Localizing and explaining reasons for non-terminating logic programs with failure-slices

Ulrich Neumerkel[*] and Fred Mesnard

Iremia, Université de la Réunion,
15, avenue René Cassin - BP 7151 -
97 715 Saint Denis Messag. Cedex 9 France
`ulrich@complang.tuwien.ac.at fred@univ-reunion.fr`

Abstract. We present a slicing approach for analyzing logic programs with respect to non-termination. The notion of a failure-slice is presented which is an executable reduced fragment of the program. Each failure-slice represents a necessary termination condition for the program. If a failure-slice does not terminate it can be used as an explanation for the non-termination of the whole program. To effectively determine useful failure-slices we combine a constraint based static analysis with the dynamic execution of actual slices. The current approach has been integrated into a programming environment for beginners. Further, we show how our approach can be combined with traditional techniques of termination analysis.

1 Introduction

Understanding the termination behavior of logic programs is rather difficult due to their complex execution mechanism. Two different intertwined control flows (AND and OR) cause a complex execution trace that cannot be followed easily in order to understand the actual reason for termination or non-termination. The commonly used *procedure box model* introduced by [2] for debugging, produces a huge amount of detailed traces with no relevance to the actual termination behavior. Similarly, the notion of proof trees is not able to explain non-termination succinctly.

Current research in termination analysis of logic programs focuses on the construction of termination proofs. Either a class of given queries is verified to guarantee termination, or —more generally— this class is inferred [9]. In both cases that class of queries is a sufficient termination condition and often smaller than the class of actually terminating queries. Further this class is described in a separate formalism different from logic programs. Explanations why a particular query does not terminate are not directly evident.

[*] On leave of: Technische Universität Wien, Institut für Computersprachen

G. Nadatur (Ed.): PPDP'99, LNCS 1702, pp. 328-341, 1999.

We present a complementary approach, that is able to localize and explain reasons for non-termination using a newly developed slicing technique based on the notion of failure-slices. Failure-slices expose those parts of the program that may cause non-termination; under certain conditions, non-termination can be proved.

Slicing [15] is an analysis technique to extract parts of a program related to or responsible for a particular phenomenon (e.g. a wrong value of a variable). Originally, slicing was developed for imperative languages by Weiser [15, 16] who observed that programmers start debugging a program by localizing the area where the error has to be. Using program analysis techniques, this process can be partially automated, simplifying the comprehension of the program. Only recently, slicing has been adopted to logic programming languages by Zhao [17], Gyimóthy [5], and Ducassé [14]. While these approaches focus on explaining (possibly erroneous) solutions of a query, we will present a slicing technique for explaining non-termination properties. It is an implementation of a previously developed informal reading technique used in Prolog-courses [11, 12] which is used within a programming environment for beginners [13].

In contrast to most other programming paradigms, there are two different notions of termination of logic programs - existential [8] and universal termination. A query terminates existentially, if one (or no) solution can be found. Universal termination requires the complete SLD-tree being finite [4]. While existential termination is easy to observe, it turned out to be rather difficult to reason about. On the other hand, universal termination, while difficult to observe, is much easier to treat formally. Further, universal termination is more robust to typical program changes that happen during program development. Universal termination is sensitive only to the computation rule but insensitive to clause selection. As has been pointed out by Plümer [7] the conjunction of two universally terminating goals always terminates. Further, reordering and duplicating clauses has no influence. For this reasons, most research on termination focused on universal termination. We will consider universal termination with the leftmost computation rule, as used for Prolog programs.

Example. The following example contains an erroneous data base causing universal non-termination of the given query. Its non-termination cannot be easily observed by inspecting the sequence of produced solutions. Glancing over the first solutions suggests a correct implementation. But in fact, an infinite sequence of redundant solutions is produced. The failure-slice on the right, generated automatically by the presented method, locates the reason for non-termination by hiding all irrelevant parts. The remaining slice has to be changed in order to make the program terminating.

The failure-slice helps significantly in understanding the program's termination property. It shows for example that clause reordering in ancestor_of/2 does not help here since this would lead to the same slice. Further it becomes evident, that the first rule in ancestor_of/2 is not responsible for termination. Often beginners have this incorrect belief confusing universal and existential termination.

