Non-discriminating Arguments and their Uses[*]

Henning Christiansen[1] and John P. Gallagher[1][2]

[1] Computer Science, Building 43.2, P.O. Box 260, Roskilde University, DK-4000 Roskilde, Denmark
[2] IMDEA Software, Madrid
Email: {henning,jpg}@ruc.dk

Abstract. We present a technique for identifying predicate arguments that play no role in determining the control flow of a logic program with respect to goals satisfying given mode and sharing restrictions. We call such arguments *non-discriminating* arguments. We show that such arguments can be detected by an automatic analysis. Following this, we define a transformation procedure, called *discriminator slicing*, that removes the non-discriminating arguments, resulting in a program whose computation trees are isomorphic to those of the original program. Finally, we show how the results of the original program can be reconstructed from trace of the transformed program with the original arguments. Thus the overall result is a two-stage execution of a program, which can be applied usefully in several contexts; we describe a case study in optimising computations in the probabilistic logic program language PRISM, and discuss applications in tabling and partial evaluation. We also discuss briefly other possible ways of exploiting the non-discriminating arguments.

1 Introduction

The first result presented here is the identification of predicate arguments that play no role in determining the control flow of a logic program computation, with respect to initial goals satisfying given mode and sharing restrictions. We call such arguments *non-discriminating* arguments. The non-discriminating arguments can be given either manually or determined by an automatic analysis.

Following this, we define a transformation procedure, called *discriminator slicing*, that removes the non-discriminating arguments, resulting in a program whose computations are isomorphic to those of the original program. The transformation can be performed on a whole program or on individual modules, assuming that mutually recursive modules do not occur.

We present a technique for decomposing the execution of a program into two stages. The first stage executes a simplified transformed program called a *mode-sliced* program, that establishes the control flow. The second stage performs computations omitted in the first stage. We present certain practical and

[*] This work is supported by the project "Logic-statistic modelling and analysis of biological sequence data" funded by the NABIIT programme of the Danish Strategic Research Council.

conceptual benefits of this two-stage execution. Non-discriminating arguments could be used in other ways, though we focus here on transforming a program.

Removing non-discriminating arguments generates a simpler program whose control flow mirrors that of the original program. The simpler program can be executed, yielding a trace of its execution. From that trace, together with the eliminated non-discriminating arguments, the results of executing the original computation can be reconstructed by re-running the trace but including the computations for the non-discriminating arguments.

There are various uses of this two-stage process, which might at first sight appear simply to do the same work as the original computation, and even with some additional overhead. We show how the technique can lead to overall optimisation. The simpler first stage can be of benefit in tabled computations. We show such a case in the probabilistic logic program language PRISM [18].

The paper is structured as follows. In Section 2 we define the concept of a discriminating argument, along with its relation to mode and sharing abstractions. Then the slicing of a program, cutting out non-discriminating arguments, is described. In Section 3 it is shown that slicing preserves computation traces, and a two-phase execution scheme is introduced along with an illustrative example. In Section 4 a particular application is studied, namely the calculation of the most probable sequence of states in a hidden Markov model programmed in PRISM; in this application exponential speedup can be achieved, due mainly to savings in the tabled structures constructed. Sections 5 and 6 contain a discussion on the applicability of the method and related work, and Section 7 concludes.

2 Preliminaries

We follow the standard terminology and notation for logic programs [10]. For now, we consider definite logic programs that allow calls to declarative built-in predicates.

Modes. We define *mode* abstractions $\{\mathsf{v}, \mathsf{nv}\}$ having the following interpretation given by a function mode. $\mathsf{mode}(\mathsf{v})$ is the set of variables and $\mathsf{mode}(\mathsf{nv})$ is the set of non-variables. $p(m_1, \ldots, m_n)$ is a *moded atom* if p is an n-ary predicate symbol and $m_j \in \{\mathsf{v}, \mathsf{nv}\}, 1 \leq j \leq n$. A finite set of moded atoms for predicate p is called a *mode* for p. We extend mode to atoms and set of atoms; $\mathsf{mode}(p(m_1, \ldots, m_n)) = \{p(t_1, \ldots, t_n) \mid t_i \in \mathsf{mode}(m_i), 1 \leq i \leq n\}$, and $\mathsf{mode}(M) = \bigcup\{\mathsf{mode}(p(\bar{m})) \mid p(\bar{m}) \in M\}$. We say that an atom A *respects* a mode M if $A \in \mathsf{mode}(M)$.

