Some Low-Level Source Transformations for Logic Programs *

John Gallagher †
Maurice Bruynooghe

Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3030 Heverlee, Belgium

March 1990

Abstract

This paper describes an algorithm performing an analysis and transformation of logic programs. The transformation achieves two goals: redundant functors are removed from the program, and procedures may be split into two or more specialised versions handling different cases. It can be applied to most logic programming languages, including concurrent logic programming languages, because the transformations perform no unfolding of the program; they only remove some redundant operations within the unifications. The main saving is in heap usage, though time performance may also be improved. One of the main purposes of the transformation is to "clean up" programs generated by other methods of transformation or synthesis. The analysis is an example of an abstract interpretation, and is guaranteed to terminate. A Prolog implementation of the algorithm, illustrating some meta-programming techniques, is given and some results are reported.

Keywords: program analysis, abstract interpretation, program transformation.

1 Introduction

The most common measure of efficiency for computations of logic programs is the number of logical inferences (reductions) performed. While serving a useful purpose, this often leads Prolog programmers to ignore the complexity of the reductions themselves. Also, few Prolog programmers who have not actually implemented the language are aware of the effects on space usage of various programming styles. This is by no means a bad thing, for one of the aims of declarative programming is surely to free the programmer from low-level concerns. Besides, the development of better compilers held forth the promise that the programmer could safely ignore the levels below unification, because many inefficiencies could be "compiled away".

*Work performed in ESPRIT Basic Research Action Project COMPULOG (3012)

†Current address: Dept of Computer Science, University of Bristol, Bristol BS8 1TR, U.K.

In this paper we tackle the problem of "transforming away" some kinds of inefficiency. The transformation depends on an analysis that performs an abstract computation starting from a given initial goal. The result of the analysis is a description of the calls to procedures in the program; these descriptions are used to construct new procedures "specialised" to the calls. In some cases two or more versions of a procedure might be constructed, handling different call types.

The transformations themselves are not very deep and many experienced Prolog programmers will no doubt already apply them. However we think they are of interest for several reasons.

- They can significantly reduce the heap consumption of some programs, and typically even improve the behaviour of carefully written programs by a small amount. We have not found an example where space or time performance was worse after the transformations were applied.
- They can be applied in conjunction with other kinds of program transformation or synthesis methods. Such methods often leave a lot of "dead" structure in the transformed program. In this case our transformation acts as a cleaning up operation. This was in fact the original motivation for this work.
- The analysis phase is an instance of a general approach to logic program analysis based on abstract interpretation.
- The analysis and transformation are computationally cheap and can be applied to almost all Prolog constructs, and in addition can be applied to some concurrent logic programs. (There are complications for languages that require mode declarations for arguments).

Two examples illustrate the main ideas. Time and space usage are similar in all WAM-based Prolog implementations. We measured performance using BIMprolog [2], a fast Prolog implementation based on the WAM.

EXAMPLE **1**. *Difference lists are often represented by a functional term such as* `Xs-Ys`. *In this way the difference list is easily identified when reading the program. However its presence introduces unnecessary operations into computations.*

The program for reversing a list, yielding a difference list, is written as follows:

```
reverse(nil, Ys-Ys).
reverse([X|Xs], Ys-Zs) :- reverse(Xs, Ys-[X|Zs]).
```

The presence of the function symbol `"-"` *plays no computational role, assuming that the program is always called with an argument containing* `"-"`, *as in* `reverse([a,b,c],Ys-[])`. *The* `reverse` *procedure is compiled to handle arbitrary calls, but every call actually contains a second argument instantiated to a structure* `X-Y`. *Each unification of a call with the head of one of the* `reverse` *clauses thus performs a redundant comparison of* `"-"` *with* `"-"`.

A more serious drawback of the use of the functor is that heap space is used to store the `"-"` *structure each time the body of the recursive clause is activated. The following program removes these two drawbacks.*

```
reverse(nil,Ys,Ys).
reverse([X|Xs],Ys,Zs) :- reverse(Xs,Ys,[X|Zs]).
```

When solving a query such as `reverse([a,b,c],Ys,[])`*, this uses half the heap space of the first program.*

EXAMPLE **2**. *The following program was produced by a transformation using the compiling control method [4],[6].*

```
fib(N,F) :-
        p(fib(N,F),fib(0,1),fib(1,1)).

p(fib(0,1),_,_).
p(fib(N,F),_,fib(N,F)).
p(fib(N,F),fib(N1,F1),fib(N2,F2)) :-
        N2 is N1 + 1,
        N3 is N2 + 1,
        N3 > 1,
        F3 is F1 + F2,
        p(fib(N,F),fib(N2,F2),fib(N3,F3)).
```