% original program
← ancestor_of(Anc, leopold_I).
child_of(karl_VI, leopold_I).
child_of(maria_theresia, karl_VI).
child_of(joseph_II, maria_theresia).
child_of(leopold_II, maria_theresia).
child_of(leopold_II, franz_I).
child_of(marie_a, maria_theresia).
child_of(franz_I, leopold_II).

ancestor_of(Anc,Desc) ←
  child_of(Desc,Anc).
ancestor_of(Anc,Desc) ←
  child_of(Child, Anc),
  ancestor_of(Child, Desc).

% failure-slice
← ancestor_of(Anc, leopold_I).
~~child_of(karl_VI, leopold_I) ← **false.**~~
~~child_of(maria_theresia, karl_VI) ← **false.**~~
~~child_of(joseph_II, maria_theresia) ← **false.**~~
~~child_of(leopold_II, maria_theresia) ← **false.**~~
child_of(leopold_II, franz_I).
~~child_of(marie_a, maria_theresia) ← **false.**~~
child_of(franz_I, leopold_II).

~~ancestor_of(Anc,Desc) ← **false,**~~
  ~~child_of(Desc,Anc).~~
ancestor_of(Anc,Desc) ←
  child_of(Child, Anc),
  ancestor_of(Child, Desc)**, false.**

This example shows also some requirements for effectively producing failure-slices. On the one hand we need an analysis to identify the parts of a program responsible for non-termination. On the other hand, since such an analysis can only approximate the minimal slices, we need an efficient way to generate all slices which then are tested for termination by mere execution for a certain time. With the help of this combination of analysis and execution we often obtain explanations also when classical termination analysis cannot produce satisfying results.

Contents. The central notions failure-slice and minimal explanation are presented in Section 2. Some rules are given in Section 3 that must hold for minimal explanations. Section 4 presents our implementation. Finally we discuss how our approach is adapted to handle some aspects of full Prolog. A complete example is found in the appendix. We conclude by outlining further paths of development.

2 Failure-slices

In the framework of the leftmost computation rule, the query $\leftarrow G$ terminates universally iff the query $\leftarrow G$, false fails finitely. Transforming a program with respect to this query may result in a more explicit characterization of universal termination. However, the current program transformation frameworks like fold/unfold are not able to reduce the responsible program size in a significant manner. We will therefore focus our attention towards approximations in the form of failure-slices.

Definition 1 (Program point). *The clause $h \leftarrow g_1, ..., g_n$ has a program point p_i on the leftmost side of the body and after each goal. A clause with n goals has therefore the following $n+1$ program points: $h \leftarrow p_i, g_1 p_{i+1}, ..., g_n p_{i+n}$. We label all program points of a program in some global order starting with the initial query. Program points in the query are defined analogously. We denote the set of all program points in program P with query Q by $p(P,Q)$.*

Definition 2 (Failure-slice).
*A program S is called a failure-slice of a program P with query Q if S contains all clauses of P and the query Q with the goal "*false*" inserted at some program points. We represent a failure-slice by the subset of program points where "*false*" has not been inserted (i.e., where "*true*" has been inserted).*

The trivial failure-slice is $p(P, Q)$, therefore the program itself. For a program with n program points there are $|\mathcal{P}(p(P, Q))| = 2^n$ possible failure-slices.

Example 1. For predicate list_invdiff/3 the set of program points $p(P, Q)$ is the set of integers $\{0, 1, 2, 3, 4, 5\}$. On the right, the slice $\{0, 2, 4\}$ is shown.

```
← /*P0*/ list_invdiff(Xs, [1,2,3], []). % P5
list_invdiff([], Xs, Xs). % P1
list_invdiff([E|Es], Xs0, Xs) ← % P2
   list_invdiff(Es, Xs0, Xs1), % P3
   Xs1 = [E|Xs]. % P4
```

```
← list_invdiff(Xs, [1,2,3], []), false.
list_invdiff([], Xs, Xs) ← false.
list_invdiff([E|Es], Xs0, Xs) ←
   list_invdiff(Es, Xs0, Xs1), false,
   Xs1 = [E|Xs].
```

Definition 3 (Partial order). *A failure-slice S is smaller than T if $S \subset T$.*

Theorem 1. *Let P be a definite program with query Q and let S and T be failure-slices of P, Q with $S \subseteq T$. If Q does not left-terminate in S then Q does not left-terminate in T.*

