémoire de Licence et Maîtrise en Informatique*, June 1992.
26. B. Le Charlier and P. Van Hentenryck, 'A universal top-down fixpoint algorithm', *Technical Report CS-92-25*, CS Department, Brown University, 1992.
27. D. Jacobs and A. Langen, 'Accurate and efficient approximation of variable aliasing in logic programs', *Proc. of the North-American Conference on Logic Programming (NACLP-89)*, Cleveland, Ohio, October 1989.
28. K. Musumbu, 'Interpretation abstraite de programmes Prolog', *Ph.D. Thesis*, University of Namur, Belgium, September 1990.
29. L. Sterling and E. Shapiro, *The Art of Prolog: Advanced Programming Techniques*, MIT Press, Cambridge, MA, 1986.
30. P. Van Hentenryck, *Constraint Satisfaction in Logic Programming*, Logic Programming Series, The MIT Press, Cambridge, MA, 1989.
31. M. Dincbas, H. Simonis and P. Van Hentenryck, 'Solving large combinatorial problems in logic programming', *Journal of Logic Programming*, **8**(1–2), 75–93 (1990).
32. B. Le Charlier and P. Van Hentenryck, 'Reexecution in abstract interpretation of Prolog', *Proc. International Joint Conference and Symposium on Logic Programming (JICSLP-92)*, Washington, DC, November 1992.
33. A. Cortesi, G. Filé and W. Winsborough, 'Prop revisted: propositional formulas as abstract domain for groundness analysis', *Proc. Sixth Annual IEEE Symposium on Logic in Computer Science (LICS '91)*, 1991, pp. 322–327.
34. B. Le Charlier and P. Van Hentenryck, 'Groundness analysis for Prolog: implementation and evaluation of the domain prop', *Technical Report CS-92-49*, CS Department, Brown University, October 1992.
35. K. Marriott and H. Sondergaard, 'Abstract interpretation of logic programs: the denotational approach, to appear in *ACM Trans. Programming Languages*.

APPENDIX: RESULTS ON THE INDIVIDUAL OPERATIONS

Table XIII. Number of operations on COMPARE

Program	Original-C	Prefix		Caching			
	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	6	6	0·00	4	1	4·00	83·33
Kalah	325	268	17·54	168	95	1·77	70·77
Queens	35	31	11·43	21	8	2·62	77·14
Press1	2716	2245	17·34	1384	676	2·05	75·11
Press2	869	710	18·30	483	221	2·19	74·57
Peep	457	399	12·69	218	61	3·57	86·65
CS	188	172	8·51	108	43	2·51	77·13
Disj	191	141	26·18	84	43	1·95	77·49
PG	105	97	7·62	62	39	1·59	62·86
Read	1983	1561	21·28	725	425	1·71	78·57
Gabriel	257	219	14·79	149	75	1·99	70·82
Plan	95	89	6·32	51	28	1·82	70·53
Qsort	67	56	16·42	36	21	1·71	68·66

Table XIV. Number of operations on SMALLER

Program	Original-C	Prefix		Caching			
	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	8	8	0·00	8	5	1·60	37·50
Kalah	858	678	20·98	522	241	2·17	71·91
Queens	68	52	23·53	52	26	2·00	61·76
Press1	8549	6775	20·75	4767	2953	1·61	65·46
Press2	1953	1533	21·51	1134	601	1·89	69·23
Peep	1396	1284	8·02	693	418	1·66	70·06
CS	468	408	12·82	354	175	2·02	62·61
Disj	472	308	34·75	246	76	3·16	83·90
PG	211	193	8·53	169	95	1·78	54·98
Read	9975	8429	15·50	4783	3510	1·36	64·81
Gabriel	523	414	20·84	371	200	1·85	61·76
Plan	249	227	8·84	171	92	1·86	63·05
Qsort	110	81	26·36	59	36	1·64	67·27

