Slicing nichtterminierender Programme

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

Zusammenfassung

Wir stellen einen Slicing-basierten Ansatz zur Terminationsanalyse von Logikprogrammen vor. Um die Terminationseigenschaften eines Programms zu erklären, werden ausführbare Programmfragmente (failure-slices) bestimmt. Falls eines dieser Programmfragmente nicht terminiert, so terminiert auch das gesamte Programm nicht. Der für die Nichttermination verantwortliche Teil des Programms kann so meist auf einige wenige Klauseln eingegrenzt werden. Zur Bestimmung dieser Programmfragmente verbindet der vorgestellte Ansatz eine globale statische Constraint-basierte Analyse mit der dynamischen Ausführung von Programmfragmenten.

1 Einführung

Prologs Terminationsverhalten ist aufgrund des komplexen Ausführungsmechanismus schwer nachzuvollziehen. Die Verflechtung zweier Kontrollflüsse (UND und ODER), zu denen sich noch gelegentlich ein weiterer für Coroutining gesellt, vereitelt den Versuch, Terminationseigenschaften direkt aus dem Programmablauf Schritt für Schritt abzulesen. Das von Byrd [2] vorgeschlagene *box model* etwa, welches die Grundlage für viele Debugger ist, erzeugt eine Unzahl irrelevanter Ableitungsschritte. Auch der Versuch, Nichttermination durch Ableitungsbäume zu veranschaulichen, hat sich aus ähnlichen Gründen als nicht zielführend erwiesen. Aus der Fülle von irrelevanten Details einer konkreten Ableitung kann man nur schwer die für das allgemeine Terminationsverhalten relevanten Informationen ablesen. Es liegt daher nahe, davon auszugehen, daß die Darstellung des konkreten Ablaufes nicht zielführend ist.

Zum Thema Terminationsverhalten konzentrieren sich gegenwärtige Forschungsarbeiten auf die automatische Erstellung von Terminationsbeweisen. Im einfachsten Fall wird der Beweis für eine vorgegebene Menge von Anfragen durchgeführt. Allgemeinere Ansätze bestimmen diese Menge [10]. In beiden Fällen wird Termination garantiert, so sich ein Beweis findet. Die Menge der tatsächlich terminierenden Anfragen ist jedoch meist viel größer. Allein aufgrund der verwendeten Formalismen, ist eine präzise Beschreibung der Menge aller terminierenden Anfragen selbst bei sehr einfachen Programmen unmöglich. So wird die Menge der terminierenden Anfragen für append/3 dadurch beschrieben, daß im ersten oder letzten Argument die Länge der Liste bekannt sein muß. append/3 terminiert jedoch auch für Listen unbekannter Länge, falls in den Listen an entsprechender Position nicht unifizierbare Elemente vorkommen. Zudem sind diese Arbeiten zur Erklärung der Nichttermination ungeeignet.

Wir stellen einen dazu komplementären Ansatz vor, mit dem wir Nichttermination besser erklären können. Dieser Ansatz wurde bisher nur informell in der Lehre [12, 13] verwendet. Durch die Entwicklung der in diesem Artikel vorgestellten Slicing-Technik konnte er in eine Programmierumgebung [14] integriert werden.

Slicing. Unter *program slicing* versteht man Analysetechniken, um jene Programmfragmente herauszufinden, die für gewisse Eigenschaften (z.B. Laufzeitfehler oder falsche Variablenwerte) verantwortlich sind. Der Begriff wurde von Weiser [21, 22] für imperative Programmiersprachen geprägt. Weiser ging

von der Beobachtung aus, daß Programmierer bei der Fehlersuche den Bereich des Quelltextes, in dem sich ein Fehler befinden muß, eingrenzen, noch bevor sie den konkreten Programmablauf verfolgen. Durch automatisches Slicing kann diese Bürde dem Programmierer teilweise abgenommen werden. Das Verstehen eines Programmes wird erleichert, was vor allem im Bereich Debugging und Reverse Engineering von Interesse ist [1]. Neben diesen Bereichen werden Slicing-Techniken auch im Bereich der Code-Generierung verwendet.