Sharing. We adopt a variant of a standard technique [19] for representing *possible sharing* among arguments of a predicate. A pair of terms $\{t_1, t_2\}$ is said to *share* if $vars(t_1) \cap vars(t_2) \neq \emptyset$, where $vars(t)$ denotes the set of variables occurring in term t. A pair-sharing abstraction (hereafter called simply a sharing abstraction) for an n-ary predicate p is given by a subset of $\{\{i, j\} \mid i, j \in \{1, \ldots, n\}, i \neq j\}$. A sharing abstraction S for n-ary predicate p denotes a set of atoms given by $\mathsf{share}(S) = \{p(t_1, \ldots, t_n) \mid \text{if } \{t_i, t_j\} \text{ share then } \{i, j\} \in S\}$, or equivalently as $\mathsf{share}(S) = \{p(t_1, \ldots, t_n) \mid \{i, j\} \notin S$ implies $\{t_i, t_j\}$ do not share$\}$. Thus a sharing abstraction represents *possible* sharing between the given argument pairs, or equivalently definite *independence* of pairs of arguments that are absent. Namely, if $\{i, j\} \notin S$ for some sharing abstraction S, then for all atoms $p(t_1, \ldots, t_n) \in \mathsf{share}(S)$, t_i and t_j share no variable. We say that an atom A *respects* a sharing abstraction S for its predicate if $A \in \mathsf{share}(S)$.

Definition 1. *A* (argument) discrimination *for an n-ary predicate p is an atom $p(d_1, \ldots, d_n)$, where $d_i \in \{\mathsf{d}, \mathsf{nd}\}$, where* d *stands for a* discriminating *argument and* nd *stands for a* non-discriminating *argument. An argument discrimination for a program Π is a set of discriminations, one for each p/n defined in Π.*

The intention of a discrimination is to identify which arguments in a predicate have an effect on the computation flow. A discriminating argument is one which could fail to match with the corresponding argument in at least one clause head. Conversely, a non-discriminating argument is one that does not influence the success of unification of a call with any clause head. The next definition makes this precise.

Definition 2. *Given a program Π and a goal A, a* discrimination for Π is correct *for the computation of Π with A if the following condition holds. For every call $p(t_1, \ldots, t_n)$ arising in the computation, and standardised-apart clause head $p(u_1, \ldots, u_n)$,*

– *$mgu(((t_1, \ldots, t_n), (u_1, \ldots, u_n))$ succeeds iff $mgu((t_{i_1}, \ldots, t_{i_k}), (u_{i_1}, \ldots, u_{i_k}))$ succeeds, where $\{i_1, \ldots, i_k\}$ is the set of discriminating arguments of p.*

Example 1. Consider the usual append program.

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

Then `append(d,nd,nd)` is a correct discrimination for the goals `append([a],U,V)` and `append([a],[b],V)`. That is, only the first argument determines whether a call matches a clause head, even though the second argument may also be non-variable.

Since Definition 2 is given in terms of the set of all calls arising in a computation, it is necessary to find some sufficient conditions in practice, since the set of calls is infinite in general. The next definition characterises a correct discrimination with respect to a program with a set of calls denoted by a mode abstraction and a sharing abstraction.

Definition 3. *Let Π be a program, M a mode and S a sharing abstraction. Let p be a predicate and $p(d_1, \ldots, d_n)$ a discrimination for p. Then the* discrimination is correct *with respect to M and S if for every standardised-apart clause head $p(u_1, \ldots, u_n)$ and $p(m_1, \ldots, m_n) \in M$, the set of pairs $\{\langle u_i, m_i \rangle \mid 1 \leq i \leq n\}$ satisfies the following condition;*

– *for all $1 \leq i \leq n$, if $d_i = \mathsf{nd}$ then u_i is a variable or $m_i = \mathsf{v}$; and*
– *there do not exist $\langle u_i, m_i \rangle$ and $\langle u_j, m_j \rangle$, $i \neq j$, such that $d_i = \mathsf{nd}$, $m_i = \mathsf{v}$ and $\{i, j\} \in S$, and u_j is non-variable; and*
– *there do not exist $\langle u_i, m_i \rangle$ and $\langle u_j, m_j \rangle$, $i \neq j$, such that $d_i = \mathsf{nd}$, u_i is a variable, $m_j = \mathsf{nv}$ and $\mathsf{share}(u_i, u_j)$.*

Informally, consider a call $p(t_1, \ldots, t_n)$ that satisfies the mode and sharing declarations for p, and a (standardised apart) clause head $p(u_1, \ldots, u_n)$. Then for each non-disriminating argument position i, according to Definition 3, at least one of t_i and u_i is a variable. Furthermore, if t_i is a variable and shares with some other argument t_j then u_j is a variable; and if u_i shares with some other argument u_j then t_j is a variable.

Example 2. Consider again the append program, the mode $\{$`append(nv,nv,v)`$\}$ and sharing abstraction $\emptyset$ for `append`. Then `append(d,nd,nd)` is a correct discrimination. Note that argument 2 is non-discriminating despite the fact that it is non-variable. Although in clause head `append([],Ys,Ys)` arguments 2 and 3 share, they are not both matched to non-variables.

The following lemma states that a correct discrimination with respect to a mode and sharing abstraction (Definition 3) safely approximates the condition of Definition 2.