Proof. Consider the SLD-tree for the query Q in S. Since Q does not terminate, the SLD-tree is infinite. The SLD-tree for T contains all branches of S and therefore will also be infinite. $\square$

Definition 4 (Sufficient explanation). *A sufficient explanation E is a subset of $\mathcal{P}(p(P, Q))$ such that for every non-terminating slice $S \notin E$, there is a non-terminating slice $T \in E$ such that $T \subset S$. The trivial sufficient explanation is $\mathcal{P}(p(P, Q))$.*

Example 2. A sufficient explanation of list_invdiff/3 is $\{\{0,1\}, \{5\}, \{0,2\}, \{0,2,4\}\}$. The slices $\{0,1\}$ and $\{5\}$ are terminating and therefore cannot help to explain non-termination. Slice $\{0,2,4\}$ is a superset of $\{0,2\}$. Some other non-terminating slices are $\{0, 2, 3\}, ..., \{0, 1, 2\}, ..., \{0, 1, 2, 3, 4, 5\}$. We note that there always exists a unique smallest sufficient explanation gathering all the minimal failure-slices.

Definition 5 (Minimal explanation). *The minimal explanation is the sufficient explanation with minimal cardinality.*

The minimal explanation contains only non-terminating slices that form an anti-chain (i.e., that are not included in each other). In our example, $\{\{0,2\}\}$ is the minimal explanation since all other non-terminating slices are supersets of $\{0,2\}$.

The minimal explanation is an adequate explanation of non-termination, since it contains all minimal slices that imply the non-termination of the whole program with the given query. It helps to correct the program, because in all minimal slices some parts highlighted by the minimal explanation must be changed in order to avoid non-termination. As long as the highlighted part remains completely unchanged, the very same non-terminating failure-slice can be produced. Further, in our experience, minimal explanations are very small compared to the set of possible explanations. For example, the minimal explanation of the program in the appendix contains one out of 128 possible slices.

Proposition 1. *Q left-terminates w.r.t. P iff the minimal explanation of P, Q is empty.*

The undecidability of termination therefore immediately implies that minimal explanations cannot be determined in general. For this reason we approach the problem from two different directions. First, we focus on determining small slices. Second, we try to obtain *a proof of (universal) non-termination* for each slice in the explanation. If all slices are non-terminating, the minimal set has been calculated.

Currently, we use a simple loop checker for proving universal non-termination that aborts signaling non-termination if a subsuming variant A of an atom A' that occurred in an earlier goal is considered. While this loop check may incorrectly prune some solutions ([1], e.g., ex. 2.1.6), it is sufficient to prove universal non-termination.

The first major obstacle when searching for a non-terminating failure slice is the large search space that has to be considered whereas the size of the minimal explanation is typically very small. For a program with n points there are 2^n different slices, most of them being not interesting, either because they are terminating or because there is a smaller slice that describes the same properties.

3 Failure propagation

In order to narrow down the set of potential slices, we formulate some criteria that must hold for slices in the minimal explanation. With the help of these rules, many slices are removed that can never be part of the minimal explanation. These rules are directly implemented, by imposing the corresponding constraints on the program points that are represented with boolean variables.

Throughout the following rules we use the following names for program points. An *entry/exit point* of a predicate is a program point immediately before/after a goal for that predicate in some clause body or the initial query. A *beginning/ending point* is the first/last program point in a clause.

Right-propagating rules

Program points that will never be used do not occur in a slice of the minimal explanation. The following rules determine some of them. These rules encode the leftmost computation rule.

R1: Unused points. *In a clause, a failing program point p_i implies the next point p_{i+1} to fail.*

- R2: Unused predicates. *If all entry points of a predicate fail, all corresponding beginning program points fail.*
- R3: Unused points after failing definition. *If in all clauses of a predicate the ending program points fail, then all corresponding exit points fail.*
- R4: Unused points of recursive clauses. *If in all clauses that do not contain a direct recursion the ending points fail, then all ending points fail.* A predicate can only be true, if its definition contains at least one non recursive clause.

Left-propagating rules

Program points that are only part of a finite failure branch cannot be part of a slice in the minimal explanation.

