Automatic Transformation of Deterministic Prolog Programs to KL1

Konstantinos Varsamos

Department of Computer Science
University of Bristol
Bristol BS8 1TR, U.K.
E-mail: varsamos@acrc.bristol.ac.uk

Abstract

Concurrent logic programming languages such as KL1, using shared variables as communication channels, set up networks of communicating processes. In this way they impose directionality on the programs that they execute, assuming that specific arguments of predicates are input and the rest are output. So, to set up such networks, the Input/Output modes of the arguments of each predicate have to be already known (which process connected to a channel is its producer and which are consumers).

This paper describes a system for transforming deterministic Prolog programs to KL1, based on an effective method for mode inference.

1 Introduction

In previous years, some work [12],[13],[11] has been done on compiling Horn-clause programs (eg. Prolog programs) for all-solutions search into committed choice concurrent logic programs (eg. KL1[15] programs). The aim of this work was to automatically generate a program which returns the same set of solutions as a Horn-clause program intended for exhaustive search by means of backtracking or OR-parallelism.

In this paper, we describe a system for transforming deterministic Prolog programs to KL1. We focus more on the mode analysis of the given Prolog program and we also consider a guard analysis. Our work on mode inference is based on the ideas presented by Debray[5], but with some important innovations that we discuss. An implementation of this system has been used for the preprocessor of the CHUKL(Constraint Handling Under KL1) system[17] developed in the Computer Science Department of University of Bristol. The results obtained from this implementation show the effectiveness of the method used, when it is applied to a certain class of Prolog programs.

In [12],[13] the mode inference that is described by Ueda can be applied only to a restricted class of Prolog programs, since every predicate should always be called with ground inputs and return with ground outputs (or stream-inputs/outputs in [13]). Ueda also presented a different approach for mode inference in [14] based on mode constraints imposed by individual program clauses. We believe that for the purposes of our system global dataflow analysis is more appropriate.

Our system can easily be modified to transform deterministic Prolog programs to programs of other committed choice languages, such as Parlog[3].

1.1 Outline of the System

To transform a deterministic Prolog Program to a KL1 program, the following steps are necessary:

1. Work out the Input/Output mode for each predicate in the Prolog Program.
2. Replace the *cut* operators '!' by the *commit* operator '|', and add the *commit* operator and any *guards* wherever else it is necessary.
3. Move the output unifications in the head of each clause to the body of the clause.

The system presented in this paper, given a deterministic Prolog program together with the modes of the arguments of those predicates that may be called by the user, infers the instantiation of the arguments of all predicates when they are called and when they succeed, uses these inferences to work out the Input/Output modes of all predicates, and continues with the remaining transformation steps.

1.2 Overview of the Paper

The remainder of the paper is organized as follows: Section 2 presents a method for mode inference, based on an algorithm given by Debray in [5], but with some improvements in the abstract representations and functions used for the simulation of unification in the set of modes, which make the method more powerful. In Section 3 is described the inference of the Input/Output modes for each predicate of the given program and in Section 4 the two transformation steps for the transformation of a deterministic Prolog program to KL1: replacement of the cut operator by the commit operator and addition of guards where it is necessary, and shift of the output unification to the body of the clause. Section 5 gives some results obtained by the current implementation of the system, and finally in Section 6 conclusions and future work end the paper.

2 Mode Inference in Prolog Programs

In general Prolog programs are undirectional. An argument of a procedure can be used for input or output purposes, or for both. However, typically most of the predicates in a particular program are executed in one direction only, that is, they are always called with a particular set of their arguments instantiated (the "input" arguments) and another set uninstantiated (the "output" arguments).

Information about predicate modes finds many uses in the static optimization of logic programs. It can be used to make unification more efficient. Mode information is important in the transformation of logic programs into committed-choice languages or functional languages. It can also be used to detect deterministic and functional computations and reduce the program's search effort.

Edinburgh Prolog allows the user to specify the mode of an argument as either bound ('+'), unbound ('-') or unknown ('?') [16]. Another solution is to have a system deduce the modes from the program.

Previous work on mode inference, besides that mentioned in the introduction, was done by Mellish [7],[8],[9] who used dependencies between variables to propagate information regarding their instantiation. However, in this approach aliasing effects resulting from unification were not taken into account and built-in predicates such as '='/2 could not be handled very precisely. Reddy proposed a different approach to mode inference, in connection with work on transforming logic programs into functional languages [10], but which applied only to a restricted class of logic programs. Bruynooghe et al. [1],[2] did mode inference as an abstract interpretation problem akin to type inference. Mannilla and Ukkonen [6] used a simple mode set that is essentially the same as Reddy's, and focused on the algorithmic aspects of the analysis. Debray and Warren [4] described an approach to mode inference based on global flow analysis, related to [1],[2] and [6]. Finally, Debray [5], improving the work that was done in [4], used a set of modes that is more detailed than any other previously used and propagated instantiation information using state transformations rather than through dependencies between variables.

The analysis procedure presented in [5] is based on the principles of abstract interpretation. Some of the abstract representations that Debray used there, together with the way he handled the dependency sets of the variables in unification in the "abstract domain" in some cases, tended to be very conservative and so his algorithm did not always give the expected results. A more precise treatment is presented here. Technical terms and notations used in this section mostly follow those that Debray used in [5].

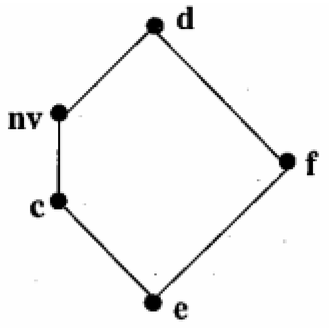
The remainder of the section assumes that the predicates in the Prolog program that is to be analyzed are static. Thus, the use of built-in predicates such as *assert* or *retract* is precluded.

2.1 Modes and Instantiation of Sets of Terms

The mode of a predicate in a program indicates how its arguments will be instantiated. For this reason we classify the terms occurring in a program into classes with regard to how they are instantiated and we consider modes over the domain of these classes Δ={**c**,**d**,**e**,**f**,**nv**}, where **c** denotes the set of closed (i.e. ground) terms, **d** ("don't-know") the set of all terms, **e** the empty set, **f** the set of free variables and **nv** the set of nonvariable terms[1].

The set Δ forms a complete lattice under inclusion:

[1] During abstract unification we will also use two other notations d_f and nv_f to specify subsets of **d** and **nv** respectively, for which we have more specific information about their dependencies.



The join operation for the ordering induced by inclusion on Δ is denoted by $\sqcup$.

Definition: The *instantiation* $\iota(T)$ of a set of terms T is given by $\iota(T) = \cap\{\delta \in \Delta \mid T \subseteq \delta\}$.

Thus, a set of terms that contains ground and nonvariable terms only will have instantiation **nv** (this is the element of Δ that best "describes" it) while a set of terms containing only ground terms will have instantiation **c**.

We now give the definition of unification over sets of terms, denoted by *s_unify*:

Definition: Given two sets of terms T_1 and T_2, *s_unify*(T_1, T_2) is the least set of terms T such that, for each pair of unifiable terms $t_1 \in T_1$, $t_2 \in T_2$ with most general unifier θ, $\theta(t_1)$ is in T.

If the result of the unification of two terms t_1 and t_2 is the term t_2, then t_2 is more instantiated than t_1. We can extend this conclusion to sets of terms and we can define an instantiation order $\trianglelefteq$ over sets of terms as follows:

Definition: Given two sets of terms T_1 and T_2, $T_1 \trianglelefteq T_2$ if and only if *s_unify*$(T_1, T_2) = T_2$.

It can be easily proved [5] that $\trianglelefteq$ is a partial order over Δ:

$$\mathbf{f} \trianglelefteq \mathbf{d} \trianglelefteq \mathbf{nv} \trianglelefteq \mathbf{c} \trianglelefteq \mathbf{e}.$$

The join operation for this order is denoted by ∇.