Lemma 1. *Let Π be a program, M a mode and S a sharing abstraction. Assume that for all A respecting M and S, every call in the computation of Π with A respects M and S. Let Δ be a discrimination for Π. Then if Δ is correct with respect to M and S for each predicate in Π then Δ is correct with respect to Π and A, for all A respecting M and S.*

Proof (Sketch). It can be verified that Definition 3 ensures that any call respecting the given mode and sharing cannot fail due to unifying the non-discriminating arguments. When unifying on a non-discriminating argument at least one of the two terms is variable. Thus the only possible cause of failure (since the two terms are standardised apart) is another occurrence of the variable that is matched to a different term. This cause is excluded by the conditions of Definition 3. It follows that every call unifies with each clause head if and only if the discriminating arguments unify, as required by Definition 2.

Constructing a discrimination from mode and sharing abstractions. Given a mode and sharing abstraction, we can construct a discrimination that satisfies the conditions of Definition 3.

Let Π be a program, M a mode and S a sharing abstraction; construct a discrimination for each predicate p as follows. The i^{th} argument of p is nd if and only if either (a) the i^{th} argument of each clause head for p is a variable, and if the i^{th} argument shares with another head argument then that argument is also a variable, or (b) the mode of the i^{th} argument is v in all mode atoms and if $\{i, j\} \in S$ then $m_j = \mathsf{v}$ also. Note also that a discrimination can be relaxed by replacing any nd by d, preserving correctness.

Example 3. Consider the usual append program, the mode {`append(nv,nv,v)`} and sharing abstraction $\emptyset$ for `append`. Then we derive `append(d,nd,nd)` as the discrimination. As there is no sharing in the calls, the 2nd argument is non-discriminating even though it is non-variable. Given the mode {`append(nv,v,v)`} and the same sharing abstraction we obtain `append(d,nd,nd)` once again. Given the mode {`append(v,v,nv)`} and the same sharing abstraction, we obtain the discrimination `append(nd,nd,d)`.

Analysis for Discrimination. Static analyses for freeness and sharing are well established, starting with [19, 4] and we can apply them for automatically constructing correct discriminations. Given an initial moded call and a sharing abstraction on the call, such an analysis returns, for each predicate, a safe mode and sharing abstraction; that is, one respected by every call. An analysis is performed for a given computation rule; in this paper we assume the standard left-to-right, depth-first strategy. As discussed later, more precise analyses could be employed to derive more accurate discriminations than the one described here, which is based on very simple modes and no information on term structure.

2.1 Discriminator slicing

Let Π be a program and Δ a discrimination for Π which is correct with respect to a mode and sharing abstraction for Π.

An argument position p_i where p is an n-ary predicate and $1 \leq i \leq n$ is *deletable* if

- that argument position is `nd` in Δ, and
- no occurrence of that argument position in the program contains a term that shares with an argument that is marked `d`.

A *slice* with respect to Δ is obtained by replacing each clause $A_0 \leftarrow A_1, \ldots, A_n$ in Π by $A'_0 \leftarrow A'_1, \ldots, A'_n$ where A'_i is obtained by deleting from A_i all deletable arguments. We call the resulting program Π^{Δ}.

Discriminator slicing with respect to a predicate. Slicing with respect to a predicate p allows the removal of body atoms in the clauses for p. A body atom can be removed if it cannot influence the choice of clause for p, or some predicate mutually recursive with p, in a computation,

Let Π be a program and Δ a discrimination for Π which is correct with respect to a mode and sharing abstraction for Π. Let G_Π be the predicate dependency graph of Π and $c_0, c_1, \ldots, c_l$ be the sequence of strongly connected components [21] of G_Π in some topologically sorted order (where c_l is the "top" component). Let $A_0 \leftarrow A_1, \ldots, A_n$ in Π be a clause whose head has predicate p and let c_k be the component containing p. Then A_i $(i \leq i \leq n)$ is deletable if

- A_i's component is c_j where $j < k$, and
- no argument of A_i shares with any `d` argument of an atom A_m, $m \neq i$, where A_m's predicate is in c_k.

Let Π_p^Δ be the program obtained by first constructing Π^Δ and then removing any deletable atoms (with respect to predicate p).

Discriminator slicing with respect to built-ins or imported predicates. built-in predicates are considered to be at the bottom of the predicate dependency graph. Thus slicing with respect to a predicate p allows removal of calls to built-ins from p's clauses that cannot affect p's control flow. However the same principle allows imported predicates can be handled in a similar way, assuming that mutually recursive predicates are not in separate modules. Thus predicate based slicing can be used to remove calls to imported predicates from a module if they cannot affect the control flow of the predicates of the module.

3 Discriminator Slicing and the Preservation of Traces

We now deal with the question of what properties of a program are preserved by discriminator slicing. The overall answer is that the control flow is preserved. To make this precise we introduce derivation trees and trace terms [3].

Derivations. A single moded atom is assumed for the top predicate, and a query to a given program consists, for simplicity, of a single call to the top predicate respecting its mode. The following characterisation of derivations and trace trees is adapted from [3].