Für Prolog wurden Slicing-Techniken erst in den letzten Jahren entwickelt. Noch Anfang 1994 stellten Ducassé und Noyé in einem Überblicksartikel über Logikprogrammierumgebungen [15] fest, daß diese Techniken noch nicht angenommen wurden. Slicing-Techniken für logikorientiere Programmiersprachen entwickelten daraufhin Zhao [23], Gyimóthy [7] und Ducassé [17]. Diese Techniken beschränken sich auf die Erklärung von möglicherweise fehlerhaften Antworten einer Anfrage durch Erzeugung entsprechender Programmfragmente. Dabei ist vor allem der Ansatz von Ducassé hervorzuheben, in dem die Programmfragmente selbst ausführbare Programme sind. Diesem Ansatz ist es möglich, die Beziehung der Programmfragmente zum ursprünglichen Programm über die Lösungsmengen der beiden Programme zu charakterisieren, während die beiden anderen Ansätze letztlich nur durch den konkreten Ablauf eines Programms gerechtfertigt werden. Wir betrachen in diesem Artikel Programmfragmente, die zur Charakterisierung und Erklärung der Terminationseigenschaft eines Programms geeignet sind.

Wie auch die meisten Arbeiten zu Terminationsbeweisen, beschränken wir uns auf universelle Links-Termination. Eine Anfrage terminiert universell, wenn der gesamte Ableitungsbaum endlich ist. Existentielle Termination [18] hingegen liegt bereits vor, wenn eine Lösung gefunden wird. Existentielle Termination ist wesentlich empfindlicher gegenüber Programmänderungen und ist daher schwer zu analysieren und zu verstehen. Wie Plümer [16] anmerkt, terminiert die Konjunktion zweier universell terminierenden Ziele immer, während dies für existentiell terminierende Ziele nicht gilt. Auch beeinflußt das Umordnen von Klauseln nur die existentielle Termination.

Beispiel. Im folgenden Programm ist das Prädikat kind_von/2 fehlerhaft definiert. Die gegebene Anfrage terminiert nicht universell. Da sie jedoch existentiell terminiert, kann der Fehler nicht unmittelbar beobachtet werden. Auch durch Betrachten der Lösungssequenz ist der Fehler nicht sofort ersichtlich. Die ersten gefundenen Lösungen legen eine korrekte Implementierung nahe. Die unendliche Sequenz redundanter Lösungen ist schwer zu erkennen, da auch viele sinnvolle Lösungen gefunden werden. Redundante Lösungen an sich sind ebenfalls kein eindeutiges Indiz für einen Fehler. Manche konkretere Anfragen terminieren, andere wiederum nicht. So findet die Anfrage vorfahre_von(karl_VI,N) auch die Lösung N = marie_antoinette, umgekehrt jedoch findet vorfahre_von(P,marie_antoinette) nicht die Lösung P = karl_VI. Fehler dieser Art sind nur schwer zu lokalisieren. Ein Debugger würde eine Unzahl von Ableitungsschritten aufzeigen, die nichts mit dem eigentlichen Fehler zu tun haben.

```
kind_von(joseph_I, leopold_I).                  ~~kind_von(joseph_I, leopold_I) ← false.~~
kind_von(karl_VI, leopold_I).                   ~~kind_von(karl_VI, leopold_I) ← false.~~
kind_von(maria_theresia, karl_VI).              kind_von(maria_theresia, karl_VI).
kind_von(joseph_II, maria_theresia).            ~~kind_von(joseph_II, maria_theresia) ← false.~~
kind_von(joseph_II, franz_I).                   ~~kind_von(joseph_II, franz_I) ← false.~~
kind_von(leopold_II, maria_theresia).           kind_von(leopold_II, maria_theresia).
kind_von(leopold_II, franz_I).                  ~~kind_von(leopold_II, franz_I) ← false.~~
kind_von(marie_antoinette, maria_theresia).     ~~kind_von(marie_antoinette, maria_theresia) ← false.~~
kind_von(karl_VI, leopold_II). % Fehler         kind_von(karl_VI, leopold_II). % Fehler

vorfahre_von(Vorfahre, Nachfahre) ←             ~~vorfahre_von(Vorfahre, Nachfahre) ← false,~~
    kind_von(Nachfahre, Vorfahre).                  ~~kind_von(Nachfahre, Vorfahre).~~
vorfahre_von(Vorfahre, Nachfahre) ←             vorfahre_von(Vorfahre, Nachfahre) ←
    kind_von(Person, Vorfahre),                     kind_von(Person, Vorfahre),
    vorfahre_von(Person, Nachfahre).                vorfahre_von(Person, Nachfahre), false.
← vorfahre_von(Vorfahre, Nachfahre).            ← vorfahre_von(Vorfahre, Nachfahre), false.
```

