Termination slicing in logic programs

Ulrich Neumerkel[1]

Technische Universität Wien
Institut für Computersprachen
A-1040 Wien, Austria
ulrich@mips.complang.tuwien.ac.at

Abstract. In this paper we present a slicing approach for termination analysis of logic programs. The notion of a failure-slice is introduced which is an executable reduced part of the program. Each failure-slice represents a necessary termination condition for the program. The minimal subset of failure-slices that constitutes a sufficient termination condition is approximated by a combination of static and dynamic techniques. A global analysis using finite domain constraints is combined with the actual execution of some failure-slices. In this manner accurate explanations are derived automatically.

1 Introduction

Slicing [14] is an analysis technique to extract parts of a program related to a particular computation. Recently, slicing has been adopted to logic programming languages [2, 12, 17]. The major uses of slicing are debugging and program understanding. While the current approaches focus on explaining (possibly erroneous) solutions of a query, we will present a slicing technique for characterizing and explaining termination properties. It is an implementation of a previously developed informal reading technique used in Prolog-courses [9, 10] and is currently integrated into a programming environment for beginners [11].

Our approach to termination differs from current research in termination analysis of logic programs that 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 [7]. 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. With the help of failure-slices we describe the precise class of terminating queries. Failure-slices expose those parts of the program that may cause nontermination.

We will consider primarily the property of universal left termination. In contrast to existential termination [6], the property of universal termination has several advantages for program development and understanding. Roughly, 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 [5]

the conjunction of two universally terminating goals always terminates. Further, reordering and duplicating clauses has no influence.

Example. The following example slightly adopted from [16] contains an erroneous data base causing universal nontermination of the given query. Its nontermination cannot be easily observed by inspecting the sequence of produced solutions. Looking at the first solutions suggests a correct implementation. But in fact, an infinite sequence of redundant solutions is produced for a query like ancestor(terach,D). On the other hand, the query ancestor(terach,moses) neither produces a solution nor terminates.

The automatically generated failure-slice on the right locates the reason for nontermination by hiding irrelevant parts. The remaining failure-slice is a necessary and in this case even sufficient termination condition for the original program.

```
% original program                    % failure slice
parent(terach, abraham).              parent(terach, abraham).
parent(terach, nachor).               ~~parent(terach, nachor) ← fail.~~
parent(abraham, isaac).               parent(abraham, isaac).
parent(nachor, lot).                  ~~parent(nachor, lot) ← fail.~~
parent(isaac, terach). % error        parent(isaac, terach).

ancestor(Ancestor, Descendant) ←      ~~ancestor(Ancestor, Descendant) ← fail,~~
   parent(Ancestor, Descendant).         ~~parent(Ancestor, Descendant).~~
ancestor(Ancestor, Descendant) ←      ancestor(Ancestor, Descendant) ←
   parent(Ancestor, Person),             parent(Ancestor, Person),
   ancestor(Person, Descendant).         ancestor(Person, Descendant), fail.

← ancestor(Ancestor, Descendant).     ← ancestor(Ancestor, Descendant), fail.
```

The failure-slice helps significantly in understanding the program's termination property. It shows for example that clause reordering does not help and that the first rule in ancestor/2 is not responsible for termination. Often beginners have this incorrect belief confusing universal and existential termination.

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 nontermination, on the other hand, since such an analysis can only approximate the minimal slices we need an efficient way to generate all interesting slices which then are tested for termination by mere execution. With the help of this combination of analysis and execution we obtain explanations also when classical termination analysis cannot produce satisfying results.

Contents. The notion of a failure-slice and its refinements are defined in Sect. 2. Sect. 3 discusses how our approach is adapted to handle many aspects of full Prolog. The actual implementation of the global analysis using finite domain constraints is presented in 4, 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 "← G" terminates universally iff the query "← G, fail" 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 $n + 1$ points: $h \leftarrow p_0 g_1, p_1 ..., g_n p_n$

Definition 2 Failure-slice. A failure-slice of a single clause is obtained by inserting the goal "fail" at some program points. The trivial failure-slice is therefore the clause itself. A program S is called a failure-slice of a program P if S contains failure-slices of all clauses of P.