Table XV. Number of operations on EXTEND

Program	Original-C	Prefix		Caching			
	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	1	1	0·00	1	1	1·00	0·00
Kalah	170	118	30·59	170	71	2·39	58·24
Queens	11	7	36·36	11	7	1·57	36·36
Press1	1065	711	33·24	1065	348	3·06	67·32
Press2	353	232	34·28	353	129	2·74	63·46
Peep	227	193	14·98	227	57	3·98	74·89
CS	77	61	20·78	77	45	1·71	41·56
Disj	105	55	47·62	105	34	3·09	67·62
PG	35	29	17·14	35	22	1·59	37·14
Read	1199	840	29·94	1199	191	6·28	84·07
Gabriel	101	66	34·65	101	49	2·06	51·49
Plan	49	43	12·24	49	26	1·88	46·94
Qsort	23	14	39·13	23	7	3·39	69·57

Table XVI. Number of operations on AI_TEST

Program	Original-C	Prefix		Caching			
	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	0	0		0			
Kalah	81	47	41·98	81	41	1·98	49·38
Queens	12	6	50·00	12	6	2·00	50·00
Press1	265	149	43·77	265	99	2·68	62·64
Press2	85	47	44·71	85	34	2·50	60·00
Peep	39	21	46·15	39	16	2·44	58·97
CS	35	15	57·14	35	13	2·69	62·86
Disj	6	4	33·33	6	3	2·00	50·00
PG	7	3	57·14	7	1	7·00	85·14
Read	383	262	31·59	383	241	1·59	37·08
Gabriel	0	0		0			
Plan	15	9	40·00	15	6	2·50	60·00
Qsort	6	2	66·67	6	2	3·00	66·67

Table XVII. Number of operations on AI_IS

Program	Original-C	Prefix		Caching			
	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	0	0		0			
Kalah	63	42	33·33	63	33	1·91	47·62
Queens	4	2	50·00	4	2	2·00	50·00
Press1	220	134	39·09	220	102	2·16	53·64
Press2	79	43	45·57	79	35	2·26	55·70
Peep	0	0		0			
CS	33	17	48·48	33	16	2·06	51·52
Disj	16	6	62·50	16	5	3·20	68·75
PG	20	12	40·00	20	9	2·22	55·00
Read	22	16	27·27	22	12	1·83	45·45
Gabriel	56	31	44·64	56	26	2·15	53·57
Plan	0	0		0			
Qsort	0	0		0			

Table XVIII. Number of operations on AI_VAR

Program	Original-C	Prefix		Caching			
	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	3	1	66·67	3	1	3·00	66·67
Kalah	22	15	31·82	22	14	1·57	36·36
Queens	0	0		0			
Press1	210	121	42·38	210	102	2·06	51·43
Press2	71	39	45·07	71	37	1·92	47·89
Peep	203	175	13·79	203	167	1·22	17·73
CS	12	6	50·00	12	6	2·00	50·00
Disj	16	8	50·00	16	8	2·00	50·00
PG	13	7	46·15	13	6	2·17	53·85
Read	264	195	26·14	264	191	1·38	27·65
Gabriel	55	32	41·82	55	24	2·29	56·36
Plan	4	2	50·00	4	2	2·00	50·00
Qsort	23	14	39·13	23	8	2·88	65·22

Table XIX. Number of operations on AI_FUNC

Program	Original-C	Prefix		Caching			
	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	9	3	66·67	9	3	3·00	66·67
Kalah	605	376	37·85	605	335	1·81	44·63
Queens	60	26	56·67	60	25	2·40	58·33
Press1	4726	3042	35·63	4726	2534	1·87	46·38
Press2	1748	1154	33·98	1748	1016	1·72	41·88
Peep	1531	1092	28·67	1531	1082	1·41	29·33
CS	528	252	52·27	528	248	2·13	53·03
Disj	494	202	59·11	494	202	2·45	59·11
PG	186	106	43·01	186	93	2·00	50·00
Read	3463	2503	27·72	3463	2361	1·47	31·82
Gabriel	411	235	42·82	411	222	1·85	45·99
Plan	98	68	30·61	98	66	1·48	32·65
Qsort	57	27	52·63	57	21	2·71	63·13