It represents a bottom-up execution of the usual clauses defining fibonacci numbers. The predicate `p` *of the above program is a meta-predicate whose arguments are object-language predicates. Assuming that the program is always called using the* `fib` *predicate, the structures in the arguments to* `p` *are redundant since all calls to* `p` *have all three arguments bound to structures* `fib(X,Y)`*.*

The result of the transformation is the following program:

```
fib(X1,X2) :-
      p1(X1,X2,0,1,1,1) .
p1(0,1,X1,X2,X3,X4) :-
      true .
p1(X1,X2,X3,X4,X1,X2) :-
      true .
p1(X1,X2,X3,X4,X5,X6) :-
      X5 is X3 + 1,
      X7 is X5 + 1,
      1 < X7,
      X8 is X4 + X6,
      p1(X1,X2,X5,X6,X7,X8) .
```

The new program uses essentially no heap space, whereas the first one consumes heap proportional to `N` *when computing* `fib(N,M)`*. It is also possible to declare more accurate modes for the transformed program, further improving time performance (see Section 6).*

We now proceed to define the operations needed to achieve these transformations.

2 Specialised Procedures

The idea of specialising a procedure to a call is the central idea of the transformation.

DEFINITION **2.1**. *Let P be a program containing a procedure consisting of the n clauses $p(t_1) \leftarrow B_1, ..., p(t_n) \leftarrow B_n$. Let $p(s)$ be an atom. We define a procedure called the specialisation of p for $p(s)$. Let $x_1 \ldots x_k$ be the distinct variables in $p(s)$, in the order of their first occurrence, and assume that $x_1 \ldots x_k$ do not occur in P. Let q be a predicate symbol not occurring in P. Let θ_i be an mgu of $p(s)$ and $p(t_i)$, for $1 \leq i \leq n$. The procedure for q is the set of clauses*

$q(x_1 \ldots x_k)\theta_i \leftarrow B_i\theta_i$ *for each i such that $\theta_i \neq fail$.*

EXAMPLE **3**. *Let P be the program:*

```
append(nil,Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

Let `append([a,b|Us],[c],Ws)` *be a query. Then define a new query* `append1(Us,Ws)`*. The procedure for* `append1` *is:*

```
append1(Us,[a|Zs]) :- append([b|Us],[c],Zs).
```

Note that the answer substitutions for the query `append1(Us,Ws)` are the same as those for `append([a,b|Us],[c],Ws)`. This is expressed more generally in the following lemma.

LEMMA **2.2**. *Let a program P, a procedure for predicate p and an atom $p(s)$ be given as in Definition 2.1. Let the procedure Q for a new predicate q be the specialisation of p for $p(s)$. Then σ is a computed answer substitution for $p(s)\theta$ in P iff σ is a computed answer for $q(x_1 \ldots x_k)\theta$ in $P \cup Q$.*

This lemma shows that if we replace a query $p(s)$ by a query $q(x_1 \ldots x_k)$ containing the variables of $p(s)$, the same answer substitutions are retained. Note that if $p(s)$ contains some nonvariable arguments, or variables with more than one occurrence, then the first reduction of $q(x_1 \ldots x_k)$ will be more efficient than the corresponding reduction of $p(s)$. This is partly because the unification with the head contains less operations than before, and partly because indexing of the procedure for q by the compiler can be more effective than the indexing for p. An artificial example illustrates the point.

EXAMPLE **4**. *Let the procedure for* `p` *be:*

```
p(s(a)) :- Q1.
p(s(b)) :- Q2.
p(s(c)) :- Q3.
```

Suppose all calls for `p` *are of the form* `p(s(X))`*. Then we can replace calls* `p(s(X))` *by* `p1(X)` *and define a new procedure:*