Definition 4. *An* AND-tree *(for program Π) is a tree each of whose nodes is labelled by an atom and a clause, such that*

1. *each non-leaf node is labelled by a clause $A \leftarrow A_1, \ldots, A_k$ and an atom $A\theta$ (for some substitution θ), and has children $A_1\theta, \ldots, A_k\theta$,*
2. *each leaf node is labelled by a clause $A \leftarrow true$ and an atom $A\theta$ (for some θ).*

It was shown by Stärk [20] that A has answer θ in program Π if and only if there is an AND-tree (for Π) with root node labelled by $A\theta$.

Furthermore, a successful derivation with left-right depth-first computation rule (or any other computation rule) can be transformed into an AND-tree. Each AND-tree can be associated with a trace term.

Definition 5. *Let T be an AND-tree; define $\alpha(T)$ to be either*

1. *f_j, if T is a single leaf node labelled by the unit clause identified by f_j; or*
2. *$f_i(\alpha(T_1), \ldots, \alpha(T_n))$, if T is labelled by the clause identified by f_i/n, and has immediate subtrees $T_1, \ldots, T_n$.*

Adding trace-terms to programs. Trace-terms can easily be added to logic programs, so that the computation returns a trace term as well as its normal result. Let Π be a program and let the i^{th} clause be $p(\bar{t}) \leftarrow q_1(\bar{t}_1), \ldots, q_{a_i}(\bar{t}_{a_i})$. Let f_i/a_i be the function symbol associated with the i^{th} clause. Transform each such clause to $p(\bar{t}, f_i(Y_1, \ldots, Y_{a_i})) \leftarrow q_1(\bar{t}_1, Y_1), \ldots, q_{a_i}(\bar{t}_{a_i}, Y_{a_i})$, where $Y_1, \ldots, Y_{a_i}$ are distinct variables not occurring elsewhere in the clause.

Finally, transform each atomic goal $\leftarrow q(\bar{s})$ to $\leftarrow q(\bar{s}, W)$, where W is a variable not occurring elsewhere in the goal. Given a program Π we denote the program obtained by adding trace terms by Π^+.

We can modify the trace term to incorporate built-ins and imported predicates for which the clauses are not available. We simply define a unique constant for each such call and add it to the trace term. E.g. if $q_j(\bar{t}_j)$ in the clause $p(\bar{t}) \leftarrow q_1(\bar{t}_1), \ldots, q_j(\bar{t}_j), \ldots, q_{a_i}(\bar{t}_{a_i})$ is a call to a built-in, we transform the clause to $p(\bar{t}, f_i(Y_1, \ldots, b, \ldots, Y_{a_i})) \leftarrow q_1(\bar{t}_1, Y_1), \ldots, q_j(\bar{t}_j), \ldots, q_{a_i}(\bar{t}_{a_i}, Y_{a_i})$, where b is the unique identifier for that built-in.

Preservation of traces by discriminator slicing. Consider the result of discriminator slicing; some arguments are removed from the program but otherwise the structure of clauses is intact. Assume that the same clause identifiers are retained in the sliced program. Then the AND-trees of the sliced program are in one-to-one correspondence with those of the original program. This implies that, if we add trace-terms to both programs as in Definition 5, then the two programs generate exactly the same trace terms.

Proposition 1. *Let Π be a program and A a goal. Let Π^+, A^+ be the result of adding trace terms to Π and A. Let Δ be a correct discrimination for Π; Π^Δ, A^Δ the discriminator slice, and $\Pi^{\Delta^+}, A^{\Delta^+}$ the result of adding trace terms. Then there is an execution of A^+ in Π^+, yielding trace term t if and only if there is an execution of A^{Δ^+} in Π^{Δ^+} yielding t.*

Proof (Sketch). The sliced program Π^{Δ^+} uses only the discriminating arguments but a call unifies with a clause head in Π^{Δ^+} iff the corresponding call with all arguments unifies with the corresponding clause head in Π^+. Clearly the trace terms do not influence the control flow. Hence the computation follows exactly the same course and so the same trace terms are generated.

In the case of a slice with respect to a predicate, the trace term is preserved apart from the subterms corresponding to the deleted atoms. We state the corresponding correctness result.

Proposition 2. *Let Π be a program and A a goal. Let Π^+, A^+ be the result of adding trace terms to Π and A. Let Δ be a correct discrimination for Π and p a predicate; Π_p^Δ, A^Δ the discriminator slice wrt p, and $\Pi_p^{\Delta^+}, A^{\Delta^+}$ the result of adding trace terms. Then there is an execution of A^+ in Π^+, yielding trace term t if and only if there is an execution of A^{Δ^+} in $\Pi_p^{\Delta^+}$ yielding t', where t' is obtained from t by deleting the subterms corresponding to the deleted atoms.*

Re-running a trace. Having generated a trace term t (using A^{Δ^+} in $\Pi_p^{\Delta^+}$), we can then insert t into the trace term argument of the goal for the original goal A^+ in Π^+. The trace term will force the computation to re-run exactly the same path as given in the trace term. While there may have been backtracking while generating the trace term, the re-run using a ground trace term is completely deterministic.