A failure slice is therefore obtained by inserting the goal "fail" at some program points.

Theorem 3. *Let P be a definite program, Q a query and S a failure slice of P: If Q does not terminate in S then Q does not terminate in P.*

To see this, consider the SLD-tree for the query Q in S. Since Q does not terminate, the SLD-tree is infinite. The SLD-tree for P contains all branches of S and therefore will also be infinite.

Failure-slices can be used as an explanation for the actual termination behavior. For a program with n program points there are 2^n possible failure-slices and explanations. Many of these slices are not interesting because they either terminate or are a simple variant of other slices possessing the same operational behavior. In the following we will define interesting failure-slices.

Throughout the following rules we use the following names for program points: An *entry point* of a predicate is the program point before a goal of that predicate. The *exit point* is the program point after that goal. The *beginning point* is the first program point in a clause. The *ending point* is the last program point of a clause.

Definition 4 Always-terminating goal. A goal is always-terminating if its call graph does not contain a loop.

Definition 5 Interesting failure-slice. The set of program points of an interesting failure-slice must satisfy the following conditions. According to their propagation direction we have the following rules.

Right-propagating rules

R1: Clausewise right-propagation of failure. A failing program point p_i implies p_{i+1} to fail. This rule encodes the left most computation rule†. If a failing goal is encountered all subsequent goals are not considered.
R2: Right-propagation of failing entry points. If all entry points of a predicate fail, the beginning program point of all clauses fail.
R3: Right-propagation of failing definition. If in all clauses of a predicate the ending program points fail, then all exit points of the program fail.
R4: Right-propagation of failure into end-recursive clauses. If in all clauses that do not contain an end-recursion the ending points fail, then all ending points fail.

Left-propagating rules

L1: Left-propagation of failing definition. If all beginning program points of a predicate fail then all entry points fail.
L2: Clausewise left-propagation of failure over always-terminating goals. A failing program point p_{i+1} implies p_i to fail if g_i is always-terminating.
L3: Left-propagation of failing exit points. If all exits points of a predicate except those after an end-recursion fail, then all ending points fail.

3 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.

3.1 DCGs

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

```
rnaloop −→
    {Bs = [_,_,_|_]},
    complseq(Bs), {fail},
    list([_,_,_|_]),
    list(Bs).

list([]) −→ {fail},
    [].
list([E|Es]) −→ {fail},
    [E],
    list(Es).
← phrase(rnaloop, Bases).
```

```
complseq([]) −→ {fail},
    [].
complseq([B|Bs]) −→
    complseq(Bs), {fail},
    [C],
    {base_compl(C,B)}.

base_compl(0'A,0'T) <− fail.
base_compl(0'T,0'A) <− fail.
base_compl(0'C,0'G) <− fail.
base_compl(0'G,0'C) <− fail.
```

3.2 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.

3.3 Moded built-ins

Built-ins that can only be used in a certain mode like is/2 restrict the applicability of our approach. Whereas in pure Prolog the most general query will outline also the termination condition for more special queries, general executable failure slices cannot be produced for moded built-ins.