```
p1(a) :- Q1.
p1(b) :- Q2.
p1(c) :- Q3.
```

Not only is the matching of the symbol `s` *avoided, but indexing on the principal functor distinguishes the clauses in the second program, but does not in the first.*

We have now described the basis for the transformation. If we can derive, for each predicate p in a program, an atom $p(s)$ that describes all calls to p, (in the sense that all calls to p are instances of $p(s)$), then we can replace the procedure for p with a procedure for q, and systematically replace all body atoms $p(t) = p(s)\theta$ by $q(x_1 \ldots x_k)\theta$. Next we introduce an analysis method that yields an approximation to the set of calls to each predicate.

3 Analysing the Set of Calls in a Prolog Computation

A Prolog clause is of the form $A \leftarrow B_1, \ldots, B_n$ where each B_i is an atom or a literal $\neg C$ or a metacall such as $bagof(X, G, S)$ or $G_1; G_2$ (disjunction), which has a goal as an argument. The atoms in clause bodies are either defined by clauses or are calls to built in procedures such as `<, =.., !, write` and so on. We aim for a transformation method with as wide a coverage of Prolog as possible. Our method as stated does not make any syntactic restriction on Prolog; but there is discussion in Section 4 on limitations of certain uses of `assert, retract` and metacalls.

Let G_0 be an atomic goal, and P a Prolog program. A *Prolog computation* is a sequence of goals $\leftarrow G_0, \leftarrow G_1, \leftarrow G_2, \ldots$, where G_{i+1} $(i \geq 0)$ is obtained in one of the following ways:

- by resolving G_i on its leftmost literal with a clause in P:
- by starting a sub-computation for A, where A is an atom in the leftmost literal (a negative literal or metacall) in G_i;
- by returning from a sub-computation activated by some previous G_k, $k < i$. That is, G_{i+1} is some instance of $\leftarrow A_2, \ldots A_m$, where $G_k = \leftarrow A_1, \ldots, A_m$ and A_1 is a metacall.

The set of atoms $Calls_G$ (assuming some fixed program P) is defined as:

$$Calls_G = \left\{ A_1 \;\middle|\; \begin{array}{c} \leftarrow A_1, \ldots, A_k \text{ occurs in a Prolog} \\ \text{computation starting from } G \end{array} \right\}$$

3.1 Approximating the Set of Calls

$Calls_G$ is infinite in general. In order to perform procedure specialisation as defined above it is necessary to compute some finitely representable approximation to $Calls_G$. We now define the concept of approximation.

DEFINITION **3.1**. *Let V be a countable set of variables, and let Atoms be the set of all atoms constructed from some finite alphabet of predicate and function symbols and V; let $S \subseteq Atoms$. We define the downward closure of S, denoted $[S]$, as $\{A\theta \mid A \in S\}$. S is downward closed if $S = [S]$.*

DEFINITION **3.2**. *Let S and T be sets of atoms. T is said to be a safe approximation of S if $[S] \subseteq [T]$.*

The downward closure of a set of atoms is taken as the basis for approximation because we wish to say, for example, that the set of calls $\{p(X), q(f(Y)\}$ is a safe approximation of $\{p(a), p(b), q(f(a))\}$; any procedure able to handle the first set can also handle the second.

The aim of the next section is to develop a way of computing some finite set of atoms S such that S is a safe approximation of $Calls_G$. We can then use the atoms in S to construct specialised procedures, as shown above.

3.2 Fixpoint Definition of Call Sets

We need a notion of *canonical* atoms [12].

DEFINITION **3.3**. *Assume that each atom A has a canonical form denoted by $\|A\|$, such that $\|A\| = \|B\|$ iff A is a variant of B. If S is a set of atoms then $\|S\|$ is defined to be $\{\|A\| \mid A \in S\}$.*

DEFINITION **3.4**. *Let A be an atom, and $H \leftarrow B_1 \ldots B_n$ a clause standardised apart from A, and let θ be an mgu of A and H.. Then for $1 \leq j \leq n$, if B_j is an atom then $\|B_j\theta\|$ is said to be immediately called by A; if B_j is a negative literal $\neg C$ or a metacall containing a call to atom C, then $\|C\theta\|$ is immediately called by A.*

Note that the atoms immediately called by A do not correspond directly to atoms selected in Prolog computations of the atomic goal A, in which the body is solved from left to right.

DEFINITION **3.5**. *Given a program P, an initial atomic goal G, and a set of canonical atoms S, the set of immediate calls derived from S is defined by a function $\mathbf{C}_G : 2^{Atoms} \rightarrow 2^{Atoms}$.*

$$\mathbf{C}_G(S) = \{\|G\|\} \cup \left\{ \|B\theta\| \;\middle|\; A \in S, B\theta \text{ is immediately called by} A \right\}$$

CLAIM. $\mathbf{C}_G$ is continuous (on the lattice of sets of canonical atoms with the $\subseteq$ partial order).

LEMMA **3.6**. *$lfp(\mathbf{C}_G)$ is a safe approximation of $Calls_G$.*

PROOF. (outline) The proof is by induction on the length of Prolog computations. Let $Calls^n_G$ (a subset of $Calls_G$), be the set of atoms selected in Prolog computations of G of length at most n. By induction it can be shown that for all n there exists some k such that $\mathbf{C}^k_G(\emptyset)$ is a safe approximation of $Calls^n_G$. $lfp(\mathbf{C}_G)$ contains $\mathbf{C}^k_G$ for all k, hence $Calls_G$ is safely approximated by $lfp(\mathbf{C}_G)$. □

Our aim is now to show how to construct safe approximations of $lfp(\mathbf{C}_G)$, which in turn are safe approximations of $Calls_G$.

3.3 Finite Approximations

An important concept for the algorithm is the *most specific generalisation* (or *msg*) of a set of atoms.

DEFINITION **3.7**. *Let T be a non-empty set of atoms. A generalisation of T is an atom s such that for all $t \in T$, t is an instance of s. A most specific generalisation (or msg) of T is a generalisation u such that for all other generalisations s of T, u is an instance of s. An msg of a non-empty set of atoms always exists and is unique modulo variable renaming. We therefore write $msg(T) = \|u\|$, if u is an msg of T. Efficient algorithms for computing a most specific generalisation of a set of atoms were given by [13] and [14].*