Furthermore, some computation on failed branches is possibly avoided in the sliced program due to the deletion of atoms that do not affect the control flow. When re-running the program there is no backtracking. Thus it is possible that there is an overall saving in executing a program in two stages; the first phase to generate a successful trace using the sliced program, but doing less work in the unsuccessful branches than would have occurred in the original program. The second phase generates the full answer using the trace, with no redundant computation.

Example 4. We consider an implementation of a finite state machine which moves from states to states and consumes/emits a letter in each transition; for simplicity, we let the state machine treat all different letters alike. The top predicate `m(`*history*`,` *letter-sequence*`)` where the *history* records, for each step consecutively numbered, the states passed and letters seen. For example, the letter sequence `[a, b]` may give rise to the history `[s(0,q1), e(0,a), s(1,q2), e(1,b), s(2,end)]`. We assume the following mode pattern indicating that the program is used for mapping sequences of letters into histories, `m(v, nv)`; the program Π is as follows.

```
m(H,Ls):- m(q0,0,H,Ls).
m(end,N,[s(N,end)],[]).
m(Q,N,[s(N,Q),e(N,L)|H],[L|Ls]):-
  suc(Q,Q1), N1 is N+1, m(Q1,N1,H,Ls).
suc(q0,q1). suc(q0,q3).
...
```

A discrimination for this program is given as follows; it can be derived automatically from a sharing and freeness analysis with respect to the goal `m(v,nv)` with no sharing between the arguments.

`m(nd,nd)`, `m(d,nd,nd,d)`, `suc(d,nd)`,

We make now a two-stage transformation of this program, producing a sliced version Π_m^{Δ} with respect to the predicate `m/4`. Notice the call to built-in "`is`" is removed by this transformation, since `N1 is N+1` processes only variables that are non-discriminating in `m/4`. We then extend this with trace terms, corresponding to the production of $\Pi_m^{\Delta^+}$.

```
mDeltaT(Ls,f1(T)):- mDeltaT(q0,Ls,T).
mDeltaT(end,[],f2).
mDeltaT(Q,[L|Ls],f3(T1,T2):-
  sucDeltaT(Q,Q1,T1), mDeltaT(Q1,Ls,T2).
sucDeltaT(q0,q1,f4).  sucDeltaT(q0,q3,f5).
...
```

Finally, we generate the trace term version of the original program, corresponding to Π^+.

```
mPlus(H,Ls,f1(T)):- mPlus(q0,0,H,Ls,T).
mPlus(end,N,[s(N,end)],[],f2).
mPlus(Q,N,[s(N,Q),e(N,L)|H],[L|Ls],f3(T1,is,T2)):-
   sucPlus(Q,Q1,T1),
   N1 is N+1, mPlus(Q1,N1,H,Ls,T2).
sucPlus(q0,q1,f4).  sucPlus(q0,q3,f5).
...
```

Instead of querying the original program by *(i)* `m(H,`*a-list-of-letters*`)`, we can pose the query equivalently as follows,

(ii) `?- mDeltaT(`*a-list-of-letters*`,T), mPlus(H,`*the-same-list-of-letters*`,T).`

For measuring the difference in runtime, we detailed a state machine that needs to explore combinatorially many failing branches before escaping through an end state. While *(ii)* may look more complicated, it runs about 20% faster than *(i)*.[3] To explain this difference, notice firstly that the call to `mPlus` is negligible wrt. runtime as it executes deterministically for a correct trace. The unifications in `mDeltaT` all involve patterns mentioned in the head of clauses, so that the Prolog compiler can reduce them to very little work at runtime. Finally, the queries to `m` and `mPlusT` involve exactly the same number of failing and successful branches, so the speedup reflects the difference in efficiency of the single clauses.

It is clear that an arbitrarily large speedup can be demonstrated by applying this technique to suitably constructed programs with heavy use of built-ins and backtracking. In fact, as noted, the technique can be generalized to remove also calls to program defined predicates or imported predicates.

4 Discriminator slicing in tabling systems, including PRISM

We now discuss a case study in which drastic speedup is achieved using discriminator slicing in relation to tabled logic programming systems.

Very briefly, we can explain tabling [16] as a mechanism utilized in the execution of Prolog programs that maintains a table of successful calls and their answers, and whenever a call is encountered, it is checked if it (or perhaps a more general call) is known in the table already; if so, there is no need to execute it once again as the answers are ready in the table.

[3] This test was made using the optimizing compiler of SICStus Prolog 4.0.4 under Mac OS X 10.5.6, 2.4 GHz Intel Core 2 Duo with 4GB RAM.

Comparing the use of tabling for a program Π and its reduced version Π^{Δ}, we notice that different calls of Π differing only in the non-discriminating arguments, will merge into a single call in Π^{Δ}. Thus the table can be much smaller, and there is a larger chance that the current call has a match in the table.