- L1: Failing definitions. *If all beginning program points of a predicate fail then all entry points fail.*
- L2: Propagation over terminating goals. *A failing program point p_{i+1} implies p_i to fail if g_{i+1} terminates.* Note that safe approximations of the termination of a goal are described below.
- L3: Left-propagation of failing exit points. *If all exit points of a predicate except those after an tail-recursion fail, then all ending points fail.*

Local recursions

Some infinite loops can be immediately detected by a clausewise inspection. Currently we consider only direct left recursions.

- M1: Local left recursion. *In a clause of the form $h \leftarrow g_1, ..., g_n, g, ...$ a failure is inserted after goal g, if for all substitution θ, $g\theta$ is unifiable with h, and the sequence of goals $g_1, ..., g_n$ can never fail.* Also in this case it is ensured that the program never terminates.

Example 3. In the following clause, rule M1 sets the program point after the recursive goal unconditionally to false. Thereby also the end point is set to false due to rule R1.

ancestor_of(Anc,Desc) ←
  ancestor_of(Child, Desc),
  child_of(Child, Anc).

ancestor_of(Anc,Desc) ←
  ancestor_of(Child, Desc), **false,**
  ~~child_of(Child, Anc), **false**.~~

A detailed example that shows the usage of the other rules is given in the appendix.

Proposition 2 (Soundness of propagating rules). *If a slice is eliminated with the above rules, this slice does not occur in the minimal explanation.*

Proof. For the right propagating rules R1-R4 it is evident that the mentioned program points will never be used with the leftmost computation rule. Therefore the program points may be either true or false, the minimal explanation therefore will contain false.

The left propagating rules prune some finite failure branches. The minimal explanation will not contain these branches. The main idea is therefore to ensure that all infinite parts are preserved.

L1: When all beginning program points fail, a finite failure branch is encountered. By setting all entry points to false, only this finite branch is eliminated.

L2: A terminating goal with a subsequent false generates a finite failure branch, which thus can be eliminated completely.

L3: Consider first the simpler case, when all exit points of a predicate fail. In this case, all ending points may be true or false, without removing a branch. A minimal slice therefore will contain just false at these places.

The ending point in a tail-recursive clause has no impact on the failure branch generated by the predicate, as long as all exit points are false.

M1: This rule describes a never terminating local left recursion. The minimal explanation may thus contain this loop. The subsequent points after the loop are therefore not needed in a minimal slice. □

While the presented rules can be used to generate a sufficient explanation, they are still too general to characterize a minimal explanation. In particular, since all rules except M1 do not take information about the arguments into account.

Safe approximation of termination

Rule L2 needs a safe approximation for the termination property. If the call graph of a (sliced) predicate does not contain cycles, the predicate will always terminate. For many simple programs (like perm/2 described in the annex), the minimal explanation can already be determined with this simple approximation that does not take the information about data flow into account.

For many recursive predicates, however, this very coarse approximation leads to imprecise results, yielding a large sufficient explanation. We sketch the approach we are currently evaluating to combine our constraint based termination analysis [9] with rule L2. We recall that the mentioned termination prover infers for each predicate p a boolean term C_t called its *termination condition.* If the boolean version of a goal $\leftarrow p(\widetilde{t})$ entails C_t then universal left-termination of the original goal $\leftarrow p(\widetilde{t})$ is ensured.

In order to apply rule L2 w.r.t. P, Q, we first tabulate [10] the boolean version of P, Q. Then, if all boolean call patterns for p entail C_t, rule L2 can be safely applied to such goals.

4 Implementation

Our implementation uses finite domain constraints to encode the relations between the program points. Every program point is represented by a boolean 0/1-variable where 0 means that the program point fails. In addition every predicate has a variable whose value indicates whether that predicate is always terminating or not. We refer to the appendix for a complete example.

4.1 Encoding the always-terminating property

While it is possible to express cycles in finite domains directly, they are not efficiently reified in the current CLP(FD) implementation of SICStus-Prolog [3]. For this reason we use a separate pass for detecting goals that are part of a cycle. These goals are (as an approximation) not always-terminating.