DEFINITION **3.8**. *Let S be a set of atoms. Let p be a predicate symbol. Define S_p to be the set of atoms in S having predicate symbol p. We define $\alpha(S)$ as follows:*

$$\alpha(S) = \left\{ msg(S_p)) \;\middle|\; S_p \neq \emptyset \right\}$$

LEMMA **3.9**. *$\alpha(S)$ is a safe approximation of S.*

PROOF. By definition of *msg*. □

A set of atoms is α-canonical if it contains at most one atom for each predicate symbol and each element of the set is in canonical form. We can now define a partial order on α-canonical sets of atoms.

DEFINITION **3.10**. *Let S and T be α-canonical sets of atoms. Define $S \preceq T$ iff $[S] \subseteq [T]$.*

DEFINITION **3.11**. *Let S be an α-canonical set of atoms. The function $\mathbf{C}^\alpha_G$ is defined as follows:*

$$\mathbf{C}^\alpha_G(S) = \|\alpha(\mathbf{C}_G(S))\|$$

LEMMA **3.12**. *$lfp(\mathbf{C}^\alpha_G)$ exists and is a safe approximation of $lfp(\mathbf{C}_G)$.*

PROOF. By induction on the sequence of approximations to the fixed points. □

3.4 Termination of the Analysis

The basis of our analysis algorithm is the computation of the least fixed point of $\mathbf{C}_G^\alpha$. This is done by computing a sequence of sets of atoms, namely, $\mathbf{C}_G^\alpha(\emptyset), \mathbf{C}_G^\alpha(\mathbf{C}_G^\alpha(\emptyset)), \ldots, \mathit{lfp}(\mathbf{C}_G^\alpha)$. Clearly each element in the sequence is an α-canonical set. Now we show that the least fixed point of $\mathbf{C}_G^\alpha$ is computed in a finite number of steps.

DEFINITION **3.13**. *Let A and B be atoms. Define $A \preceq B$ iff $\exists\theta.A = B\theta$. Define $A \prec B$ iff $A \preceq B \wedge B \not\preceq A$.*

DEFINITION **3.14**. *Let t be an atom or a term. Define $s(t)$ to be the number of symbols in t, recursively defined by*

$$s(t) = \begin{cases} 1 & \text{if } t \text{ is a variable or a constant} \\ s(r_1) + \cdots + s(r_k) + 1 & \text{if } t = f(r_1, \ldots r_k) \end{cases}$$

Let the number of distinct variables in t be $v(t)$. Define $h(t) = s(t) - v(t)$. Note that $h(t) > 0$ for all non-variable t.

LEMMA **3.15**. *Let A and B be atoms. Then $A \prec B \Rightarrow h(A) > h(B)$.*

PROOF. (outline)

Assume $A \prec B$ and consider $A = B\theta$ where $\theta = \{X \mapsto T\}$. Suppose there is just one occurrence of X in B. Let $w(T)$ be the number of occurrences of variables in T which also occur in B. Then $h(A) = h(B) + h(T) + w(T)$. Then $h(A) > h(B)$ since $h(T) > 0$ and $w(T) \geq 0$; the extension of the argument for more than one occurrence of X in B, and for θ containing more than one substitution pair is straightforward. □

This shows that given an atom A_0 there is no infinite chain of atoms $A_0 \prec A_1 \prec A_2 \ldots$, since $h(A_i) > h(A_{i+1})$ for all i. It can easily be seen that successive elements of the sequence $\mathbf{C}_G^\alpha(\emptyset), \mathbf{C}_G^\alpha(\mathbf{C}_G^\alpha(\emptyset)), \ldots, \mathit{lfp}(\mathbf{C}_G^\alpha)$ that are not equal differ either by adding one or more atoms with different predicate names, or by replacing an atom A by an atom B such that $A \prec B$. Since the number of predicates in the program is finite and there are no infinite increasing $\prec$ chains the *lfp* is computed after a finite sequence of applications of $\mathbf{C}_G^\alpha$ to $\emptyset$.

3.5 Multiple Versions of Procedures

Using the function $\mathbf{C}_G^\alpha$ we get just one atom describing calls to each predicate p, and hence we define one specialised procedure for each predicate. We now present a way of deriving possibly more than one atom describing calls to p, and from them we will derive multiple procedures.

DEFINITION **3.16**. *Assume that every clause in a program has a unique identifier, say a number. We define the set of immediate choices of an atom A (denoted choice(A)), as the set of identifiers of clauses whose head unifies with A.*

DEFINITION **3.17**. *Let S be a set of atoms. We can partition S into a finite number of sets $S_{I_1}, S_{I_2}, \ldots$, where $S_{I_k} = \{A \mid A \in S, choice(A) = I_k\}$. That is, all atoms that have the same set of immediate choices I_k are in the set S_{I_k}. Define a function $\beta : 2^{Atoms} \to 2^{Atoms}$:*

$$\beta(S) = \left\{ \; msg(S_{I_k}) \;\middle|\; S_{I_k} \neq \emptyset \; \right\}$$

That is, $\beta(S)$ contains one element for each immediate choice set represented in S. Note that $choice(msg(S_{I_k})) = I_k$, so that $\beta(S)$ contains atoms with the same set of choices as those in S.

A β-canonical set of atoms is a set of canonical atoms containing at most one atom A for each set of clause identifiers I, such that $choice(A) = I$. A partial order on β-canonical sets of atoms is given by $S \preceq T$ iff $[S] \subseteq [T] \wedge \forall A \in S. \exists B \in T. choice(A) = choice(B)$. (Note that we cannot use simple downward closure to define $\preceq$ because now we want to distinguish sets $\{p(a), p(X)\}$ and $\{p(X)\}$ if $choice(p(X)) \neq choice(p(a))$).

Now we define a function analogous to $\mathbf{C}^{\alpha}_{G}$.

DEFINITION **3.18**. *Let S be a β-canonical set of atoms. Define $\mathbf{C}^{\beta}_{G} : 2^{Atoms} \to 2^{Atoms}$ to be:*

$$\mathbf{C}^{\beta}_{G}(S) = \|\beta(\mathbf{C}_G(S))\|$$

The function is continuous with respect to the partial order $\preceq$ on β-canonical sets, and the least fixed point can be found in a finite number of steps, as with $\mathbf{C}^{\alpha}_{G}$. However this time there may be more than one atom for each predicate. Atoms that match one set of clauses are not merged with atoms that match another set.

EXAMPLE **5**. *Let P be the following program:*