We have applied this principle in a preprocessor for the probabilistic-logic PRISM [18] system, where in some cases, it reduces time complexity from exponential or worse to linear for applications of the system's generic Viterbi algorithm. For a detailed explanation of PRISM's facilities we refer to its manual [17]; PRISM is based on BProlog [25] from which it inherits a tabling mechanism.

Programs in PRISM are basically Prolog programs extended with random variables, called multi-valued switches. With given probabilities for the switches, a probabilistic semantics is induced that associates a probability to each true atom in the program so that a program becomes a probabilistic model. One possible application of a PRISM program is to find the set of outcomes of switches that provides the maximum probability, called the Viterbi probability, for a given observation represented as a top-level goal g; this can be done by a call `viterbif(`*g*`)`. Another useful option is `viterbig(`*g*`)`, which instantiates the variables in g to terms that provide the highest Viterbi probability. These predicates are given in alternative versions that also provide the Viterbi probability as well as an "explanation" which basically is a representation of the proof tree, including outcomes for the switches, that gives rise to the Viterbi probability.

The `viterbig` facility is interesting, among others applications, for prediction of structures in genomic sequence data. Many different models can be used and PRISM appears as a very flexible tool for developing such models. Here we will illustrate a Hidden Markov Model [14] (HMM) which can be represented as a predicate `hmm(`*annotation*`,`*sequence*`)` where *sequence* is a sequence of the letters a, c, g, t, in length between hundreds and in principle up to billions, and *annotation* is a description of those structures that the biologists find interesting (e.g., proposed positions of genes or detailed intron-exon structures).

A call such as `viterbig(hmm(A,`*sequence*`))` typically leads to a combinatorial explosion, but with our program slicing method we can achieve linear complexity which is the best possible. We will explain the details for the following PRISM program, Π_{HMM}; it defines a general HMM which records the sequence of states during a run. The most probable sequence is the so-called Viterbi path.[4]

```
values(letter(_state), [a,c,g,t]).
values(next_state(_state), [q1,q2,end]).
hmm(A,S):- hmm(q1,A,S).
hmm(end,[end],[]).
hmm(Q,[Q|Qs],[L|S]):-
   Q \= end,
   msw(letter(Q),L), msw(next_state(Q), Q1),
   hmm(Q1,Qs,S).
```

[4] This code is inspired by similar examples in the PRISM manual [17].

We have left out additional program lines that set the probabilities for the switches defined by the values declarations; notice that a parameterized pattern such as `letter(_state)` indicates a family of random variables. The `msw` predicate is a reference to a switch and may instantiate its second argument to any outcome of the switch.

The strategy applied for Viterbi computations in PRISM is to explore all possible proof trees being stored as a so-called explanation graph, but using sharing of subtrees whenever their top nodes are identical; this sharing is implemented through clever use of the underlying tabling mechanism in a way that we shall not describe here. When the program above is used for finding the best path for a given sequence, calls of the form

$$\texttt{hmm(}s_k\texttt{, Qs,[}\ell_k\texttt{,}\ldots\texttt{,}\ell_n\texttt{])}$$

are made to the recursive predicate for all k and possible value of s_k. PRISM considers all possible answers for it, and they are entered in the underlying table; these answers amount to all possible instances given by substitutions of the form

$$\texttt{Qs} \rightarrow \texttt{[}s_k\texttt{,}\ldots\texttt{,}s_n\texttt{]}.$$

The explanation graph needs to include all correspondingly instantiated nodes, of which there are clearly exponentially many.

On the other hand, the annotation arguments are non-discriminating and can be removed while still retaining the same set of proof trees (modulo mappings to adds/remove the annotation arguments). However, without the annotations far more nodes can share; more precisely the exponential amount of different nodes for each k and s_k reduces to a single node in the graph. To see this, consider reduced program $\Pi^{\Delta}_{\mathsf{HMM}}$ which is as follows.

```
hmm(S):- hmm(q1,S).
hmm(end,[]).
hmm(Q,[L|S]):-
   Q \= end,
   msw(letter(Q),L), msw(next_state(Q), Q1),
   hmm(Q1,S).
```

The recursive calls are now of the form $\texttt{hmm(}s_k\ \texttt{,[}\ell_k\texttt{,}\ldots\texttt{,}\ell_n\texttt{])}$ where all arguments are grounded, thus only one possible answer, namely the empty substitution corresponding. This means that the explanation graph can be viewed as a structure a width equal to the number of different states and a length equal to the sequence length. The construction of this graph can be done in time linear in the size sequence length.

It is now so fortunate that `viterbif` can return a representation of the best proof tree extracted as a subgraph of the explanation graph. This tree can then be mapped into a desired annotation in one efficient run of a program such as our Π^{+}_{HMM} adapted for PRISM's particular proof tree format.

We have developed a little preprocessor, called `autoAnnotations` [1], which given a program Π_{HMM} as above, produces automatically $\Pi^{\Delta}_{\mathsf{HMM}}$ as well Π^{+}_{HMM}.

The current version of this system requires the user to indicate the arguments and body calls to be removed. With the analysis methods described here, this can be done fully automatically from a single mode declaration for the top predicate.