3.4 Cut

The cut operator is a heritage of the early years of logic programming. Its semantics prevents an effective analysis because cuts in general 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.
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 [4] 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), fail,
    ~~d(V,X,DV)~~.
~~d(U+V,X,DU+DV) ← fail~~,
    ‡,
    ~~d(U,X,DU)~~,
    ~~d(V,X,DV)~~.
... .
```

3.5 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.

3.6 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.

4 Implementation

Our implementation is using finite domain constraints to encode the relation between the program points. Every program point is represented by a boolean 0/1-variable. 0 meaning that the program point fails. In addition every predicate has a variable that says, 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 [1]. For this reason we use a separate pass for detecting goals that are part of a cycle. To each predicate a level is associated. The level of a predicate is smaller or equal to the levels of the predicates it depends on. The assignment with the minimal number of predicates at the same level is searched. If two predicates are on the same level they are part of a cycle. In this manner we identify all goals that are part of cycles. 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 straight forward. Each predicate gets a variable AllTerm. 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 ← (P0) r, (P1) s, (P2) q (P3).

AllTermQ ⇔ (¬P0 ∨ AllTermR) ∧ (¬P1 ∨ AllTermS) ∧ (¬P2 ∨ 0)

4.2 Interesting failure-slices

The rules for interesting failure-slices can be encoded in a straight forward manner. For example rule R1 is encoded for predicate q/1 as follows:

¬P0 ⇒ ¬P1, ¬P1 ⇒ ¬P2, ¬P2 ⇒ ¬P3.

4.3 Weighting

Since we use failure-slices for explanations we are interested in minimal slices. The most straight-forward approach simply tries to maximize the number of failing program points. Often these slices are not very easy to read because they still contain many different clauses and predicates. For these reasons we use three weights in the following order.

1. minimal number of predicates
2. minimal number of clauses
3. minimal number of succeeding program points

These weights lead naturally to an implementation in finite domain constraints. By labeling these weights in the above order, we obtain minimal solutions first.

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 is only an approximation to the actual program behavior. Since failure-slices are executable we execute the remaining slices to detect potentially nonterminating 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 Summary

To summarize, our approach to termination 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 that 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 interesting failure slices.

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

   Procedurally the following happens:
   (a) fvectPQ_weights/2 imposes the constraints within FVect and Weights.
   (b) The first assignment for the Weights is searched, starting from minimal values. Therefore, failure-slices with the minimal number of predicates etc. are generated first.
   (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. If the failure-slice does not terminate within that period the failure-slice is discarded.

The control flow analysis is therefore executed on the fly while searching for failure-slices.

5 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.

Integrating termination proofs. Our approach might be refined by integrating termination proofs into our system. In this manner failure-slices that are guaranteed to terminate could be discarded. The most promising candidate seems to be Mesnard's approach [3] which uses constraints in a similar manner. It could probably be used to further constrain the failure vector of the program points. Another possible direction would be to try to fold different failure slices with the help of fold/unfold-transformations.

Redundant slices. Many different failure-slices have the same termination property. For example, in the case of the predicate perm/2 in the appendix there are four different interesting failure-slices. All four represent the very same termination class. Trying to remove some of them seems to be an interesting future direction.

Argument slicing. The existing slicing approaches 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 therefore preferably to perform argument slicing after a failure-slice has been found.

References

1. M. Carlsson, G. Ottosson, B. Carlson, An Open-Ended Finite Domain Constraint Solver. PLILP, 1997.
2. T. Gyimóthy and J. Paakki. Static slicing of logic programs. AADEBUG 95.
3. S. Hoarau, F. Mesnard, Inferring and Compiling Termination for Constraint Logic Programs, Technical Report 1998 (to appear).
4. R. A. O'Keefe, The Craft of Prolog. 1990, MIT-Press.
5. L. Plümer, Termination Proofs for Logic Programs, LNAI 446, Springer-Verlag, 1990.
6. T. Vasak, J. Potter, Characterization of Termination Logic Programs, IEEE SLP, 1986.

7. F. Mesnard, Inferring Left-terminating Classes of Queries for Constraint Logic Programs: JICSLP'96 7–21, 1996, MIT-Press.
8. L. Naish, Types and the Intended Meaning of Logic. in F. Pfenning (ed.) Types in Logic Programming, 1992, MIT-Press.
9. U. Neumerkel. Mathematische Logik und logikorientierte Programmierung, Skriptum zur Laborübung, 1993-1997.
10. U. Neumerkel Teaching Prolog and CLP. Tutorial. PAP'95 Paris, 1995 and ICLP'97 Leuven, 1997.
11. U. Neumerkel. GUPU: A Prolog course environment and its programming methodology. Proc. of the Poster Session at JICSLP'96 (Fuchs, Geske Eds.), GMD-Studien Nr. 296, 1996 Bonn.
12. St. Schoenig, M. Ducassé: A Backward Slicing Algorithm for Prolog. SAS 1996: 317-331
13. Ch. Speirs, Z. Somogyi, H. Søndergaard: Termination Analysis for Mercury. SAS 1997: 160-171
14. M. Weiser: Programmers Use Slices When Debugging. CACM 25(7): 446-452 (1982)
15. M. Weiser: Program Slicing. IEEE TSE 10(4): 352-357 (1984).
16. L. Sterling, E. Shapiro: The Art of Prolog, MIT-Press 1986.
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, Japan, December 1994
18. J. Zhao, J. Cheng, K. Ushijima: Program Dependence Analysis of Concurrent Logic Programs and Its Applications, Proc. 1996 International Conference on Parallel and Distributed Systems (ICPADS'96) , 282-291, IEEE Computer Society Press, 1996.

A Failure-slices for perm/2