2.2 Abstract Representations

2.2.1 Instantiation States

During static analysis at each point in the program, we have to maintain information about variable bindings. In a call to a predicate such information can be propagated across each clause for that predicate to obtain information about variable bindings, when it returns. Thus, the behavior of a program can be summarized by specifying, at each point of the program, a description of the terms that each variable in a clause can be instantiated to at that point, together with the set of variables it can depend on (possible aliasing and sharing information between variables at that point). Such abstract representations are called *instantiation states* (or *ι-states*). To describe the set of terms that a variable in a clause can be instantiated to at any point in a program because such sets of terms can be arbitrarily large, we use elements of the mode set. In this way, we have finitely computable approximations to them suitable for purposes of static analysis.

When a clause is selected for resolution against a goal, its variables are renamed so that it is variable-disjoint with the goal. A use of a clause C in a computation where the variables of C have been renamed via a renaming substitution σ is referred to as *σ-activation* of C. The finite set of all variable names appearing in a clause C (referred as *program variables* of C) is written V_c. For any term t, *vars*(t) is the set of variables occurring in t. Then we have the following definition:

Definition: An *instantiation state* A_c at a point in a clause C is a mapping

$$A_c : V_c \to \Delta \times 2^{V_c},$$

satisfying the following: if for any variable x in V_c, $A_c(x) = \langle\delta, V\rangle$, then for any *$\sigma$-activation* of C in the computation,

(i) if $\sigma(x)$ can be instantiated to a term t at that point, then $t \in \delta$; and

(ii) if for any variable y in V_c, $\sigma(y)$ can be instantiated to a term t' at that point such that $vars(t) \cap vars(t') \neq \emptyset$, then $y \in V$.

We can extend the notion of *ι-states* to nonvariable terms, as well. Given an *ι-state* A[2] and a nonvariable term t :

- if t is a constant, then $A(t) = \langle \mathbf{c}, \emptyset \rangle$; else
- if t is a compound term $f(t_1, \ldots, t_n)$ and $A(t_1) = \langle \delta_i, D_i \rangle$, $1 \leq i \leq n$, then $A(t) = \langle \delta, D \rangle$, where $\delta = \mathbf{c}$ if $\delta_i = \mathbf{c}$, $1 \leq i \leq n$, and **nv** otherwise; and $D = \bigcup_{i=1}^{n} D_i$.

Given a *ι-state* A_c and for any variable $v \in V_c$, if $A_c(v) = \langle \delta, V \rangle$ then δ is the *instantiation* of v in A, written $inst(A_c(v))$ and V is its *dependency set*, written $deps(A_c(v))$.

Example: Consider the program

p(X) :- q(X,Y),r(Y).
q(Z,Z).
r(a).

Let the variable X be uninstantiated at a call of p and A be the *ι-state* at the program point between the literals q and r. Then $A(X) = \langle \mathbf{f}, \{X, Y\} \rangle$ and $A(Y) = \langle \mathbf{f}, \{X, Y\} \rangle$, indicating that X and Y are still uninstantiated at that point and may share a variable.

2.2.2 Instantiation Patterns

To reason about how a predicate may be called or what will return after a call, it is necessary to pass information between clauses (from the caller to the callee at the time of a call and from the callee to the caller at the time of a return). Since *instantiation states* describe the bindings of different variables in a clause and because clauses have their variables renamed before they are used in resolution, a different representation has to be used to represent calls and returns. Such a representation has to specify the instantiation of each argument when a predicate is called or a call returns, and also any sharing of variables between the arguments of the predicate. So we use *instantiation patterns* or *ι-patterns*. An *instantiation pattern* for a tuple of terms describes the bindings of different elements in the tuple and possible dependencies between them, but without any reference to variable names. We use a different definition for *ι-patterns* than that given in Debray [5]. We distinguish cases where the instantiation of an argument is ground, since it cannot share variables with any other argument of the predicate and we also distinguish those terms where there is a variable appearing more than once, from the rest of the terms, so that we can obtain more precise information about their dependencies after unification. Thus, we define *ι-pattern* as follows:

Definition: Given an *ι-state* A and an n-tuple of terms $\bar{t} = \langle t_1, \ldots, t_n \rangle$, let $A(t_i) = \langle \delta_i, V_i \rangle$ for $1 \leq i \leq n$. Then the *instantiation pattern* of $\bar{t}$ induced by A is

$$i\_pat(\bar{t}, A) = \langle \langle \delta_1, S_1 \rangle, \ldots, \langle \delta_n, S_n \rangle \rangle,$$

where

- $S_i = \emptyset$, if $\delta_i = \mathbf{c}$; or
- $S_i = \{k \mid V_i \cap V_k \neq \emptyset, k \neq i\}$, if $\delta_i \neq \mathbf{c}$ and there is no variable appearing in t_i more than once; or
- $S_i = \{k \mid V_i \cap V_k \neq \emptyset\}$, if $\delta_i \neq \mathbf{c}$ and there is a variable appearing in t_i more than once.

The kth element $\langle \delta_k, S_k \rangle$ of an *ι-pattern* $\bar{I} = \langle \langle \delta_1, S_1 \rangle, \ldots, \langle \delta_n, S_n \rangle \rangle$ is denoted by $\bar{I}[k]$. Given a tuple of terms $\bar{t} = \langle t_1, \ldots, t_n \rangle$ in an *ι-state* A, if $\bar{I}$ is the *ι-pattern* $i\_pat(\bar{t}, A)$, then for $1 \leq i \leq n$, the *instantiation* of the kth element (written $inst(\bar{I}[k])$) is δ_k, while the *share set* of $\bar{I}[k]$ (written $share(\bar{I}[k])$) which gives the indices of the elements in $\bar{t}$ that t_k shares variable with, is the set S_k.

Example: Consider a call $p(f(X, X), g(a), h(X, Y), Z)$ in an *ι-state* A :

$$A = \{X \rightarrow \langle \mathbf{f}, \{X\} \rangle,\ Y \rightarrow \langle \mathbf{f}, \{Y, Z\} \rangle,\ Z \rightarrow \langle\ \mathbf{nv}, \{Y, Z\} \rangle\}.$$

[2] Because V_c is fixed once C has been specified, when there is no scope for confusion we can drop the subscript C from the name of the *ι-state*.

This call is represented by the *ι-pattern*

$$\iota\_pat(\langle f(X,X), g(a), h(X,Y), Z\rangle, A) \;=\; \langle\langle \mathbf{nv}, \{1,3\}\rangle, \langle \mathbf{c}, \emptyset\rangle, \langle \mathbf{nv}, \{1,4\}\rangle, \langle \mathbf{nv}, \{3\}\rangle\rangle.$$

This indicates that the first argument of the call is a nonvariable term that shares a variable with the third argument and there is a variable that occurs more than once in this term; the second argument is a ground term; the third argument is a nonvariable term that shares variables with the first and the fourth argument; and the fourth argument is a nonvariable term that shares a variable with the third argument.

The *ι-pattern* of a call to a predicate is referred to as a *calling pattern*, while the *ι-pattern* of a return from a call is referred to as the *success pattern* for that call.

Finally, *instantiation patterns* of length n (for any n) can be ordered element-wise by inclusion in a straightforward manner. The least upper bound of two *ι-patterns* is then

$$\langle\langle\delta_{11}, S_{11}\rangle, \ldots, \langle\delta_{1n}, S_{1n}\rangle\rangle \sqcup \langle\langle\delta_{21}, S_{21}\rangle, \ldots, \langle\delta_{2n}, S_{2n}\rangle\rangle = \langle\langle\delta_{11}\sqcup\delta_{21}, S_{11}\cup S_{21}\rangle, \ldots, \langle\delta_{1n}\sqcup\delta_{2n}, S_{1n}\cup S_{2n}\rangle\rangle.$$

2.3 Abstract Unification

At the time of a call to a predicate the arguments of the call are unified with those in the head of a clause for that predicate, and at the time of a return from a call they are "back-unified" to propagate the effects of the return to the caller.