We tested runtime for Viterbi computations with the reduced and non-reduced version. While the non-reduced version did not return an answer for sequences of length 20 within several hours, the reduced one $\Pi^{\Delta}_{\mathsf{HMM}}$ followed by our postprocessor Π^{+}_{HMM} could produce Viterbi paths for lists of lengths up to 20,000 in few minutes on a machine with sufficient amount of RAM.

5 Discussion

It is clear that the slicing technique and two-phase execution can slow a program down. In the worst case no arguments or atoms are deletable so the program will be executed twice in its entirety. On the other hand there are examples such the one described above where spectacular speedup is achieved. Thus the technique needs to be used with care and targeted to appropriate examples. It seems hard to characterise precisely the class of programs that could benefit from its application. Applications that require explicit manipulation of computation trees or traces could be candidates; these might include online partial evaluation, where an explicit representation of computation trees are constructed and some notion of tabling is used to handle infinite branches of the tree [12, 7]. In the case of the Viterbi calculation in Section 4, a key point is that one computation path (the most probable) is returned but the whole tree must be constructed first. Thus savings while constructing the tree are worthwhile. Search problems in which some structure is computed while searching for a solution are also liable to optimisation, as discussed in Example 4. The transformation can eliminate cases where partial solutions are constructed on failing branches of the search and then thrown away. Other examples in this class are non-deterministic parsers constructing a syntax tree. It is likely that the arguments of the parser constructing the syntax tree are non-discriminating; thus it could be a substantial optimisation for highly non-deterministic grammars to generate a trace of a successful parse and then deterministically construct the syntax tree afterwards.

Apart from optimisation, there could be applications of discriminator slicing for refactoring of logic programs [23]. The sliced program expresses the control part of the program. Thus two predicates p_1, p_2 that have isomorphic discriminator slices with respect to p_1, p_2 respectively could be said to share the same control. Such information could be useful for understanding, documenting or comparing programs. Furthermore the idea of recording and replaying program traces has applications in several other fields, especially debugging and understanding of concurrent programs [15].

6 Related Work

Program slicing [24, 22] is a collection of techniques for removing parts of the code of a program that are irrelevant with respect to some chosen part of the program's semantics (the *slicing criterion*). Most slicing criteria concern the values of some selected variables; by contrast our slicing technique preserves control flow. However *path slicing* [5] is more similar; it concerns removing code that does not affect a given computation path; we differ in that our slicing criterion is the set of all computation paths rather than a specific one, and in that the underlying analyses are different from those used in an imperative language. We have done some initial investigation on the formulation of our approach as a value-based slicing technique using trace terms. The approach is to construct a slice of the program augmented with trace terms (Section 3) with respect to the trace term argument; the slice in principle contains only the operations needed to preserve the trace terms, that is, the control flow. Using a logic program slicing approach such as Leuschel and Vidal's [9] incorporating partial evaluation and the redundant argument filtering transformation [8] it is possible to handle simple examples but it is unclear at present whether the analyses incorporated in these tools are powerful enough.

Applications of mode and sharing analysis are many, including automatic parallelisation [13], occur-check elimination [19, 4], non-failure analysis [2] and determinacy analysis [11]. The latter two applications resemble discriminator analysis in some respects and give pointers to obtaining more precise discriminations. The conditions for detecting non-failure, for example, are that each call unifies with at least one clause head, while the conditions for detecting determinacy are that each call unifies with exactly one clause head. Adapting these conditions and applying them argumentwise should lead to conditions expressing, for each argument or group of arguments, whether they unify with each clause head. If these conditions are true for some argument and every clause head, then that argument is non-discriminating. Further study is required.

We chose to apply discriminations to obtain a sliced program, but the same information could be applied dynamically during execution. In this respect there is a relation to control flow generation. Automatic generation of delay mechanisms and reordering of subgoals in clause bodies for improving efficiency and termination properties of logic programs have been considered by [6]. These techniques are somewhat orthogonal to ours, but we notice that our method may be adapted to generate delay declarations for calls which, in the construction of our reduced programs, are subject to deletion. Consider the finite state machine program of Example 4 in which the call `N1 is N+1` can be removed in the reduced program. Instead of deleting this call, we may delay it by an inline application of `freeze` or by replacing it with a call `addOne(N1,N)` where the new predicate is defined as follows.

```
:- block addOne(?,-).
addOne(N1,N):- N1 is N+1.
```

We can obtain an improved efficiency by only executing calls to `addOne` that occur in a successful execution of the program, by changing the top clause of the program into the following.

```
m(H,Ls):- m(q0,Zero,H,Ls),Zero=0.
```

(This step employs the data flow analysis of the program). An optimizing compiler and a runtime system with low overhead for delays may produce a program that runs almost as efficiently as the reduced version. It is not clear at present whether the approach of [6] will detect this opportunity for optimisation; the reordering of calls in the body for an optimised execution in their approach may possibly be utilized in order to classify more arguments as non-discriminating.