```
rev(nil,Ys-Ys).
rev([X|Xs],Ys-Zs) :- rev(Xs,Ys-[X|Zs]).
```

Let $G =$ `rev([U|Us],Vs-nil)`*. Then*

$$lfp(\mathbf{C}^{\alpha}_{G}) = \{\texttt{rev(Zs,Vs-Ws)}\};$$

$$lfp(\mathbf{C}^{\beta}_{G}) = \{\texttt{rev([Z|Zs],Vs-nil)},\texttt{rev(Zs,Vs-[W|Ws])}\}.$$

This second result distinguishes the calls that match only the second clause from those that match both clauses.

4 The Transformation

From the analysis of the previous section we obtain a finite set S of atoms. Using $\mathbf{C}^{\alpha}_{G}$ we get at most one atom per predicate; using $\mathbf{C}^{\beta}_{G}$ we get possibly more than one per predicate. The next stage is to define a transformed program. The transformation has two stages.

1. Derive a specialised procedure for each atom in S, as shown in Section 2.
2. Rename the goals in the bodies of the specialised clauses.

The first of these steps has already been defined. For each atom $A \in S$, let $rename(A)$ denote the atom $q(x_1 \ldots x_k)$ (with new predicate q) used to construct the specialised procedure for A (Section 2).

The second step depends on which analysis function was used.

- If $\mathbf{C}_G^\alpha$ was used, then suppose $H \leftarrow Q$ is a specialised clause. Let C be a non built-in atom occurring in Q, and let A be a variant of the atom in S with the same predicate name as C, where the clause and A share no variables. Let θ be an mgu of C and A. Replace $H \leftarrow Q$ by $(H \leftarrow Q')\theta$, where Q' is the result of replacing C in Q by $rename(A)$. Repeat until all non built-in atoms have been renamed.
- If $\mathbf{C}_G^\beta$ was used, then suppose $H \leftarrow Q$ is a specialised clause. Let C be a non built-in atom occurring in Q, and let A be a variant of the atom in S such that $choice(C) = choice(A)$. where the clause and A share no variables. Let θ be an mgu of C and A. Replace $H \leftarrow Q$ by $(H \leftarrow Q')\theta$, where Q' is the result of replacing C in Q by $rename(A)$. Repeat until all non built-in atoms have been renamed.

A special clause to handle the initial goal may be defined, so as to preserve the same external interface of the program. Let G be the initial (atomic) goal, and let $C \in S$ be the appropriate atom (with the same predicate if $\mathbf{C}_G^\alpha$ was used, or the same immediate choices if $\mathbf{C}_G^\beta$ was used). Let θ be an mgu of G and C; then the initial clause is $G\theta \leftarrow rename(C)\theta$. Instead of this initial clause, we can instead not rename the initial goal, but just produce specialised clauses for it (see example below).

EXAMPLE **6**. *Let the program and initial goal be as in the previous example. Then using* $\mathbf{C}_G^\alpha$ *we obtain the program:*