To simulate unification over *ι-states* we have to consider the instantiation "inherited" by a variable x occurring in a term during unification. For this reason, we define the function *inherited_inst*:

Definition: Let t_1 be a term in an *ι-state* A, with $inst(A(t_1)) = \delta_1$, and let x be a variable occurring in t_1. The instantiation inherited by x when t_1 is unified with a term whose instantiation is δ_2 is given by

$inherited\_inst(A, x, t_1, \delta_2) \equiv$
if $x = t_1$ **then** δ; **else if** $\delta_2 = \mathbf{f}$ **then** $inst(A(x))$; **else** $sub\_inst(\delta)$;

where $\delta = \delta_1 \nabla \delta_2$.

The function *sub_inst* describes the instantiation of the set of all proper subterms of all the elements of a set of terms, and is given by the following table:

δ	$sub\_inst(\delta)$
d	**d**
nv	**d**
c	**c**
f	**e**
e	**e**

Example: Let X be a free variable in a *ι-state* A and a term $f(X)$ being unified with a ground term. The instantiation of X after the unification is $inherited\_inst(A, X, f(X), \mathbf{c}) \equiv sub\_inst(\mathbf{nv}\nabla\mathbf{c}) = sub\_inst(\mathbf{c}) = \mathbf{c}$[3].

The function *inherited_inst* gives the instantiation of the resulting term after unification ignoring any aliasing effects. Of course, any dependencies between variables must also be taken into account.

During analysis, only one of the two tuples of terms that are to be unified is known. The other one is represented by an *ι-pattern*. So, we need a function which will indicate for every variable appearing in the known tuple, the dependencies induced by the tuple represented by the *ι-pattern*, after unification. Let $\bar{t} = \langle t_1, \ldots, t_n\rangle$, and for any any variable v, let $occ(v, \bar{t}) = \{j \mid v \in vars(t_j)\}$ be the indices of the elements of $\bar{t}$ in which v occurs. Then, we define the function *inherited_dep* :

Definition: Consider an *n-tuple* of terms $\bar{t} = \langle t_1, \ldots, t_n\rangle$, an *ι-pattern* $\bar{I}$ of length n and a variable $x \in \cup(vars(t_i))$, $1 \le i \le n$. The indices of the elements t_i that x may depend on any variable in $vars(t_i)$, taking into account only any sharing of variables in the tuple of terms represented by $\bar{I}$, are given by

$$inherited\_dep(x, \bar{t}, \bar{I}) = \{j \mid k \in occ(x, \bar{t}), j \in share(\bar{I}[k])\}.$$

Debray [5] uses a function called *c_closure* to indicate sharing of variables after unification. By using this function he does not take advantage of the information that we have about the occurrences of variables in the known tuple of terms. By considering the *instantiation patterns* for both terms, his treatment

[3]This is sound in the sense that, if the unification were to fail, the resulting instantiation of X would be **e**, which is contained in **c**

becomes conservative and general at this point. By using the function *inherited_dep*, we exploit any information that we have about the two tuples of terms that unify.

Example: Let X, Y, Z, U, V be free variables not aliased to any other variable and consider the unification of the tuple of terms $\bar{t_1} = \langle X, f(Y), Y, Z\rangle$ with the tuple $\bar{t_2} = \langle U, f(U), V, V\rangle$ represented during static analysis by the *ι-pattern* $\bar{I_2} = \langle\langle f, \{2\}\rangle, \langle nv, \{1\}\rangle, \langle f, \{4\}\rangle, \langle f, \{3\}\rangle\rangle$. Then, $occ(Y, \bar{t_1}) = \{2,3\}$ and $inherited\_dep(Y, \bar{t_1}, \bar{I_2}) = \{1,4\}$, indicating that after unification, because of sharing of variables in $\bar{t_2}$, Y may depend on any variable appearing in the first and fourth argument.

2.3.1 The Basic Functions of Abstract Unification

Now we are in a position to describe *abstract unification*. We decompose it into the following three functions:

- *init_unify* derives any changes in variable instantiations or their dependency sets resulting from unification, taking into account any sharing of variables in the tuple presented by the *ι-pattern*;
- *propagate_inst* propagates these changes to instantiations of variables by taking into account dependencies between variables;
- *normalize* "cleans up" the dependency sets of variables using information about variable groundness and returns the resulting *ι-state* after unification.

The function *init_unify*: The function *init_unify* returns a pair consisting of an *ι-state* and a set of program variables. The *ι-state* reflects the effects of unification on variables in a clause, considering the instantiations inherited from each position in which a variable occurs in the known tuple of terms and propagating any sharing information given by the tuple represented by the *ι-pattern*.The dependency set of a variable whose instantiation in this stage does not become ground is obtained by taking the union of its previous dependency set with the union of the dependency sets of all the elemenents of the known tuple that their indices are returned by *inherited_dep*. The set of variables returned by *init_unify* consists of those variables whose instantiations or dependency sets changed in this step.

If a term t_{1i} from the known tuple $\bar{t_1}$ is unified with the term t_{2i} from the unkown tuple $\bar{t_2}$, where $t_{2i} = x$ and x is a free variable, and x also appears in term t_{2j} $(j \neq i)$ in $\bar{t_2}$ where t_{2j} is unified with the ground term t_{1j} from $\bar{t_1}$, then t_{1i} becomes ground after unification.

If a term t_{1i} from the known tuple $\bar{t_1}$ is unified with the term t_{2i} from the unknown tuple $\bar{t_2}$, where $t_{2i} = x$ and x is a free variable, and there is also a term $t_{2j} = x$ $(j \neq i)$ in $\bar{t_2}$ where t_{2j} is unified with the nonvariable term t_{1j} from $\bar{t_1}$, then t_{2i} becomes a nonvariable term after unification and depends totally on every variable appearing in t_{1j}. If every variable appearing in t_{1j} becomes ground later then t_{2i} will become ground as well.

The function *init_unify* to handle precisely such cases as the above and to be able to take advantage of the specific information about instantiation and dependencies that can be obtained from sharing of variables in the term represented by the *ι-pattern*, for some terms, has to distinguish these terms from the rest of the terms of the same instantiation. For this reason, during abstract unification such terms are denoted by $(in)_f$ where (in) is their instantiation, $(in) \in \{\mathbf{nv}, \mathbf{d}\}$ and we use the function *init_unify* instead of *a_unify_init* that Debray uses in [5]. The join operation for the partial order $\trianglelefteq$ which also handles $(in)_f$ is written ∇_f and is defined as ∇ including also the cases:

δ_1	δ_2	$\nabla_f(\delta_1, \delta_2)$
nv_f	f	nv_f
nv_f	d	nv
nv_f	nv	nv
nv_f	c	c
nv_f	e	e
nv_f	nv_f	nv_f
nv_f	d_f	nv_f
d_f	f	d_f
d_f	d	d
d_f	nv	nv
d_f	c	c
d_f	e	e
d_f	d_f	d_f

Thus, if the instantiation of any one of two terms is free, then the instantiation resulting from their join operation is free, as well. If the instantiation of any one of two terms is ground or "empty", then the instantiation resulting from their join operation is ground or "empty", respectively. If the instantiation of both terms are in $\{nv_f, d_f\}$ then we can still have specific information about the instantiation and dependencies of variables after unification and so the resulting instantiation of the join operation of the two terms is in $\{nv_f, d_f\}$. Otherwise, the resulting instantiation is the corresponding instantiation from Δ.