Durch das in diesem Artikel vorgestellte Slicing kann der Bereich des Programms, der für die universelle Nichttermination einer Anfrage verantwortlich ist, stark eingeschränkt werden. In diesem Fall sind nur drei Fakten und die rekursive Regel von vorfahre_von/2 für die Nichttermination verantwortlich. Wenn auch nur eine der verbliebenen Klauseln aus dem Programm entfernt wird, so terminieren alle Anfragen universell. Offenbar muß sich der Fehler in diesen Klauseln befinden.

Das gewonnene Programmfragment (failure-slice) trägt beträchtlich zum Verstehen der Terminationseigenschaft eines Programms bei. So kann daraus ersehen werden, daß das Umordnen von Klauseln keinen Einfluß auf Termination hat. Weiters ist ersichtlich, daß im rekursiven Prädikat vorfahre_von/2 die erste Regel die Terminationseigenschaft nie beeinflußt. Häufig werden Regeln wie diese unrichtigerweise als Abbruchbedingung bezeichnet.

Aus diesem Beispiel können schon die Anforderungen an eine effektive Implementierung des Slicings abgelesen werden. Programmteile, die sicher nicht für Nichttermination verantwortlich sind, sollten durch die Programmanalyse rasch erkannt werden. Da eine Programmanalyse nur eine Annäherung an das tatsächliche Verhalten ist, wird das verbleibende Programmfragment auch ausgeführt werden müssen, um jene Teile, die rasch terminieren, auszusortieren. In diesen Beispiel kommen letztlich alle möglichen Kombinationen der Fakten in Frage. Es ist daher besonders wichtig, daß das probeweise Ausführen vieler verschiedener Programmfragmente effizient realisiert wird.

Übersicht. Die betrachteten Programmfragmente (*failure-slices*) werden vorerst definiert. Es wird gezeigt, warum diese als Erklärung der Nichttermination eines Programms verwendet werden können. In Abschnitt 3 beschreiben wir die konkrete Implementierung. Abschnitt 4 diskutiert die Unterstützung von Prologs gesamten Sprachumfang.

2 Failure-slice

Im Kontext von Prologs Berechnungsregel, die das auf der linken Seite stehende Literal der Resolvente selektiert, terminiert eine Anfrage $\leftarrow$ *Ziel* genau dann universell, wenn die Anfrage $\leftarrow$ *Ziel, false* in endlicher Zeit scheitert. Ausgehend von dieser Anfrage könnte man versuchen, das Programm im Bezug auf diese Anfrage hin zu spezialisieren, um alle unwesentlichen Programmteile zu entfernen. Die gegenwärtigen Programmtransformationstechniken wie fold/unfold ermöglichen jedoch keine wesentliche Reduktion der Programmgröße für diese Anfrage. Statt nun ein Programmfragment zu erzeugen, das dieselben Terminationseigenschaften wie das ursprüngliche Programm aufweist, wenden wir uns kürzeren Programmfragmenten zu, die die Terminationseigenschaft des Programmes annähern.