A predicate is now always-terminating if it contains only goals that are always-terminating. The encoding in finite domain constraints is straightforward. Each predicate gets a variable AlwTerm. In the following example we assume that the predicates r/0 and s/0 do not form a cycle with q/0. So only q/0 forms a cycle with itself.

q $\leftarrow$ (P0) r, (P1) s, (P2) q (P3).

$$\text{AlwTermQ} \Leftrightarrow (\neg \text{P0} \vee \text{AlwTermR}) \wedge (\neg \text{P1} \vee \text{AlwTermS}) \wedge (\neg \text{P2} \vee 0)$$

If a separate termination analysis is able to determine that q/0 terminates for all uses in P, Q, the value of AlwTermQ can already set accordingly.

4.2 Failure propagation

The rules for minimal explanations can be encoded in a straightforward manner. For example rule R1 is encoded for predicate q/1 as follows:

$\neg$P0 $\Rightarrow$ $\neg$P1, $\neg$P1 $\Rightarrow$ $\neg$P2, $\neg$P2 $\Rightarrow$ $\neg$P3.

4.3 Labeling and weighting

Since we are interested in obtaining minimal explanations, we use the number of program points as a weight to ensure that the smallest slices are considered first. The most straightforward approach simply tries to maximize the number of failing program points. To further order slices with the same number of program points, we prefer those slices that contain a minimal number of predicates. Therefore we use three weights in the following order.

1. minimal number of program points that succeed
2. minimal number of predicates
3. minimal number of clauses

These weights lead naturally to an implementation in finite domain constraints. By labeling these weights in the above order, we obtain minimal solutions first. Further only those solutions that are no extension to already found minimal slices are considered.

4.4 Execution of failure-slices

With the analysis so far we are already able to reduce the number of potentially non-terminating failure-slices. However, our analysis just as any termination analysis is only an approximation to the actual program behavior. Since failure-slices are executable we execute the remaining slices to detect potentially non-terminating slices. With the help of the built-in time_out/3 in SICStus Prolog a goal can be executed for a limited amount of time. In most situations the failure-slices will detect termination very quickly because the search space of the failure is significantly smaller than the original program.

Instead of compiling every failure-slice for execution we use a single enhanced program —a generic failure-slice— which is able to emulate all failure-slices in an efficient manner.

Generic failure-slice. All clauses of the program are mapped to clauses with a further auxiliary argument that holds a failure-vector, a structure with sufficient arity to hold all program points. At every program point n a goal arg(n,FVect,1) is inserted. This goal will succeed only if the corresponding argument of the failure-vector is equal to 1.

```
p(...) ←     slicep(...,FVect) ←
                 arg(n1,FVect,1),
    q(...),      sliceq(...,FVect),
                 arg(n2,FVect,1),
    ...,         ...,
                 arg(ni,FVect,1),
    r(...).      slicer(...,FVect),
                 arg(ni+1,FVect,1).
```

4.5 Proof of non-termination

Slices that are part of a minimal explanation must all be non terminating. To this end, we execute the slice with a simple loop checker under a timeout. Since our loop checker is significantly slower than direct execution, we use it as the last phase in our system.

5 Full Prolog

In this section we will extend the notion of failure-slices to full Prolog. To some extent this will reduce the usefulness of failure-slices for programs using impure features heavily.

5.1 Finite domain constraints

Failure-slices are often very effective for CLP(FD) programs. Accidental loops often occur in parts that set up the constraints. For the programmer it is difficult to see whether the program loops or simply spends its time in labeling. Since labeling is usually guaranteed to terminate, removing the labeling from the program will uncover the actual error. No special treatment is currently performed for finite domain constraints. However, we remark that certain non-recursive queries effectively do not terminate in SICStus Prolog (or require a very large amount of time) like ← S # >0, S # > T, T # > S. Examples like this cannot be detected with our current approach. If such goals appear in an non-recursive part of the program, they will not show up in a failure-slice.

5.2 DCGs

Definite clause grammars can be sliced in the same way as regular Prolog predicates. Instead of the goal false, the escape {false} is inserted.

```
← phrase(rnaloop, Bases).
rnaloop ⟶
    {Bs = [_,_,_|_]},
    complseq(Bs), {false},
    ~~list([_,_,_|_])~~,
    ~~list(Bs)~~.

~~list([]) ⟶ {false},~~
    ~~[]~~.
~~list([E|Es]) ⟶ {false},~~
    ~~[E]~~,
    ~~list(Es)~~.

~~complseq([]) ⟶ {false},~~
    ~~[]~~.
complseq([B|Bs]) ⟶
    complseq(Bs), {false},
    ~~[C]~~,
    ~~{base_compl(C,B)}~~.

~~base_compl(0'A,0'T) ← false~~.
~~base_compl(0'T,0'A) ← false~~.
~~base_compl(0'C,0'G) ← false~~.
~~base_compl(0'G,0'C) ← false~~.
```

5.3 Moded built-ins

Built-ins that can only be used in a certain mode like is/2 pose no problems, since failure-slices do not alter actual modes.