Now, we can define *init_unify* as follows:

Definition: Consider an *ι-state* A_0 for a clause C, an n-tuple of terms $\bar{t} = \langle t_1, \ldots, t_n \rangle$ and an *ι-pattern* $\bar{I} = \langle \langle \delta_1, S_1 \rangle, \ldots, \langle \delta_n, S_n \rangle \rangle$. Then, $init\_unify(A_0, \bar{t}, \bar{I}) = \langle A_1, V_1 \rangle$, where A_1 is an *ι-state* for C and $V_1 \subseteq V_c$, is defined as follows:

- for any variable v in V_c,
  - if $occ(v, \bar{t}) \neq \emptyset$, then $A_1(v) = \langle \delta, D \rangle$, where
    - if $\nabla\{inherited\_inst(A_0, v, t_i, \delta_i) \mid i \in occ(v, \bar{t})\}$ =c then δ =c and $D = \emptyset$; else
    - if there is a $k \in occ(v, \bar{t})$ such that $inst(\bar{I}[k])$ =f and for any $j \in share(\bar{I}[k])$, $A_0(t_j) = \langle \mathrm{c}, \emptyset \rangle$ then δ =c and $D = \emptyset$; else
    - $\delta = \nabla_f\{m_i \mid i \in occ(v, \bar{t})\}$, where
      - if $inst(\bar{I}[i])$ =f, $j \in share(\bar{I}[i])$, $inst(\bar{I}[j])$ =f and $inst(A_0(t_j))$ =**nv** then $m_i = (inherited\_inst(A_0, v, t_i, \mathrm{nv}))_f$; else
      - $m_i = inherited\_inst(A_0, v, t_i, \delta_i))$,

      and $D = deps(A_0(v)) \cup (\cup\{deps(A_0(t_j)) \mid j \in inherited\_dep(v, \bar{t}, \bar{I})\})$;
  - if $occ(v, \bar{t}) = \emptyset$, then $A_1(v) = A_0(v)$;
- $V_1 = \{v \in V_c \mid A_0(v) \neq A_1(v)\}$.

Example 1: Given an *ι-state* $A_0 = \{X \rightarrow \langle \mathrm{f}, \{X\} \rangle, Y \rightarrow \langle \mathrm{f}, \{Y\} \rangle\}$, consider the unification of the tuple of terms $\bar{t_1} = \langle X, Y, f(Y), f(a) \rangle$ with the tuple $\langle U, U, V, V \rangle$ represented in static analysis by the *ι-pattern* $\bar{I} = \langle \langle \mathrm{f}, \{2\} \rangle, \langle \mathrm{f}, \{1\} \rangle, \langle \mathrm{f}, \{4\} \rangle, \langle \mathrm{f}, \{3\} \rangle \rangle$.

Then, $init\_unify(A_0, \bar{t}, \bar{I}) = \langle A_1, V_1 \rangle$, where $A_1 = \{X \rightarrow \langle \mathrm{f}, \{X, Y\} \rangle, Y \rightarrow \langle \mathrm{c}, \emptyset \rangle\}$ and $V_1 = \{X, Y\}$, indicating that the instantiation of Y becomes ground after unification and X is a free variable that may depend on Y.

Example 2: Given an *ι-state* $A_0 = \{X \rightarrow \langle \mathrm{f}, \{X\} \rangle, Y \rightarrow \langle \mathrm{f}, \{Y\} \rangle, Z \rightarrow \langle \mathrm{f}, \{Z\} \rangle\}$, consider the unification of the tuple of terms $\bar{t_1} = \langle f(Y, Z), X, Z \rangle$ with the tuple $\langle U, U, a \rangle$ represented in static analysis by the *ι-pattern* $\bar{I} = \langle \langle \mathrm{f}, \{2\} \rangle, \langle \mathrm{f}, \{1\} \rangle, \langle \mathrm{c}, \emptyset \rangle \rangle$.

Then, $init\_unify(A_0, \bar{t}, \bar{I}) = \langle A_1, V_1 \rangle$, where $A_1 = \{X \rightarrow \langle nv_f, \{X, Y, Z\} \rangle, Y \rightarrow \langle \mathrm{f}, \{Y, X\} \rangle, Z \rightarrow \langle \mathrm{c}, \emptyset \rangle\}$ and $V_1 = \{X, Y, Z\}$, indicating that after unification X becomes a nonvariable term depending totally on Y and Z, Y remains unbound but may depend on X and Z becomes a ground term.

The function *propagate_inst*: The next step is to propagate these changes derived by *init_unify* to instantiations of variables by taking into account dependencies between variables. This is done by the function *propagate_inst*.

To infer the instantiation of a variable x after unification, we have to know the instantiation of every variable that x depends on. In the same way to infer the instantiations of the variables that x depends on, we have to know the instantiation of every variable that they depend on, and so on. Thus, the function *propagate_inst* has to be defined recursively, in order to propagate any change in the instantiation of a variable to the variables that it depend on, until a fixed point has been reached (i.e. there are no other changes to propagate).

Suppose the *ι-state* obtained by *unify_init* is A, the set of variables inferred to be affected by unification is V and consider a variable x with $A(x) = \langle \delta, D' \rangle$. If $D' = \{x\}$, then the new instantiation of x is $\delta' = \delta$ otherwise $D = D' - \{x\}$ and there are the following possibilities:

1. If δ =e, then execution cannot reach that point, so δ' must also be e.
2. If δ =f, then
   - if all variables in $V \cap D$ are still uninstantiated after *init_unify*, or if those that are instantiated have instantiation nv_f or d_f, then the instantiation of x is unaffected by the unification and δ' =f;

- if there is at least one variable in $V \cap D$ whose instantiation changes to c after unification and there is no variable in $V \cap D$ that has instantiation nv or d, then δ' =c;
- otherwise, some variable that x depends on has become instantiated but we are not able to infer precisely the new instantiation of x, so δ' =d.

3. If δ =c, then x is a ground term and cannot be affected by aliasing effects, so δ' =c;
4. If δ =nv, then we are not able to infer with greater precision the instantiation of x and, so δ' =nv.
5. If $\delta = nv_f$, then
   - if every variable in $V \cap D$ is ground after unification, then x becomes ground as well, so δ' =c;
   - otherwise, x remains a nonvariable term, $\delta' = nv_f$.
6. If δ =d, then we are not able to infer precisely the instantiation of x and, so δ' =d.
7. If $\delta = d_f$, then
   - if every variable in $V \cap D$ is ground after unification, then x becomes ground as well, so δ' =c;
   - otherwise, we are not able to infer precisely the instantiation of x and, so $\delta' = d_f$.

Now, it can be more obvious why the notations nv_f and d_f are used. When the instantiation of a term is nv or d nothing more about its instantiation can be inferred using dependency information, while for any term that has instantiation nv_f or d_f any information about variable groundness in its dependency set can be used to have a more precise inference.

Based on the above case analysis, the function *propagate_inst*, taking advantage of the use of the notations nv_f and d_f, can be recursively defined as follows:

Definition: Let A_0 be an *ι-state* defined on a set of program variables V_c and $V \subseteq V_c$. Then $propagate\_inst(\langle A_0, V\rangle) = A$ is an *ι-state* defined on V_c as follows:

- for every $x \in V_c$, if $A_0(x) = \langle \delta, D'\rangle$ and $D' = \{x\}$, then $A_1(x) = A_0(x)$; else $D = D' - \{x\}$ and
  - if δ =f and there is a variable y in $V \cap D$ such that $inst(A_0(y)) = \delta_1$, where $\delta_1 \in \{\text{nv},\text{d}\}$, then $A_1(x) = \langle \text{d}, D'\rangle$;
  - if δ =f, there is a variable y in $V \cap D$ such that $inst(A_0(y))$ =c and for every other variable $z \in V \cap D$, $z \neq y$, $inst(A_0(z)) = \delta_1$, where $\delta_1 \in \{\text{f},\text{c},nv_f,d_f\}$, then $A_1(x) = \langle \text{c}, \emptyset\rangle$;
  - if $\delta = nv_f$ and every variable y in $V \cap D$ has $inst(A_0(y))$ =c, then $A_1(x) = \langle \text{c}, \emptyset\rangle$;
  - if $\delta = d_f$ and every variable y in $V \cap D$ has $inst(A_0(y))$ =c, then $A_1(x) = \langle \text{c}, \emptyset\rangle$;
  - otherwise, $A_1(x) = A_0(x)$;
- if $A_0 = A_1$, then $A = A_0$ else $propagate\_inst(\langle A_1, V\rangle) = A$.