Table XX. Number of operations on EXTG

Program	Original-C	Prefix		Caching			
	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	3	3	0·00	3	3	1·00	0·00
Kalah	226	174	23·01	226	151	1·50	33·19
Queens	22	18	18·18	22	17	1·29	22·73
Press1	1669	1297	22·29	1669	1037	1·61	37·87
Press2	577	449	22·18	577	389	1·48	32·58
Peep	367	315	14·17	367	291	1·26	20·71
CS	130	114	12·31	130	106	1·23	18·46
Disj	149	99	33·59	149	98	1·52	34·23
PG	62	56	9·68	62	47	1·32	24·19
Read	1495	1122	24·95	1495	990	1·51	33·78
Gabriel	173	138	20·23	173	120	1·44	30·64
Plan	67	61	8·96	67	56	1·20	16·42
Qsort	42	33	21·43	42	29	1·45	30·95

Table XXI. Number of operations on RESTRG

Program	Original-C	Prefix		Caching			
	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	3	1	66·67	3	1	3·00	66·67
Kalah	226	128	43·36	226	115	1·97	49·12
Queens	22	10	54·55	22	9	2·44	59·09
Press1	1669	962	42·36	1669	716	2·33	57·10
Press2	577	322	44·19	577	263	2·19	52·69
Peep	367	222	39·51	367	212	1·73	42·23
CS	130	76	41·54	130	69	1·88	46·92
Disj	149	69	53·69	149	68	2·19	54·36
PG	62	38	38·71	62	29	2·14	53·23
Read	1495	945	36·79	1495	820	1·82	45·15
Gabriel	173	98	43·35	173	80	2·16	53·76
Plan	67	49	26·87	67	44	1·45	34·33
Qsort	42	22	47·62	42	16	2·63	61·90

Table XXII. Number of operations on EXTC

Program	Original-C	Prefix		Caching			
	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	6	2	66·67	6	2	3·00	66·67
Kalah	229	136	40·61	229	136	1·68	40·61
Queens	29	13	55·17	29	13	2·23	55·17
Press1	1835	1177	35·86	1835	1177	1·56	35·86
Press2	701	458	34·66	701	458	1·53	34·66
Peep	530	380	28·30	530	380	1·39	28·30
CS	153	81	47·06	153	81	1·89	47·06
Disj	124	64	48·39	124	64	1·94	48·39
PG	80	44	45·00	80	44	1·82	45·00
Read	1181	850	28·03	1181	850	1·39	28·03
Gabriel	190	113	40·53	190	113	1·68	40·53
Plan	78	56	28·21	78	56	1·39	28·21
Qsort	34	14	58·82	34	14	2·43	58·82

Table XXIII. Number of operations on RESTRC

Program	Original-C	Prefix		Caching			
	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	6	4	33·33	6	4	1·50	33·33
Kalah	229	182	20·52	229	156	1·47	31·88
Queens	29	21	27·59	29	19	1·53	34·48
Press1	1835	1512	17·60	1835	761	2·41	58·53
Press2	701	585	16·55	701	388	1·81	44·65
Peep	530	473	24·39	530	411	1·29	22·45
CS	153	119	22·22	153	99	1·55	35·29
Disj	124	94	24·19	124	91	1·36	26·61
PG	80	62	22·50	80	47	1·70	41·25
Read	1181	1027	13·04	1181	559	2·11	52·67
Gabriel	190	153	19·47	190	106	1·79	44·21
Plan	78	68	12·82	78	46	1·70	41·03
Qsort	34	25	26·47	34	17	2·00	50·00

Table XXIV. Number of operations on UNION

Program	Original-C	Prefix		Caching			
	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	11	9	18·18	11	6	1·83	45·45
Kalah	485	455	6·19	485	259	1·87	46·60
Queens	58	50	13·79	58	29	2·00	50·00
Press1	3266	3045	6·77	3266	2000	1·63	38·76
Press2	1209	1140	5·71	1209	674	1·79	44·25
Peep	711	676	4·92	711	522	1·36	26·58
CS	324	290	10·49	324	157	2·06	51·54
Disj	257	227	11·67	257	93	2·76	63·81
PG	166	148	10·84	166	80	2·08	51·81
Read	1993	1899	4·72	1993	1240	1·61	37·78
Gabriel	360	322	10·56	360	176	2·05	51·11
Plan	160	150	6·25	160	75	2·13	53·13
Qsort	68	57	16·18	68	38	1·79	44·12