Definition Programmpunkt. Jede Klausel besitzt an erster Stelle im Rumpf und nach jedem Ziel einen Programmpunkt. Eine Klausel $h \leftarrow g_1, ..., g_n$ mit n Zielen besitzt daher die folgenden $n+1$ Programmpunkte: $h \leftarrow p_0 g_1, p_1 ..., g_n p_n$

Definition Failure-slice Programmfragment. Eine *failure-slice* unterscheidet sich von dem dazugehörigen Programm dadurch, daß an einigen Programmpunkten das Ziel *false* eingefügt ist.

Theorem Erhaltung der Nichttermination. Für ein Programm P mit der Anfrage Q und einem dazugehörigen *failure-slice* Programmfragment S gilt: Wenn Q in S nicht terminiert, dann terminert Q nicht in P.

Die nichtterminierende Anfrage Q in S besitzt einen unendlichen Ableitungsbaum. Der Ableitungsbaum der Anfrage Q in P enthält alle Zweige von S, daher ist auch dieser Ableitungsbaum unendlich. Die Anfrage wird daher auch in P nicht terminieren.

Aus diesem Grund muß also jedes Programmfragment terminieren, damit auch das gesamte Programm terminiert kann. Ein nichtterminierendes Programmfragment darf somit als Erklärung der Nichttermination des gesamten Programms verwendet werden.

Für ein Programm mit n Programmpunkten gibt es 2^n mögliche Programmfragmente. Viele dieser Programmfragmente werden jedoch als Erklärung der Nichttermination ungeeignet sein, weil sie

entweder terminieren, oder aber eine Variante anderer, einfacherer Programmfragmente sind, die das gleiche Verhalten besitzen. Die Menge der für die Nichttermination interessanten Programmfragmente läßt sich durch statische und dynamische Techniken reduzieren, die in den folgenden Abschnitten beschrieben werden.

3 Implementierung

3.1 Statische Analyse

Zum Ausschließen terminierender Programmfragmente ließe sich jede beliebige Terminationsanalyse verwenden. Da jedoch bis zu 2^n verschiedene Programme zu untersuchen wären, beschränken wir uns auf eine wesentlich einfachere Analyse, die versucht, den großen Suchraum erst gar nicht aufkommen zu lassen. Mittels Constraints zwischen den Programmpunkten kann eine einfache Form der statischen Analyse realisert werden, die alle Programmfragmente auf einmal betrachtet. Die folgenen Kriterien stellen im wesentlichen eine Kontrollflußanalyse dar. Sie wurden so formuliert, daß sie direkt mit Constraints realisiert werden können. Die sonst übliche auf einem Fixpunktalgorithmus basierende Abstrakte Interpretation kann so umgangen werden.

Die Programmpunkte werden durch boolsche 0/1 Variablen dargestellt. Der Wert 0 bedeutet, daß an dieser Stelle das Ziel false eingefügt wird. Zusätzlich wird für jedes Prädikat eine Variable benötigt, die besagt, ob das Prädikat sicher immer terminiert. Im Anhang findet sich dazu ein komplettes Beispiel.

Je nach Propagierungsrichtung verwenden wir die folgenden Regeln. Diese Regeln werden direkt als Constraints zwischen den Programmpunkten realisiert.

Nach rechts propagierende Regeln.

R1: Scheitert in einer Klausel ein Programmpunkt p_i, so scheitert auch p_{i+1}.

R2: Wenn alle Programmpunkte vor Zielen eines Prädikats scheitern, dann scheitern die ersten Programmpunkte einer jeden Klausel des Prädikats.

R3: Wenn die letzten Programmpunkte der Klauseln eines Prädikats scheitern, so scheitern auch alle Programmpunkte nach einem Ziel des Prädikats.