5.4 Cut

The cut operator is a heritage of the early years of logic programming. Its semantics prevents an effective analysis because for general usages cuts require to reason about existential termination. Existential termination may be expressed in terms of universal termination with the help of the cut operator. A goal G terminates existentially if the conjunction G, ! terminates universally. For this reason, goals in the scope of cuts and all the predicates the goal depends on must not be sliced at all. A simple cut at the level of the query therefore prevents slicing completely.

C1: Goals in front of cuts and all its depending predicates must not contain failure points. There must be no loop check in this part.
C2: In the last clause of a predicate, failure points can be inserted anywhere. By successively applying this rule, the slice of the program may be still reduced.
C3: In all other clauses failures may only be inserted after all cuts.

Notice that these restrictions primarily hinder analysis when using deep cuts. Using recommended shallow cuts [6] does not have such a negative impact. In the deriv-benchmark for example, there are only shallow cuts right after the head. Therefore, only program points after the cuts can be made to fail besides from clauses at the end.

```
d(U+V,X,DU+DV) ←
   !,
   d(U,X,DU), false,
   ~~d(V,X,DV)~~.
~~d(U-V,X,DU-DV) ← false~~,
   ~~!~~,
   ~~d(U,X,DU)~~,
   ~~d(V,X,DV)~~.
```

5.5 Negation

Similar to cuts Prolog's unsound negation built-in \+/1 "not" is handled. The goal occurring in the "not" and all the predicates it depends on must not contain any injected failures. Similarly "if-then-else" and if/3 are treated. The second order predicates setof/3 and findall/3 permit a more elaborate treatment. The program point directly after such goals is the same as the one within findall/3 and setof/3. Therefore, failure may be propagated from right to left.

5.6 Side effects

Side effects must not be present in a failure-slice. However, this does not exclude the analysis of predicates with side effects completely. When built-ins only produce side effects that cannot affect Prolog's control (e.g. a simple write onto a log file provided that Prolog does not read that file, or reading something once from a constant file) still some failure-slice may be produced. Before such side effecting goals a failure is injected, therefore ensuring that the side effect is not part of the failure-slice. We note that the classification into harmless and harmful side effects relies on the operating system environment and is therefore beyond the scope of a programming language.

6 Summary

To summarize our approach, slicing proceeds in the following manner:

1. The call graph is analyzed to detect goals that are part of a cycle.
2. A predicate fvectPQ_weights(FVect,Weights) is generated and compiled. It describes the relation between the program points in P with respect to the query Q with the help of finite domain constraints. All program points are represented as variables in the failure-vector FVect. FVect therefore represents the failure-slice. Weights is a list of values that are functions of the program points. Currently three weights are used: The number of predicates, the number of clauses and the number of succeeding program points.

3. The generic failure-slice is generated and compiled.
4. Now the following query is executed to find failure-slices.

```
← fvectPQ_weights(FVect, Weights),
    FVect =.. [_|Fs],
    labeling([], Weights),
    labeling([], Fs),
    time_out(slicePQ(...,FVect), t, time_out),
    loopingslicePQ(...,FVect,Result).
```

   Procedurally the following happens:
   (a) fvectPQ_weights/2 imposes the constraints within FVect and Weights.
   (b) An assignment for the weights is searched, starting from minimal values.
   (c) An assignment for the program points in the failure vector is searched. A potential failure-slice is thus generated.
   (d) The failure-slice is actually run for a limited amount of time to discard some terminating slices.
   (e) The loop checker is uses to determine if non-termination can be proven.

The analysis thus executes on the fly while searching for failure-slices.

7 Conclusion and future work

We presented a slicing approach for termination that combines both static and dynamic techniques. For the static analysis we used finite domain constraints which turned out to be an effective tool for our task. Usual static analysis considers a single given program. By using constraints we were able to consider a large set of programs *at the same time*, thereby reducing the inherent search space considerably. Since failure-slices are executable their execution helps to discard terminating slices.