```
% Original program                % Minimal failure-slice          % 2nd failure-slice
perm([], []). % P1               ~~perm([], []) ← fail~~.         ~~perm([], []) ← fail~~.
perm(Xs, [X|Ys]) ← % P2          perm(Xs, [X|Ys]) ←               perm(Xs, [X|Ys]) ←
   del(X, Xs, Zs), % P3             del(X, Xs, Zs), fail,            del(X, Xs, Zs),
   perm(Zs, Ys). % P4               ~~perm(Zs, Ys)~~.                perm(Zs, Ys), fail.

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

← perm(Xs, Ys). % P0, P8         ← perm(Xs, Ys), fail.            ← perm(Xs, Ys), fail.
```

```
% implementation

← fvectPQ_weights(FVect,Weights), FVect=..[_|Ps], labeling([],Weights), labeling([], Ps).
% FVect = s(1,0,1,0,0,0,1,0,0), Weights = [2,2,3]. % displayed above
% FVect = s(1,0,1,1,0,1,0,0,0), Weights = [2,2,4]. % displayed above
% FVect = s(1,0,1,1,0,1,1,0,0), Weights = [2,3,5].
% FVect = s(1,0,1,1,0,1,1,1,0), Weights = [2,3,6].

% Definition of the failure-vector
fvectPQ_weights(s(P0, P1, P2, P3, P4, P5, P6, P7, P8), [NPreds, NClauses, NPoints]) ←
   domain_zs(0..1,[P0, P1, P2, P3, P4, P5, P6, P7, P8]),
   % Given Query
   P0 = 1, P8 = 0,
   % Allways terminating
   AllTermPerm ⇔ ( ¬P2 ∨ AllTermDel ) ∧ ( ¬P3 ∨ 0 ),
   AllTermDel ⇔ ( ¬P6 ∨ 0 ),
   % R1: right propagation of failure
   ¬P2 ⇒ ¬P3, ¬P3 ⇒ ¬P4, ¬P6 ⇒ ¬P7,
   % R2: right propagation of failing entry points
   /*perm/2:*/ ¬P0 ∧ ¬P3 ⇒ ¬P1 ∧ ¬P2, /*del/3:*/ ¬P2 ∧ ¬P6 ⇒ ¬P5 ∧ ¬P6,
   % R3: right propagation of failing definition
   /*perm/2:*/ ¬P1 ∧ ¬P4 ⇒ ¬P8 ∧ ¬P4, /*del/3:*/ ¬P5 ∧ ¬P7 ⇒ ¬P3 ∧ ¬P7,
   % R4: right propagation into end-recursive clause
   /*perm/2:*/ ¬P1 ⇒ ¬P4, /*del/3:*/ ¬P5 ⇒ ¬P7,
   % L1: left propagated of failing definition
   /*perm/2:*/ ¬P1 ∧ ¬P2 ⇒ ¬P3 ∧ ¬P0, /*del/3:*/ ¬P5 ∧ ¬P6 ⇒ ¬P2 ∧ ¬P7,
   % L2: left propagation over always terminating goals
   ¬P4 ∧ AllTermPerm ⇒ ¬P3, ¬P8 ∧ AllTermPerm ⇒ ¬P0,
   ¬P3 ∧ AllTermDel ⇒ ¬P2, ¬P7 ∧ AllTermDel ⇒ ¬P6,
   % L3: left propagating failing exit points
   ¬P8 ⇒ ¬P1 ∧ ¬P4, ¬P3 ⇒ ¬P5 ∧ ¬P7,
   % 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.
```

Table of Contents

This article was processed using the LaTeX macro package with LLNCS style