R4: Die letzten Programmpunkte in endrekursiven Regeln scheitern, wenn die letzten Programmpunkte in allen anderen Regeln scheitern.

Nach links propagierende Regeln.

L1: Wenn alle ersten Programmpunkte aller Klauseln eines Prädikats scheitern, dann scheitern auch alle Programmpunkte vor Zielen des Prädikats.

L2: Wenn ein Programmpunkt nach einem terminierenden Ziel scheitert, dann scheitert auch der Programmpunkt vor dem Ziel.

L3: Wenn die Programmpunkte nach allen Zielen zu einem Prädikat außer jenen nach einer Endrekursion scheitern, dann scheitern alle letzten Programmpunkte der Klauseln des Prädikats.

Gewichtung. Da Programmfragmente zur Erklärung der Nichttermination dienen, sollen sie textlich möglichst kurz gehalten sein. Prädikate, die etwa zur Gänze scheitern, brauchen in der Erklärung nicht mehr vorzukommen. Klauseln, die zur Gänze scheitern, sind übersichtlicher darstellbar. Aus diesem Grund, werden die folgenden Werte in dieser Reihenfolge minimiert:

1. Anzahl der Prädikate, die Programmpunkte enthalten.
2. Anzahl der Klauseln mit einem Programmpunkt
3. Anzahl der erfolgreichen Programmpunkte

3.2 Dynamische Ausführung

Mit dem Analyseteil wird die Anzahl der möglicherweise nichtterminierenden Programmfragmente bereits start eingeschränkt. Zur weiteren Verfeinerung werden diese Programmfragmente für eine gewisse Zeit ausgeführt. Fragmente die innerhalb dieser Zeit terminieren, können so aussortiert werden. Da der Suchraum der Programmfragmente kleiner als der der anfänglichen Anfrage ist, terminieren sie zumeist rasch. Statt jedes einzelne Programmfragment in ausführbaren Prologcode zu übersetzen, was meist aufwendiger ist als die eigentliche Ausführung, verwenden wir ein einziges generisches Programmfragment, das bereits vor dem Analyseteil erzeugt werden kann. Dabei wird jede Klausel des ursprünglichen Programms um ein zusätzliches Argument erweitert, das einen boolschen Vektor aller Programmpunkte enthält. Bei jedem Programmpunkt wird ein Ziel arg(n,FVect,1) eingefügt. Dieses Ziel scheitert, wenn der Programmpunkt scheitern soll.

```
p(...) ←                slice_p(...,FVect) ←
                            arg(n1,FVect,1),
  q(...),                   slice_q(...,FVect),
                            arg(n2,FVect,1),
  ...,                      ...,
                            arg(ni,FVect,1),
  r(...).                   slice_r(...,FVect),
                            arg(ni+1,FVect,1).
```

3.3 Zusammenfassung

Unser Verfahren zur Erzeugung nichtterminierender Programmfragmente geht wie folgt vor.

1. Zyklen im Ausrufsgraphen werden bestimmt, um die sicher terminierenden Teile zu bestimmen.
2. Constraints für fvectPQ_weights(FVect,Weights) werden erzeugt, das die Abhängigkeiten zwischen den Programmpunkten beschreibt. Die Programmpunkte werden als Variable im Vektor FVect dargestellt. FVect ist also eine kompakte Repräsentation eines Programmfragments. Weights ist eine Liste der verwendeten Gewichtungen. (Ein konkretes Beispiel findet sich im Anhang).
3. Das ausführbare generische Programmfragment wird erzeugt und übersetzt.
4. Die folgende Anfrage wird nun ausgeführt:

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

   Prozedural betrachtet geschieht folgendes:

   (a) Durch fvectPQ_weights werden die Constraints zwischen den Programmpunkten aufgebaut.
   (b) Eine Lösung für die Gewichtung wird gesucht, beginnend mit den minimalen Werten. Daher werden Programmfragmente mit der minimalen Anzahl an Prädikaten zuerst erzeugt.
   (c) Alle Programmpunktvariablen werden mit konkreten Werten belegt.
   (d) Zuletzt wird das generische Programmfragment mit der erzeugten Belegung ausgeführt. Ziele die innerhalb einer gewissen Zeit terminieren, werden so ausgeschlossen.