Tighter integration of termination proofs. Our approach might be further refined by termination proofs. In principle, any system for proving termination could be integrated in our system to test whether a particular slice terminates. In this manner some more terminating slices can be eliminated from a sufficient explanation. There are however several obstacles to such an approach. First, most termination proofs are rather costly, in particular, when a large set of slices is detected as terminating. We consider using a constraint based approach as presented in [9] that will be parameterized by the program points. We expect a significantly more efficient implementation than those that tests for termination at the latest possible moment. On the other hand, the precision of the analysis should not suffer from this generalization.

Stronger rules constraining sufficient explanations. In another direction we are investigating to formulate strong rules to constrain the search space of sufficient explanations. In particular in programs as ancestor_of/2, there is still an exponential number of slices that must be tested dynamically. We envisage the usage of dependencies between alternate clauses to overcome this problem.

Argument slicing. The existing slicing approaches [17, 5, 14] all perform argument slicing. We have currently no implementation of argument slicing. While it improves program understanding, argument slicing does not seem to be helpful for further reducing the number of clauses or program points. It appears preferable to perform argument slicing after a failure-slice has been found.

Acknowledgments. The initial work on failure-slices was done within INTAS project INTAS-93-1702.

References

1. R. N. Bol. Loop Checking in Logic Programming. Thesis, Univ. Amsterdam, 1991.
2. L. Byrd. Understanding the control flow of Prolog programs. Logic Programming Workshop, Debrecen, Hungary, 1980.
3. M. Carlsson, G. Ottosson and B. Carlson. An Open-Ended Finite Domain Constraint Solver. PLILP'97 (H. Glaser, P. Hartel and H. Kuchen Eds.), 191–206, LNCS 1292, Springer-Verlag, 1997.
4. P. Deransart and J. Maluszynski. A Grammatical View of Logic Programming. MIT-Press, 1993.
5. T. Gyimóthy and J. Paakki. Static slicing of logic programs. AADEBUG 95 (M. Ducassé Ed.), 87–103, IRISA-CNRS, 1995.
6. R. A. O'Keefe. The Craft of Prolog. MIT-Press, 1990.
7. L. Plümer. Termination Proofs for Logic Programs, LNAI 446, Springer, 1990.
8. T. Vasak, J. Potter. Characterization of Termination Logic Programs, IEEE SLP, 140–147, 1986.
9. F. Mesnard. Inferring Left-terminating Classes of Queries for Constraint Logic Programs. JICSLP'96 (M. Maher Ed.), 7–21, MIT-Press, 1996.
10. F. Mesnard and S. Hoarau. A tabulation algorithm for CLP. *Proc. of the 1st International Workshop on Tabling in Logic Programming, Leuven*, 1997. Revised report `www.univ-reunion.fr/~gcc`.
11. U. Neumerkel. Mathematische Logik und logikorientierte Programmierung, Skriptum zur Laborübung, 1993-1997.
12. U. Neumerkel. Teaching Prolog and CLP. Tutorial. PAP'95 Paris, 1995 and ICLP'97 Leuven, 1997.
13. U. Neumerkel. GUPU: A Prolog course environment and its programming methodology. Proc. of the Poster Session at JICSLP'96 (N. Fuchs and U. Geske Eds.), GMD-Studien Nr. 296, Bonn, 1996.
14. St. Schoenig, M. Ducassé. A Backward Slicing Algorithm for Prolog. SAS 1996 (R. Cousot and D. Schmidt Eds.), 317–331, LNCS 1145, Springer-Verlag, 1996.
15. M. Weiser. Programmers Use Slices When Debugging. CACM 25(7), 446–452, 1982.
16. M. Weiser. Program Slicing. IEEE TSE 10(4), 352–357, 1984.
17. J. Zhao, J. Cheng, and K. Ushijima. Literal Dependence Net and Its Use in Concurrent Logic Programming Environment: Proc. Workshop on Parallel Logic Programming FGCS'94, pp.127–141, Tokyo, 1994.