```
rev1(nil,Ys,Ys).
rev1([X|Xs],Ys,Zs) :- rev1(Xs,Ys,[X|Zs]).
```

In this program the `"-"` *symbol is eliminated. Using* $\mathbf{C}_G^\beta$ *we obtain the program:*

```
rev([U|Us],Vs,nil) :- rev2(Us,Vs,U,nil).

rev2(nil,[Y|Ys],Y,Ys).
rev2([X|Xs],Ys,Z,Zs) :- rev2(Xs,Ys,X,[Z|Zs]).
```

This contains a procedure to handle the initial goal, which matches only one clause, and another procedure to handle the more general case. In this case we did not rename the initial goal.

4.1 Correctness

Assuming for the moment the absence of `assert` and `retract` and similar procedures, the correctness of the transformation follows from the safety of the analysis, and Lemma 2.2. We sketch the proof. Suppose A is some (non built-in) call in the computation of G. Then by the safety of the analysis there is a procedure in the transformed program that handles an atom more general than A. By induction on the length of computation and using Lemma 2.2 it can be shown that the same answer substitutions are given for the renamed version of A in the transformed program.

4.2 Meta Calls and Side Effects

The presence of certain uses of side effects and meta calls can cause problems, because of the renaming transformation. One difficulty may be the interaction of `assert` with program procedures which get renamed during the transformation. Another difficulty is the presence of uninstantiated meta calls such as `call(G), bagof(X,P,S)` and the like. It is not known at transformation time what atoms may appear as the arguments to such calls. Furthermore, if these meta call arguments are constructed by predicates such as `functor` or `"=.."`, then the renaming of goals may mean that the transformed program constructs the wrong names.

The following restrictions on the uses of `assert`, `retract` and meta calls appear to ensure correctness though no proof is offered here.

1. `assert` and `retract` do not act on predicates occurring in the program text.
2. No uninstantiated meta calls occur in the transformed program text.

Many typical uses of side effect predicates (e.g. to store temporary information) can be handled.

These conditions can be checked during the analysis phase. Our approach is to print a warning during the transformation if an uninstantiated meta call or a harmful assert/retract appears. The programmer then has the responsibility of altering the source program and rerunning the analysis.

5 Implementation

An implementation of the analysis and transformation has been carried out using Prolog.

5.1 Data Structures

Canonical atoms are represented as ground atoms and are generated in the program by the Prolog built-in `numbervars`. The predicate `canonical(A)` instantiates an atom A to its canonical form. A predicate `melt(A,M)` takes a canonical term A and returns a term with fresh variables. The use of ground atoms obtained by `canonical(A)` simulates ground representation of object terms in meta-level expressions.

A set of canonical atoms is represented as an incomplete structure; updates are carried out by instantiating variables within the structure. This technique saves space because the

structure need not be copied on each iteration up to the fixed point. The form of α-canonical sets is

```
[ atom(p1,[p1(s1),p1(s2),...|_]), atom(p2,[p2(t1)...|_],
   ....
   atom(pn,[pn(r1),....|_]) |_]
```

Each `pj` is distinct predicate symbol. The element `atom(pj, [pj(u1),pj(u2)...|Tail])` represents an ascending sequence of canonical atoms `[pj(u1),pj(u2)...|Tail])`. The last atom before the variable tail is the most recent addition and is the most general; the set of canonical atoms derived from the whole structure is the set of last elements in the lists for each predicate.

The form of β-canonical sets is similar. Each `kj` in the structure is a distinct set of clause identifiers.

```
[ atom(k1,[p1(s1),p1(s2),...|_]), atom(k2,[p2(t1)...|_],
   ....
   atom(kn,[pn(r1),....|_]) |_]
```

The element `atom(kj, [pj(u1),pj(u2)...|Tail])` represents a sequence of atoms having the same immediate choices `kj`.

Elements for new predicates or immediate choices sets respectively are added by instantiating the tail of the whole structure.

The techniques of programming with incomplete data structures and canonical atoms are described in Chapter 15 of [16].

5.2 Algorithm

The core of the algorithm is expressed by the following procedure for `iterate`. The first argument of `iterate` is an incomplete structure representing a set of *canonical* atoms as described above. The second argument is a set of "new" atoms produced by the previous iteration.