Example 1: Consider the previous example 1. Since, after *init_unify* X may only depend on Y and the instantiation of Y has become ground, the instantiation of X must become ground as well. This is what *propagate_inst* infers:
$propagate\_inst(\langle\{X \to \langle \text{f}, \{X,Y\}\rangle, Y \to \langle \text{c}, \emptyset\rangle\}, \{X,Y\}\rangle) = \{X \to \langle \text{c}, \emptyset\rangle, Y \to \langle \text{c}, \emptyset\rangle\}$.

Example 2: For the previous example 2, *propagate_inst* does not have any effect on the resulting *ι-state*. $propagate\_inst(\langle A_1, V_1\rangle) = A_1$, where
$A_1 = \{X \to \langle nv_f, \{X,Y,Z\}\rangle, Y \to \langle \text{f}, \{Y,X\}\rangle, Z \to \langle \text{c}, \emptyset\rangle\}$.

The function *normalize*: The final step is to "clean up" the dependency sets of variables and return the resulting *ι-state* after unification. If a variable x has instantiation c, then x can be deleted from the dependency set of any other variable. Also, there is no need to keep the notations nv_f and d_f, since any particular information about the instantiations of variables has been considered. Thus, nv_f and d_f can be replaced by nv and d respectively. For these purposes the function *normalize* is defined as follows[4]:

Definition: Let A_0 be an *ι-state* defined on a set of variables V_c and let $ground(A_0) = \{v \in V_c \mid inst(A(v)) = \text{c}\}$. Then $normalize(A)$ is an *ι-state* A_1, with domain V_c, defined as follows: for each $x \in V_c$, if $A_0(x) = \langle \delta, D\rangle$, then

[4]The dependency set of the variables whose instantiation is ground was set to $\emptyset$ right after the instantiation of the variable had been inferred, and so there is no need for *normalize* to consider that as in [5].

- $inst(A_1(x)) = \delta'$, where
  - if $\delta \in \{$f,d,nv,c,e$\}$ then $\delta' = \delta$; else
  - if $\delta = nv_f$ then $\delta' =$nv; else
  - if $\delta = d_f$ then $\delta' =$d; and
- $deps(A_1(x)) = D'$, where $D' = D - ground(A_0)$.

Examples: For the previous example 1, *normalize* does not have any effect on the resulting *ι-state*. For the second example, since $inst(A_1(Z)) =$c, Z can be deleted from the dependency set of X. Also, the instantiation of X can be denoted by **nv**:
$normalize(\{X \to \langle nv_f, \{X,Y,Z\}\rangle, Y \to \langle \mathrm{f}, \{Y,X\}\rangle, Z \to \langle \mathrm{c}, \emptyset\rangle\}) = \{X \to \langle \mathrm{nv}, \{X,Y\}\rangle, Y \to \langle \mathrm{f}, \{Y,X\}\rangle, Z \to \langle \mathrm{c}, \emptyset\rangle\}$.

The main function *update_i_state*: We can now define a function *update_i_state* that simulates unification in the "abstract domain":

Definition: If A_0 is an *ι-state* defined on a set of variables V_c, $\bar{t}$ is an n-tuple of terms all of whose variables are in V_c, and $\bar{I}$ ia an *ι-pattern* of length n, then $update\_i\_state(A_0, \bar{t}, \bar{I})$ is an *ι-state* defined on V_c, by:

$$update\_i\_state(A_0, \bar{t}, \bar{I}) = normalize(propagate\_inst(init\_unify(A_0, \bar{t}, \bar{I}))).$$

Thus, for the previous two examples, the resulting *ι-states* after unification are:
$update\_i\_state(\{X \to \langle \mathrm{f}, \{X\}\rangle, Y \to \langle \mathrm{f}, \{Y\}\rangle\}, \langle X, Y, f(Y), f(a)\rangle, \langle\langle \mathrm{f}, \{2\}\rangle, \langle \mathrm{f}, \{1\}\rangle, \langle \mathrm{f}, \{4\}\rangle, \langle \mathrm{f}, \{3\}\rangle\rangle)$
$= \{X \to \langle \mathrm{c}, \emptyset\rangle, Y \to \langle \mathrm{c}, \emptyset\rangle\}$, and
$update\_i\_state(\{X \to \langle \mathrm{f}, \{X\}\rangle, Y \to \langle \mathrm{f}, \{Y\}\rangle, Z \to \langle \mathrm{f}, \{Z\}\rangle\}, \langle f(Y,Z), X, Z\rangle, \langle\langle \mathrm{f}, \{2\}\rangle, \langle \mathrm{f}, \{1\}\rangle, \langle \mathrm{c}, \emptyset\rangle\rangle)$
$= \{X \to \langle \mathrm{nv}, \{X,Y\}\rangle, Y \to \langle \mathrm{f}, \{Y,X\}\rangle, Z \to \langle \mathrm{c}, \emptyset\rangle\}$.
Debray in [5] cannot obtain such precise results.

2.4 Propagation of Flow Information

For the analysis, the user has to specify which predicates may be called from the outside and for each such exported predicate, instantiation patterns that describe how it may be called. Thus, the module being analyzed is of the form $\langle P, EXPORTS(P)\rangle$, where P is the Prolog program and $EXPORTS(P)$ is the predicates p that are exported by the program[5].

During static analysis, given a calling pattern for a predicate, only those success patterns will be considered admissible that might actually correspond to computations for that predicate, starting with a call described by that calling pattern. Similarly, only some of the possible calling patterns will in fact be encountered during computations and therefore, during static analysis not all calling paterns for a predicate will be admissible. For $n \geq 0$, let Γ_n denote the set of pairs $\Delta \times 2^{\{1,\ldots,n\}}$. With each *n-ary* predicate p in a program we associate a set $CALLPAT(p) \subseteq (\Gamma_n)^n$, the set of *admissible calling patterns*, and a relation $SUCCPAT(p) \subseteq (\Gamma_n)^n \times (\Gamma_n)^n$, associating with each calling pattern an *admissible success pattern*. Given a module $\langle P, EXPORTS(P)\rangle$, these sets are defined to be the smallest sets satisfying the following:

- If $\langle p, \bar{I}\rangle \in EXPORTS(P)$, then $\bar{I}$ is in $CALLPAT(p)$.
- Let q_0 be a predicate in the program, $\bar{I}_c \in CALLPAT(q_0)$, and let there be a clause in the program of the form

  $$q_0(\bar{X}_0) : -q_1(\bar{X}_1), \ldots, q_n(\bar{X}_n).$$

  Let the *ι-state* at the point immediately after the literal $q_j(\bar{X}_j)$, $0 \leq j \leq n$, be A_j, where A^{init} is the initial *ι-state* of the clause; $A_0 = update\_i\_state(A^{init}, \bar{X}_0, \bar{I}_c)$; then, for $1 \leq i \leq n$, $cp_i = i\_pat(\bar{X}_i, A_{i-1})$ is in $CALLPAT(q_i)$; and if $\langle cp_i, sp_i\rangle$ is in $SUCCPAT(q_i)$, then $A_i = update\_i\_state(A_{i-1}, \bar{X}_i, sp_i)$.
- The success pattern for the clause is given by $\bar{I}_s = i\_pat(\bar{X}_0, A_n)$, and $\langle \bar{I}_c, \bar{I}_s\rangle$ is in $SUCCPAT(q_0)$.

[5] $EXPORTS(P)$ may contain more than one entry for a predicate if it can be called with different calling patterns.

2.4.1 The Algorithm

We use the same algorithm as Debray used, which can be found in [5]. The global data structures maintained by the algorithm consist of a worklist, $NEEDS\_PROCESSING$, which contains the predicates that have to be processed, and the tables $CALLPAT(p)$ and $SUCCPAT(p)$, for each predicate p in the program. Initially, $NEEDS\_PROCESSING$ contains the set of predicates appearing in $EXPORTS(P)$; $CALLPAT(p)$ contains the calling patterns for p that are specified in $EXPORTS(P)$ if p is an exported predicate, otherwise it is empty, and $SUCCPAT(p)$ is empty for each predicate p in the program.