A Failure-slices for perm/2

% Original program
perm([], []). % P1
perm(Xs, [X|Ys]) ← % P2
    del(X, Xs, Zs), % P3
    perm(Zs, Ys). % P4

del(X, [X|Xs], Xs). % P5
del(X, [Y|Ys], [Y|Xs]) ← % P6
    del(X, Ys, Xs). % P7

← perm(Xs, [1,2]). % P0, P8

% First failure-slice {2,6}
~~perm([], []) ← false~~.
perm(Xs, [X|Ys]) ←
    del(X, Xs, Zs), false,
    ~~perm(Zs, Ys), false~~.

~~del(X, [X|Xs], Xs) ← false~~.
del(X, [Y|Ys], [Y|Xs]) ←
    del(X, Ys, Xs), false.

← perm(Xs, [1,2]), false.
% does not terminate

% 2nd failure-slice {2,3,5}
~~perm([], []) ← false~~.
perm(Xs, [X|Ys]) ←
    del(X, Xs, Zs),
    perm(Zs, Ys), false.

del(X, [X|Xs], Xs).
~~del(X, [Y|Ys], [Y|Xs]) ← false~~,
    ~~del(X, Ys, Xs), false~~.

← perm(Xs, [1,2]), false.
% terminates

```
% Determination of failure-slices to be tested for termination/non-termination
← fvectPQ_weights(FVect,Wghts), FVect=..[_|Ps], labeling([],Wghts), labeling([],Ps).
% FVect = s(0,1,0,0,0,1,0,0), Wghts = [3,2,2]. % {2,6} does not terminate
% FVect = s(0,1,1,0,1,0,0,0), Wghts = [4,2,2]. % {2,3,5} terminates; deleted
% FVect = s(0,1,1,0,1,1,0,0), Wghts = [5,2,3]. % {2,3,5,6} ⊃ {2,6}; not considered
% FVect = s(0,1,1,0,1,1,1,0), Wghts = [6,2,3]. % {2,3,5,6,7} ⊃ {2,6}; not considered
%       ⇒ The minimal explanation E = {{2,6}}

% Definition of the failure-vector
fvectPQ_weights(s(P1, P2, P3, P4, P5, P6, P7), [NPoints, NPreds, NClauses]) ←
    domain_zs(0..1, [P1, P2, P3, P4, P5, P6, P7]),
    P0 = 1, P8 = 0, % Given Query

    % R1: unused points in clause
    ¬P2 ⇒ ¬P3, ¬P3 ⇒ ¬P4, ¬P6 ⇒ ¬P7,
    % R2: unused predicates
    /*perm/2:*/ ¬P0 ∧ ¬P3 ⇒ ¬P1 ∧ ¬P2, /*del/3:*/ ¬P2 ∧ ¬P6 ⇒ ¬P5 ∧ ¬P6,
    % R3: failing definition
    /*perm/2:*/ ¬P1 ∧ ¬P4 ⇒ ¬P8 ∧ ¬P4, /*del/3:*/ ¬P5 ∧ ¬P7 ⇒ ¬P3 ∧ ¬P7,
    % R4: right propagation into recursive clause
    /*perm/2:*/ ¬P1 ⇒ ¬P4, /*del/3:*/ ¬P5 ⇒ ¬P7,

    % L1: failing definition
    /*perm/2:*/ ¬P1 ∧ ¬P2 ⇒ ¬P3 ∧ ¬P0, /*del/3:*/ ¬P5 ∧ ¬P6 ⇒ ¬P2 ∧ ¬P7,
    % L2: over (always) terminating goals
    ¬P4 ∧ AlwTermPerm ⇒ ¬P3, ¬P8 ∧ AlwTermPerm ⇒ ¬P0,
    ¬P3 ∧ AlwTermDel ⇒ ¬P2, ¬P7 ∧ AlwTermDel ⇒ ¬P6,
    % L3: failing exit points
    ¬P8 ⇒ ¬P1 ∧ ¬P4, ¬P3 ⇒ ¬P5 ∧ ¬P7,

    % Always terminating
    AlwTermPerm ⇔ (¬P2∨AlwTermDel) ∧ (¬P3∨0), AlwTermDel ⇔ (¬P6∨0),

    % Weights:
    NPreds #= min(1,P1+P2) + min(1,P5+P6),
    NClauses #= P1+P2+P5+P6,
    NPoints #= P0+P1+P2+P3+P4+P5+P6+P7+P8.
```