```
iterate(S,nil).
iterate(S,More) :-
        More \= nil,
        immediate_calls(More,T),
        insertalpha(T,S,More1),
        iterate(S,More1).
```

On each iteration, the immediate calls resulting from new calls from the previous iteration are computed, and `insertalpha` (or `insertbeta` as the case may be) tries to add them to the structure `S`. An atom `T` is added to `S` if:

1. there is no existing atom in `S` with the same predicate name (with the same immediate choice set) as `T`, or

2. the atom at the end of the list of atoms for `T`'s predicate name (immediate choice set) is `C`, and $msg(C,T)$ is more general than `C`.

The set `More1` is the set of new canonical atoms added to `S`. Termination occurs when no more new atoms are added on some iteration.

Initialisation is performed by setting `More` to the (canonical) initial goal and calling iterate.

```
analyse(G,S) :-
        canonical(G),
        iterate(S,[G]).
```

Upon termination of `analyse` the structure `S` represents the set of atoms describing calls in the computation of `G`.

6 Results

The β algorithm was implemented and has been tested on a range of programs. The β version is more precise than the α version, and gives transformed programs having better run times and store usage. The cost is a somewhat more complex analysis algorithm and sometimes larger transformed programs. We have not yet made a thorough comparison between the performance of the two versions. It may be that the advantages of β may not be worth the extra computation cost in many cases.

EXAMPLE **7**. *The program implementing the analysis algorithm was applied to itself. (The program contained no obvious redundancies such as difference list structures). Heap consumption in the transformed program was reduced by 7%. The time taken for the transformation was approximately 15 seconds.*

EXAMPLE **8**. *A version of the above program was written containing difference list structures in some heavily used procedures. Heap consumption increased by 3%. The transformation removed this overhead completely, giving a total of 10% reduction.*

EXAMPLE **9**. *The program for computing fibonacci numbers quoted in Example 2, was transformed (as already illustrated). Heap consumption for computing* `fib(20,M)` *was reduced from 360 heap cells to 0.*

Furthermore, the fact that the arguments in the transformed program are flattened allows much more accurate mode declarations to be given. In BIMprolog `i` *means input mode and* `o` *means output mode. Unknown mode is* `?`*. In the original program (Example 2) the mode* `p(?,i,i)` *was given. In the transformed program the mode* `p1(i,o,i,i,i,i)` *was declared.*

The time for taken 1000 computations of `fib(20,X)` *with the transformed program with modes was 32% faster than the original program with modes.*

The speed-up is not of course directly due to the transformation; the point is that more accurate modes could be declared for the transformed program.

EXAMPLE **10**. *A Prolog compiler written in Prolog [17] was transformed. This program was already tuned and optimised. The heap consumption was further reduced by 0.5%, and time reduced by approximately 2%. This was a fairly large (70K) program, and it is quoted to show that the transformation was applied to large well-written programs and still gave some improvement.*

7 Related Work and Future Research

The analysis algorithm has many aspects in common with more complex analyses. It could be described in a framework for abstract interpretation such as [9],[5], or [11], but we preferred to give a self-contained description here. The problem of computing some finite description of the set of calls in a computation is common also to mode analysis and call type analysis. In [8] we defined an analysis algorithm for performing more powerful specialisations. The present work is a special case of the analysis and transformation methods defined there. It is much simpler than general mode analysis or program specialisation because we took no account of answers to calls. In reality (in Prolog) the calls of an atom in a clause body are affected by the answers to atoms to its left, but our approximation ignored this.

The idea of generating multiple versions of clauses to handle different call types derived from an abstract interpretation was proposed in [18].

The renaming method we used to flatten calls and remove redundant structures has been suggested elsewhere, but to our knowledge our presentation is the most general. Similar renamings have been incorporated into partial evaluators in [15],[7], and recently related ideas were presented in [3].

7.1 Extensions of the Method

The transformation is very generally applicable because it is simple. In general the presence of cuts, non-logical built-ins and side-effects hampers its extension by more powerful transformation methods. However our method can safely be extended in at least one respect, which we call the *first call optimisation*. The idea is as follows: suppose $H \leftarrow B_1 \ldots B_n$ is a clause, and suppose $B_1^1 \ldots B_1^k$ are the heads of clauses that match B_1. We compute $msg(B_1^1 \ldots B_1^k) = B_1'$, and compute θ, an mgu of B_1 and B_1'. The clause $H \leftarrow B_1 \ldots B_n$ can be safely replaced by $(H \leftarrow B_1 \ldots B_n)\theta$. The idea of this transformation is that if there is some argument structure that is common to all the clause heads $B_1^1 \ldots B_1^k$ that structure will be "pushed up" into the call B_1. The renaming and specialisation procedure is then applied.

This transformation can be applied safely in general only to the first call since otherwise we may get *back substitutions* that can affect the behaviour of programs with cuts, metapredicates and so on. The following example (shown in a different context in [10]) illustrates its use:

EXAMPLE **11**. *Let P be the program*