During analysis, if a new success pattern is found for a predicate p, every predicate that calls p has to be reanalyzed. For this reason, before analysis begins, the call graph of the program is constructed, and is used to compute, for each predicate p, the set $CALLERS(p)$ of the predicates that call p.

The analysis begins with the predicates and their calling patterns specified in $EXPORTS(P)$. For each such predicate and calling pattern, the clauses for the predicate are analyzed by propagating instantiation states across the literals of each clause. This yields instantiation patterns describing how the clause may succeed. In order to propagate instantiation states across these clauses, it becomes necessary to analyze predicates called by the exported predicates, and so on. This is repeated until no new calling or success patterns can be obtained for any predicate in the program, at which point the analysis terminates.

The algorithm returns the tables $CALLPAT(p)$ and $SUCCPAT(p)$ giving the admissible calling and success patterns, for each predicate p in the program P, with respect to the set of exported predicates and external calling patterns specified in $EXPORTS(P)$. To compute the mode of each predicate p when it is called we use the set of calling patterns $CALLPAT(p)$:

If $\sqcup CALLPAT(p)$ is $\langle\langle\delta_1, S_1\rangle, \ldots, \langle\delta_n, S_n\rangle\rangle$, then the mode of p is $\langle\delta_1, \ldots, \delta_n\rangle$.

2.4.2 Example

To illustrate the method described above, we give the following example, taken from [5]. It illustrates the handling of aliasing and shows the effect of our improvements on [5].

Example: Consider the program

$p(X,Y) : -q(X,Y), r(X), s(Y).$
$q(Z,Z).$
$r(a).$
$s(\_).$

Assume that the user has specified that p is the only exported predicate and is called with the following calling pattern:

$$\langle\langle \mathrm{f}, \emptyset\rangle, \langle \mathrm{f}, \emptyset\rangle\rangle.$$

Initially, the table $CALLPAT(p)$ contains only $\langle\langle \mathrm{f}, \emptyset\rangle, \langle \mathrm{f}, \emptyset\rangle\rangle$, and the table $SUCCPAT(p)$ is empty as well as every table for the rest of the predicates.

Processing the clause for p, let the *ι-state* after the kth literal (counting the head as the 0th literal) be A_k. The *ι-state* resulting from unification of the head with the call is

$$A_0 = \{X \rightarrow \langle \mathrm{f}, \{X\}, Y \rightarrow \langle \mathrm{f}, \{Y\}\rangle\}.$$

The calling pattern for q is therefore $\langle\langle \mathrm{f}, \emptyset\rangle, \langle \mathrm{f}, \emptyset\rangle\rangle$ and is added to $CALLPAT(q)$.

The clause for q is then analyzed. The *ι-state* resulting from unification of the head of the clause for q with the call is $A_q = \{Z \rightarrow \langle \mathrm{f}, \{Z\}\}$ and so the success pattern of the call to q is $\langle\langle \mathrm{f}, \{2\}\rangle, \langle \mathrm{f}, \{1\}\rangle\rangle$. Thus, the pair $(\langle\langle \mathrm{f}, \emptyset\rangle, \langle \mathrm{f}, \emptyset\rangle\rangle, \langle\langle \mathrm{f}, \{2\}\rangle, \langle \mathrm{f}, \{1\}\rangle\rangle)$ is added to $SUCCPAT(q)$. The *ι-state* after the literal $q(X,Y)$ in the clause for p is

$$A_1 = \{X \rightarrow \langle \mathrm{f}, \{X,Y\}, Y \rightarrow \langle \mathrm{f}, \{Y,X\}\rangle\}.$$

The calling pattern for r is $\langle\langle \mathrm{f}, \emptyset\rangle\rangle$, and is added to $CALLPAT(r)$. Its success pattern is infered to be $\langle\langle \mathrm{c}, \emptyset\rangle\rangle$ and so the pair $(\langle\langle \mathrm{f}, \emptyset\rangle\rangle, \langle\langle \mathrm{c}, \emptyset\rangle\rangle)$ is added to $SUCCPAT(r)$.

Since the instantiation of X became ground and Y is uninstantiated, depending on X, the instantiation of Y becomes ground as well. Thus, the *ι-state* after the literal $r(X)$ in the clause for p is

$$A_2 = \{X \rightarrow \langle \mathrm{c}, \emptyset, Y \rightarrow \langle \mathrm{c}, \emptyset\rangle\}.$$

The calling pattern for s is therefore $\langle\langle \mathrm{c}, \emptyset\rangle\rangle$, and is added to $CALLPAT(s)$. Its success pattern is the same and, so the pair $(\langle\langle \mathrm{c}, \emptyset\rangle\rangle, \langle\langle \mathrm{c}, \emptyset\rangle\rangle)$ is added to $SUCCPAT(s)$.

The ι-*state* A_3 at the end of the clause for p is the same as $A_2 = \{X \to \langle \mathbf{c}, \emptyset, Y \to \langle \mathbf{c}, \emptyset \rangle\}$, and therefore the success pattern of the call to p is $\langle\langle \mathbf{c}, \emptyset\rangle, \langle \mathbf{c}, \emptyset\rangle\rangle$ and the pair $(\langle\langle \mathbf{f}, \emptyset\rangle, \langle \mathbf{f}, \emptyset\rangle\rangle, \langle\langle \mathbf{c}, \emptyset\rangle, \langle \mathbf{c}, \emptyset\rangle\rangle)$ is added to $SUCCPAT(p)$.

For the same example, the calling pattern that Debray infers in [5] for s is $\langle\langle \mathbf{d}, \{1\}\rangle\rangle$, and the success pattern inferred for p is $\langle\langle \mathbf{c}, \emptyset\rangle, \langle \mathbf{d}, \{2\}\rangle\rangle$, which are not as precise as our inferences.

3 Inference of the Input/Output Modes in a Prolog Program

To infer the Input/Output modes in a Prolog program, we can first use the mode inference system described in the previous section to infer the calling modes of the arguments of each predicate. After having inferred these modes we have to classify the arguments in the heads of the clauses for a predicate into input arguments and output arguments. So, for every clause for that predicate, we check the instantiation of each argument in its head and compare it with the calling mode that we have already inferred for that argument. To decide whether the mode of an argument is input or output, we make the following assumptions:

Inferred calling mode of argument	Instantiation of argument in the clause head	Input/Output mode
c	f	Input
c	c	Input
c	nv	Input
nv	f	Input
d	f	Input
f	f	Output
f	c	Output
f	nv	Output
nv	c	Output
nv	nv	Output
d	c	Output
d	nv	Output

To infer the Input/Output modes for each predicate, we assume the following: If an argument in the head of a clause for a predicate is inferred to be an output argument, then it is assumed that every argument in the same position in the head of any clause for that predicate is an output argument. Otherwise, we assume that every argument in that position in the head of any clause for that predicate is an input argument. Thus, for the case where we infer different Input/Ouput modes in the same position in the head of different clauses for a predicate, we handle the argument in this position as an output argument for this predicate. So the output unifications that are imposed in this position in the head of some clauses will later be moved to the body of the clauses and a possible deadlock will be avoided.

If the inferred calling mode of an argument is ground or the instantiation of the argument in the clause head is free then we assume that the mode of the argument is *Input*, otherwise it is *Output*. The case $(\mathbf{f}, \mathbf{f})$ is excluded from these assumptions. If an argument in a clause head is a free variable and the inferred calling mode for this argument is free as well, then this variable may become instantiated to a ground or non-variable term during the execution of the clause, and so it can be assumed that it is an output argument.