7 Conclusion

We have presented the concept of discriminating arguments and shown that they may be detected automatically given a moded goal. A slicing transformation was defined in which the slice preserves the computation tree structure. Using trace terms we then defined a two-phase execution in which the control flow is first established and then the full results are generated from the trace. Applications where this approach could be beneficial were presented and discussed.

Acknowledgements. We thank Yoshitaka Kamaya, Ole Torp Lassen, Pedro López-García and Manuel V. Hermenegildo for useful discussions.

References

1. H. Christiansen. Efficient Viterbi for PRISM models with annotations, 2009. Website with source code; http://www.ruc.dk/∼henning/PRISMannotations.
2. S. K. Debray, P. López-García, and M. V. Hermenegildo. Non-failure analysis for logic programs. In L. Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming, ICLP'97*, pages 48–62. MIT Press, 1997.
3. J. P. Gallagher and L. Lafave. Regular approximation of computation paths in logic and functional languages. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, volume 1110 of *Springer-Verlag Lecture Notes in Computer Science*, pages 115–136, 1996.
4. D. Jacobs and A. Langen. Accurate and efficient approximation of variable aliasing in logic programs. In E. L. Lusk and R. A. Overbeek, editors, *Logic Programming, Proceedings of the North American Conference 1989, NACLP'89*, pages 154–165. MIT Press, 1989.
5. R. Jhala and R. Majumdar. Path slicing. In V. Sarkar and M. W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005 (PLDI'05)*, pages 38–47, 2005.
6. A. King and J. C. Martin. Control generation by program transformation. *Fundam. Inform.*, 69(1-2):179–218, 2006.
7. M. Leuschel. Advanced logic program specialisation. In J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, editors, *Partial Evaluation - Practice and Theory*, volume 1706 of *Lecture Notes in Computer Science*, pages 271–292. Springer, 1999.
8. M. Leuschel and M. H. Sørensen. Redundant argument filtering of logic programs. In J. P. Gallagher, editor, *Logic Programming Synthesis and Transformation, 6th International Workshop, LOPSTR'96, Stockholm, Sweden, August 28-30, 1996, Proceedings*, pages 83–103, 1996.

9. M. Leuschel and G. Vidal. Forward slicing by conjunctive partial deduction and argument filtering. In S. Sagiv, editor, *Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, pages 61–76, 2005.
10. J. Lloyd. *Foundations of Logic Programming: 2nd Edition*. Springer-Verlag, 1987.
11. P. López-García, F. Bueno, and M. V. Hermenegildo. Determinacy analysis for logic programs using mode and type information. In S. Etalle, editor, *Logic Based Program Synthesis and Transformation, 14th International Symposium, LOPSTR 2004, Verona, Italy, August 26-28, 2004, Revised Selected Papers*, volume 3573 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2004.
12. B. Martens and J. P. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proc. International Conference on Logic Progrmaming, (ICLP'95), Tokyo*. MIT Press, 1995.
13. K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In E. L. Lusk and R. A. Overbeek, editors, *Logic Programming, Proceedings of the North American Conference 1989, NACLP'89*, pages 166–189. MIT Press, October 1989.
14. L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.
15. M. Ronsse, K. D. Bosschere, M. Christiaens, J. C. de Kergommeaux, and D. Kranzlmüller. Record/replay for nondeterministic program executions. *Commun. ACM*, 46(9):62–67, 2003.
16. K. F. Sagonas and D. S. Warren. A portable compiler for integrating HiLog into Prolog systems. In M. Bruynooghe, editor, *Logic Programming, Proceedings of the 1994 International Symposium*, page 682. MIT Press, 1994.
17. T. Sato. PRISM, Programming in Statistical Modeling (PRISM web site), checked 2009. http://mi.cs.titech.ac.jp/prism/.
18. T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research (JAIR)*, 15:391–454, 2001.
19. H. Søndergaard. An application of abstract interpretation of logic programs: Occur check reduction. In B. Robinet and R. Wilhelm, editors, *ESOP 86, European Symposium on Programming, Saarbrücken, Federal Republic of Germany, March 17-19, 1986, Proceedings*, volume 213 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 1986.
20. R. F. Stärk. A direct proof for the completeness of SLD-resolution. In E. Börger, H. K. Büning, and M. M. Richter, editors, *CSL '89, 3rd Workshop on Computer Science Logic, Kaiserslautern, Germany, October 2-6, 1989, Proceedings*, volume 440 of *Lecture Notes in Computer Science*, pages 382–383. Springer, 1989.
21. R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.
22. F. Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.
23. W. Vanhoof. Searching semantically equivalent code fragments in logic programs. In S. Etalle, editor, *Logic Based Program Synthesis and Transformation, 14th International Symposium, LOPSTR 2004, Verona, Italy, August 26-28, 2004, Revised Selected Papers*, volume 3573 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2004.
24. M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.
25. N.-F. Zhou. B-Prolog web site, 1994–2009. http://www.probp.com/.