```
member(X,[X|_]).
member(X,[_|Xs]) :- member(X,Xs).
```

The clause heads for `member` *both have a list structure in the second argument. This can be pushed up into the recursive call to* `member`*. Flattening results in the following program.*

```
member1(X,X,_).
member1(X,_,[Y|Xs]) :- member1(X,Y,Xs).
```

The transformed program is more efficient than the original since the list structure is matched only once for each call, instead of twice, and because failing goals fail one step earlier.

A modification to the function generating immediate calls is all that is needed to incorporate the first call optimisation. We have done this but it is not clear whether it is generally useful. Sometimes unwanted structures get pushed into the heads of clauses. More study of the extension is needed.

More precision can be gained by using smaller partitions of sets of atoms. Instead of partitioning sets using the predicate name or the immediate choices we can look at type descriptions, or some longer computation paths like the *characteristic paths* described in [8].

Acknowledgements

We would like to thank André Marien and Bart Demoen for useful discussions, and clear explanations of what goes on inside BIMprolog!

References

[1] S. Abramsky and C. Hankin (eds.); *Abstract Interpretation of Declarative Languages*, Ellis Horwood 1987.

[2] BIMprolog Reference Manual; BIM, Kwikstraat 4, B-3078 Everberg, Belgium.

[3] F. Bry; Query Evaluation in Recursive Databases: Bottom-up and Top-down Reconciled; Report IR-KB-64, ECRC, Arabellastr. 17, D-8000 Munich, (1989).

[4] M. Bruynooghe, D. De Schreye, and B. Krekels; Compiling Control; Journal of Logic Programming 6 (1989), pp. 135-162.

[5] M. Bruynooghe; A Practical Framework for the Abstract Interpretation of Logic Program: to appear in Journal of Logic Programming.

[6] D. De Schreye, B.Martens, G. Sablon and M. Bruynooghe; Compiling Bottom-up and Mixed Derivations into Top-down Executable Logic Programs; Report CW-103, Dept. of Computer Science, Katholieke Universtiet Leuven, Belgium, (1989).

[7] Fujita, H.; An Algorithm for Partial Evaluation with Constraints; ICOT Technical Memorandum, TM-0484, August 1987.

[8] J. Gallagher and M. Bruynooghe; The Derivation of an Algorithm for Program Specialisation; to appear in Int. Conf. on Logic Programming, Jerusalem, June 1990.

[9] N.D. Jones, H. Søndergaard; A Semantics-Based Framework for the Abstract Interpretation of Prolog; Chapter 6 in [1].

[10] K. Marriott, L. Naish and J-L. Lassez; Most Specific Logic Programs; Proceedings of the Fifth International Conference and Symposium on Logic Programming; Washington, Seattle; August 1988.

[11] K. Marriott and H. Søndergaard; Semantics-Based Dataflow Analysis of Logic Programs; in "Information Processing 89" (ed. G. Ritter), North-Holland 1989.

[12] C.S. Mellish; Abstract Interpretation of Prolog Programs; Proc. 3rd ICLP, LNCS 225; Springer-Verlag; 1986; also Chapter 8 in [1].

[13] G. Plotkin; A Note on Inductive Generalisation; in *Machine Intelligence* Vol.5, (eds. B. Meltzer and D. Michie), Edinburgh University Press (1974).

[14] J.C. Reynolds; Transformational Systems and the Algebraic Structure of Atomic Formulas; in *Machine Intelligence* Vol.5, (eds. B. Meltzer and D. Michie), Edinburgh University Press (1974).

[15] S. Safra; Partial Evaluation of Concurrent Prolog and its Implications; Masters Thesis, Technical Report CS86-24, Dept. of Computer Science, Weizmann Institute, (1986).

[16] L. Sterling and E. Shapiro; *The Art of Prolog*; MIT Press (1986)

[17] Van Roy, P.; A Prolog Compiler for the PLM; Master's Report Plan II, Computer Science Division, University of California, Berkeley, 1984.

[18] W. Winsborough; Path-Dependent Reachability Analysis for Multiple Specialization; Proceedings of the North American Conference on Logic Programming, Cleveland, MIT Press, October 1989.