   Die Kontrollflußanalyse wird also zugleich mit der Suche nach Programmfragmenten durchgeführt. Statt jedes einzelne Programmfragment für sich zu analysieren, werden alle auf einmal analysiert und erzeugt. Dadurch kann der große Suchraum beträchtlich reduziert werden.

4 Unterstützung weiterer Sprachkonstrukte

Unser Ansatz kann neben purem Prolog fast alle Sprachkonstrukte von Prolog unterstützen. Je purer diese Sprachkonstrukte sind, umso einfacher lassen sie sich in unseren Ansatz integrieren. Bei vielen dieser Konstrukte wird jedoch die Aussagekraft der generierten Programmfragmente reduziert. Gerade bei Seiteneffekten werden nicht mehr alle Programmfragmente, die zu Nichttermination führen können, angezeigt.

DCGs. Grammatiken (Definite Clause Grammars) können auf ähnliche Weise wie gewöhnliche Prologprädikate analysiert werden. Das Ziel {false} kann direkt in eine Grammatik eingefügt werden.

Constraints. Bei den meisten Constraint-Erweiterungen für Prolog dürfte es keine Integrationsprobleme geben. Falls jedoch durch die Beschaffenheit des Constraint-Lösers die Unifikation beliebiger Terme nicht terminieren könnte, so können die zuvor angeführten Analysen zur Erkennung von terminierenden Prädikaten nicht mehr verwendet werden. Man müßte davon ausgehen, daß jedes Prädikat eventuell nicht terminiert. Die meisten gegenwärtigen Implementierungen von Constraints verwenden jedoch terminierende Unifikationsalgorithmen.

Besonders nützlich haben sich failure-slices zur Analyse von CLP(FD)-Programmen erwiesen. Die typische Struktur eines CLP(FD)-Programms besteht aus zwei Teilen. Im ersten werden die Beziehungen zwischen Variablen mittels Constraints aufgebaut. Im zweiten werden konkrete Wertebelegungen mittels labeling gesucht. Verwendet man zum labeling eine der einfachen vorgegebenen Strategien, kann man davon ausgehen, daß der zweite Teil sicher immer terminieren wird. Da jedoch die Lösungssuche gelegentlich sehr aufwendig ist, läßt sich die universelle Termination des Programms nicht direkt beobachten—oft begnügt man sich daher mit dem Auffinden einer Lösung. Sehr häufig treten im ersten Teil Programmierfehler in der Form von Endlosableitungen auf. Da labeling für sich schon sehr aufwendig ist, kann man nicht direkt erkennen, ob eine sehr lange dauernde Berechnung terminiert oder nicht. Mit einem Programmfragment, das den zweiten Teil ausschließt, ist es nun viel eher möglich, diese Programmierfehler im ersten Teil zu erkennen.

Die sehr allgemein gehaltene Implementierung von CLP(FD) in SICStus-Prolog [3] erlaubt jedoch auch Constraints über unbegrenzte Wertebereiche. Dadurch führen einige Anfragen zu sehr lange dauernden Ableitungen, die jedoch terminieren, etwa $\leftarrow$ S #>0, S #> T, T #> S. Diese Anfrage terminiert erst, wenn der gesamte mögliche Wertebereich einer Zahl (der in den gegenwärtigen Implementierungen endlich ist) ausgeschöpft ist. Es ist davon auszugehen, daß derartige Anfragen fehlerhaft sind. Zur Zeit sehen wir keine effektive Möglichkeit, wie wir diese Fälle behandeln könnten. Kommt ein ähnliches Konstrukt in einem nichtrekursiven Teil des Programms vor, so wird es nicht in einem Programmfragment angezeigt werden.