The mode information that we have is not always enough to infer the directionality of an argument. This happens in the cases $(\mathbf{d}, \mathbf{nv})$ and $(\mathbf{nv}, \mathbf{nv})$, where the first element of the pairs is the inferred mode of an argument and the second element is the instantiation of that argument in the head of a clause. For these cases, the above assumptions give a sound result, by inferring an output mode for such an argument, so that the output unification that this argument imposes will later be moved to the body of the clause and a possible deadlock will be avoided[6].

For the case $(\mathbf{d}, \mathbf{f})$, where the corresponding argument in the head of a clause for a predicate is a free variable, there is a possibility for that variable to appear more than once, both as an input and as an output argument in the head of the clause, so that during unification of the head of the clause with a call to that predicate, the variable becomes instantiated in both position. In such cases, most of the times, the mode of the argument in the output position is inferred to be output from the other arguments in the same position in the rest of the clauses for that the predicate. However, this is a point that we have to be

[6]This is what we regard a sound result.

aware of, because the resulting transformed program may deadlock in KL1. Of course we may sacrifice partial accuracy in favour of soundness by changing these assumptions to infer the output mode, so that a possible deadlock would be avoided.

For the rest of the cases the above assumptions give both accurate and sound results.

Example: Consider the clauses for the *append*/3 predicate :

$append([], L, L).$
$append([H|L1], L2, [H|L3]) : -append(L1, L2, L3).$

Using the method described in the previous section, we infer that for the calling pattern $\langle \mathbf{c}, \mathbf{c}, \mathbf{f} \rangle$ the success pattern for *append*/3 is $\langle \mathbf{c}, \mathbf{c}, \mathbf{c} \rangle$.

So, according to the previous assumptions, for the first clause for *append*/3 we infer that the first and the second arguments are input arguments and the third argument is an output argument. For the second clause we obtain the same results. Thus, we finally infer that for *append*/3, when it is called as mentioned above, the Input/Output modes are $\langle \mathbf{In}, \mathbf{In}, \mathbf{Out} \rangle$.

4 Guard Analysis and Shift of Output Unifications

In the next step of the transformation, an analysis for each predicate in the given Prolog program is required to find any "mutually exclusive" built-in predicates appearing in the clauses for that predicate, which we call *test guards* (because they act like *guards* in KL1) or any other appearance of such *test guards* in the program and to interpret any occurrence of the *cut* operator. Since our system is a prototype implementation that is still used for experimental purposes, and to make this analysis easier, we impose some restrictions at this point on the Prolog program that is to be transformed:

- The *cut* operator is allowed in the Prolog program only if it acts like a *commit* operator and the *test guard* is an arithmetic built-in predicate (i.e. $>, <, >=, =<, =:=, = \backslash =$) or a built-in predicate used for comparison of terms (i.e. $@>, @<, @>=, @=<, ==, \backslash ==$)[7].
- Instead of the *cut* operator, an exclusive *test guard* may be used. For this reason if a built-in predicate from those previously mentioned is the first subgoal in the body of a clause, it is assumed to be a *test guard*.
- Only one *test guard* may be used at each clause and only two consecutive clauses may be "mutually exclusive".
- The variables appearing in two "mutually exlusive" *test guards* must have the same name and appearing in the same order in both *test guards*.

Example: A Prolog program that satisfies the above assumptions is

$p(X, Y, Z) : -X > Y, !, q(X, Z).$
$p(X, Y, Z) : -q(Y, Z).$
$q(X, a) : -X >= 0.$
$q(X, b) : -X < 0.$

Thus, for the transformation, when we find a *test guard* at the begining of the body of a clause followed by the *cut* operator, we replace the *cut* operator by the *commit* operator[8] and we add its "mutual exclusive" *test guard* together with the *commit* operator at the begining of the body of the following clause.

If the *test guard* we find is not followed by the *cut* operator, then we add the *commit* operator after that *test guard* in the same clause, and if the next clause in the program contains the "mutual exclusive" *test guard* of the previous one, we also add the *commit* operator after it.

The final step of the transformation is to move the output unifications in the head of each clause to the clause body. That means that, if the mode of an argument *arg* in the head of every clause for a predicate is found to be an output argument, then it will be replaced by a variable v that does not appears anywhere else in the clause, and the unification "$v = arg$" will be added to the body of the clause. If the output argument is a free variable that does not appear in the place of any input argument in the head of the clause then no change needs to be made for this argument (no output unification occurs in the head of the clause because of this argument). If the body of a clause is empty but the clause has a guard, then

[7] A *test guard* can only be one of these built-in predicates.

[8] The operator ',' before and after the *cut* symbol must also be removed.

the built-in predicate *true* has to be added as the clause body.

Example 1: For the program given in the previous example, let the modes for p be $\langle \mathbf{In}, \mathbf{In}, \mathbf{Out} \rangle$ and for q $\langle \mathbf{In}, \mathbf{Out} \rangle$. Applying the above transformations the resulting program is:

$p(X,Y,Z) : -X > Y \mid q(X,Z).$
$p(X,Y,Z) : -X =< Y \mid q(Y,Z).$
$q(X,Y) : -X >= 0 \mid Y = a.$
$q(X,Y) : -X < 0 \mid Y = b.$

Example 2: For the *append* example presented in the previous section the transformed KL1 program is :

$append([], L, K) : -K = L.$
$append([H|L1], L2, K) : -K = [H|L3], append(L1, L2, L3).$

5 Example Results

Some example results obtained by our current implementation of the system are now presented:

Example 1: Considering the following Prolog program, for generating a list of integers from B to E,

```
integers(B,E,[]):- B>E.
integers(B,E,[B|Ints]):- B=<E,B1 is B+1,
                         integers(B1,E,Ints).
```

and assuming that the calling pattern for predicate *integers*/3 is [(c,[]),(c,[]),(f,[])], we obtain the following results from our system:

```
The calling patterns and the corresponding success patterns for the predicate
integers/3 are:

Calling Pattern: [(c,[]),(c,[]),(f,[])]
Success Pattern: [(c,[]),(c,[]),(c,[])]

(c:ground term, f:free variable)
                 ------------------------
The Input/Output modes for each predicate are inferred to be:

integers(in,in,out)
                 ------------------------
The transformed program is:

integers(A,B,C):-A>B | C=[].
integers(A,B,C):-A=<B | C=[A|D],E:=A+1,integers(E,B,D).
```

Example 2: For the *quick sort* Prolog program

```
qsort([],[]).
qsort([X|L],R):-split(L,X,L1,L2),
                qsort(L1,R1),
                qsort(L2,R2),
                append(R1,[X|R2],R).

split([],_,[],[]).
split([Y|Ys],X,[Y|Ls],Bs):-Y=<X,!,
                           split(Ys,X,Ls,Bs).
split([Y|Ys],X,Ls,[Y|Bs]):-split(Ys,X,Ls,Bs).

append([],L,L).
append([X|L1],L2,[X|L3]):-append(L1,L2,L3).
```

assuming that the user has specified *qsort/2* as the only exported predicate and [(c,[]),(f,[])] as its calling pattern, the following results are obtained:

```
The calling patterns and the corresponding success patterns for the predicate
qsort/2 are:

Calling Pattern: [(c,[]),(f,[])]
Success Pattern: [(c,[]),(c,[])]

The calling patterns and the corresponding success patterns for the predicate
split/4 are:

Calling Pattern: [(c,[]),(c,[]),(f,[]),(f,[])]
Success Pattern: [(c,[]),(c,[]),(c,[]),(c,[])]

The calling patterns and the corresponding success patterns for the predicate
append/3 are:

Calling Pattern: [(c,[]),(c,[]),(f,[])]
Success Pattern: [(c,[]),(c,[]),(c,[])]

(c:ground term, f:free variable)
                   -----------------------
The Input/Output modes for each predicate are inferred to be:

qsort(in,out)
split(in,in,out,out)
append(in,in,out)
                   -----------------------
The transformed program is:

qsort([],A):-A=[].
qsort([A|B],C):-split(B,A,D,E),qsort(D,F),qsort(E,G),append(F,[A|G],C).

split([],A,B,C):-B=[],C=[].
split([A|B],C,D,E):-A=<C | D=[A|F],split(B,C,F,E).
split([A|B],C,D,E):-A>C | E=[A|F],split(B,C,D,F).

append([],A,B):-B=A.
append([A|B],C,D):-D=[A|E],append(B,C,E).
```

6 Conclusions

A system for transforming deterministic Prolog programs to KL1 is described in this paper. The results obtained from its implementation show the effectiveness of the method used, when it is applied to a certain class of Prolog Programs.

The work on mode inference described in this paper is based on the ideas presented by Debray [5], but with some important innovations:

- In the way we define *instantiation patterns* we distinguish those terms where there is a variable appearing more than once, from the rest of the terms, so that we can obtain more precise information about their dependencies after unification.
- Debray [5] uses a function called *c_closure* to indicate sharing of variables after unification. By using this function he does not take advantage of the information that we have about the occurrences of variables in the known tuple of terms. By considering the *instantiation patterns* for both terms, his treatment becomes conservative and general at this point. Instead of the function *c_closure*, we use the function *inherited_dep* which exploits any information that we have about the two tuples of terms that unify.

- With the way Debray [5] handles abstract unification, he does not encounter every possible instance that may appear during actual unification of two terms. For example, given an *ι-state* $A_0 = \{X \rightarrow \langle \mathrm{f}, \{X\}\rangle, Y \rightarrow \langle \mathrm{f}, \{Y\}\rangle\}$ and considering the unification of the tuple of terms $\bar{t_1} = \langle X, f(Y)\rangle$ with the tuple $\langle U, U\rangle$ represented in static analysis by the *ι-pattern* $\bar{I} = \langle\langle \mathrm{f}, \{2\}\rangle, \langle \mathrm{f}, \{1\}\rangle\rangle$, the result that *update_i_state* returns, according to the treatment that Debray [5] proposes, is $\{X \rightarrow \langle \mathrm{f}, \{X, Y\}\rangle, Y \rightarrow \langle \mathrm{f}, \{Y, X\}\rangle\}$ (in his treatment, he assumes that if an instantiation of a variable X is free, after *a_unify_init* (which correspons to *init_unify*, that we use), and X depends on a variable Y whose instantiation is free, then X remains a free variable). It is obvious that this result is wrong, because X in the above case is instantiated to $f(Y)$ after unification. It was shown when we described abstract unification that in the way we handle cases like this one, using the notations nv_f and d_f, we obtain the right results.

- Debray [5] is also interested in the inference of data dependencies in logic programs, and for this reason he propagates any changes in the dependency set of a variable to the dependency sets of the other variables. In this way, the inferences that we can make for the modes of a predicate are often conservative. We are actually interested in propagating any change in the instantiation of a variable to the instantiation of the variables that it depends on, so that we can infer any possible changes to their instantiation as well. If we know the variables that a variable directly depends on, then we can more precisely infer any change in its instantiation and use any information about variable groundness. By propagating the changes in the dependency set of a variable to the dependency sets of the others variables, we may lose such information and obtain conservative results. This can be noticed in the given example in **2.4.2**, where we correctly infer that during analysis in the clause for p, after the literal $r(X)$, since the instantiation of X is ground and Y is still free but depends on X, the instantiation of Y becomes ground as well. Debray [5] for the same example infers that the instantiation of Y becomes **d** "don't-know", which is sound but also conservative.

Even if the way we handle unification in "abstract domain" is more precise than other treatments previously used, we are still not able to always get precise results. This is because for the unification of two nonvariable terms we cannot precisely infer the instantiation of the variables appearing in these terms. Of course we always have sound results.

To improve the analysis we used, we can try to propagate information like those that we obtain by using the notations nv_f and d_f during abstract unification across the literals of a clause, so that we do not lose and can exploit any extra information.

The system can also check before the analysis of a clause of a predicate if its head actually unifies with the calling tuple and if it does not, ignore it and go on with the rest of the clauses. In principle, this can improve the precision of the analysis.

As other future work, the class of Prolog programs where the described transformation can be applied can be enlarged by relaxing the restrictions that we impose on the Prolog program that is to be transformed. This can be achieved by applying a more efficient guard analysis to the given Prolog program.

Of course, the system will become more powerful if it can be extended to handle non-deterministic Prolog programs, as well.

Acknowledgements

I would like to express my gratitude to Dr. Steve Gregory and Rong Yang for their help and support during the preparation and writing of this paper. Their time and contributions are deeply appreciated.

I would also like to thank Prof. D.H.D. Warren for his comments and guidance that have been very important to me.

The system described in this paper was developed as part of my M.Sc.Thesis and is partially supported by ICOT as part of the FGCS Follow-on Project.

References

[1] M. Bruynooghe,G. Janssens,A. Callebaut and B. Demoen, "Abstract Interpretation: Towards the Global Optimisation of Prolog Programs". In Proceedings of the Fourth IEEE Symposium on Logic Programming (San Francisco, September 1987). IEEE, New York, 1987

[2] M. Bruynooghe and G. Janssens, "An instance of abstract interpretation integrating type and mode inferencing". In Proceedings of the Fifth International Conference on Logic Programming (Seattle, Wash., August 1988). MIT Press, Cambridge, Mass., 1988, pp. 669-683

[3] Keith Clark and Steve Gregory, "PARLOG: Parallel Programming in Logic", ACM Transactions on Programming Languages and Systems, Vol. 8, No. 1, January 1986, pp. 1-49

[4] S.K. Debray and D.S. Warren, "Automatic Mode inference for Logic Programs", J. Logic Programming 5,3 (Sept.1988), pp. 207-229

[5] S.K. Debray, "Static Inference of Modes and Data Dependencies in Logic Programs", ACM Transactions on Programming Languages and Systems, Vol. 11, No. 3, July 1989, pp. 418-450

[6] H. Mannila and E. Ukkonen, "Flow Analysis of Prolog Programs". In Proceedings of the Fourth IEEE Symposium on Logic Programming (San Francisco, September 1987). IEEE, New York, 1987

[7] C.S. Mellish, "The Automatic Generation of Mode Declarations for Prolog Programs", DAI Research Paper 163, Dept. of Artificial Intelligence, Univ. of Edinburgh, August 1981

[8] C.S. Mellish, "Some Global Optimizations for a Prolog Compiler", J. Logic Programming 2,1 (April 1985), pp. 43-66

[9] C.S. Mellish, "Abstract interpretation of Prolog programs". In Proceedings of the Third International Logic Programming Conference (London, July 1986). LNCS 225, Springer, New York, 1986

[10] U.S. Reddy, "Transformation of Logic Programs into Functional Programs". In Proceedings of the 1984 International Symposium on Logic Programming (Atlantic City, N.J., February 1984). IEEE, New York, 1984, pp. 187-196

[11] H. Tamaki, "Stream-based Compilation of Ground I/O Prolog into Committed-choice Languages". In Proceedings of the Fourth International Logic Programming Conference, MIT Press, 1987, pp. 376-393

[12] Kazunori Ueda, "Making Exhaustive Search Programs Deterministic". In Proceedings of the Third International Logic Programming Conference, Springer-Verlag, 1986, pp. 270-282

[13] Kazunori Ueda, "Making Exhaustive Search Programs Deterministic, Part II". In Proceedings of the Fourth International Logic Programming Conference, MIT Press, 1987, pp. 356-375

[14] Kazunori Ueda, "A New Impementation Technique for Flat GHC". In Proceedings of the Seventh International Logic Programming Conference, MIT Press, 1990, pp. 3-17

[15] K. Ueda and T. Chikayama, "Design of the kernel language for the Parallel Inference Machine". Computer Journal 33, 1990, 6, pp. 494-500.

[16] D.H.D. Warren, "Implementing Prolog — compiling predicate logic programs", Res. Reps. 39 and 40, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1977

[17] Rong Yang, "CHUKL: Constraint Handling Under KL1". In Proceedings of the Workshop on Parallel Logic Programming and its Programming Environments. ICOT, Oregon, March 1994