Vordefinierte Prädikate. Vordefinierte Prädikate, die nur mit einer gewissen Instanzierung verwendet werden können, wie is/2 oder =../2 bedürfen keiner gesonderten Behandlung, alle Wertebelegungen kommen genauso vor, wie im ursprünglichen Programm. Durch diese Prädikate kann jedoch die allgemeinste Anfrage nicht mehr verwendet werden, um sämtliche mögliche Ursachen für Nichttermination anzuzeigen. Es können also nur mehr konkrete Anfragen betrachtet werden.

Cut. Der Cut-Operator, grenzt die Anwendbarkeit unseres Ansatzes sehr stark ein, da die Bedeutung des Cuts nur mit Hilfe der existentiellen Termination verstanden werden kann. Durch Cut kann man die Problemstellung der existentiellen Termination mittels universeller Termination ausdrücken. Ein Ziel G terminiert existentiell, wenn die Konjunktion G, ! universell terminiert. Durch das Einfügen von false kann nun die existentielle Termination eines Ziels verändert werden. Aus diesem Grund, dürfen alle Ziele und alle indirekt davon abhängenden Ziele, die durch das Cut betroffen sind, nicht verändert werden. Im einfachsten Fall, wenn die Anfrage von der Form G, ! ist, kann unsere Analyse nur das triviale mit dem Programm identische Programmfragment als Erklärung liefern. Es gibt jedoch viele Anwendungen des Cuts, die wesentlich einfacher sind, und noch immer genügend Erklärungsspielraum

zulassen. Vor allem sogenannte *shallow cuts*, die direkt nach dem Kopf bzw. nach einigen einfachen vordefinierten Prädikaten vorkommen, erlauben eine weitgehend uneingeschränkte Analyse.

Seiteneffekte. Prädikate mit Seiteneffekten behindern die Analyse besonders stark. Es dürfen nur solche Programmfragmente gebildet werden, die selbst keine Prädikate mit Seiteneffekten enthalten. Jeder Programmpunkt direkt vor einem seiteneffektbehafteten Prädikat muß scheitern. Die so erzeugten Programmfragmente können daher nur die Nichttermination der seiteneffektfreien Teile erklären.

5 Schlußfolgerungen

Wir stellten eine Slicing-Technik zur Erklärung nichtterminierender Programme vor. Durch die Verbindung von statischen und dynamischen Techniken werden sehr genaue Erklärungen generiert. Die statische Analyse wurde durch finite-domain constraints realisiert, die sich für diese Anwendung bewährt haben. Während übliche Ansätze der statischen Analyse ein einzelnes Programm betrachten, konnten wir dank Constraints eine große Menge von Programmen zugleich betrachten.

Weitere Arbeiten

Integration von Terminationsbeweisern. Aufgrund der einfachen Struktur der gegenwärtigen Implementierung erscheint es besonders einfach, diese um bessere Analyseverfahren zu erweitern, und damit irrelevante Erklärungen auszuschließen. Schließlich gehen wir gegenwärtig von einer einfachen Kontrollflußanalyse aus, die Datenabhängigkeiten außer Acht läßt. Je nach Beschaffenheit anderer Analyseverfahren bieten sich die folgenden Integrationspunkte an.

Die üblichen Terminationsanalysen verwenden Techniken der Abstrakten Interpretation, etwa Termilog [9], dessen Analyseablauf *bottom-up* erfolgt. Solche Verfahren lassen sich nicht direkt mit unserer Analyse verbinden. Eine Umformulierung auf Constraints ist nicht offensichtlich. Sie können jedoch verwendet werden, um die durch unser Verfahren gefundenen Programmfragmente auf Termination zu prüfen. Der hohe Rechenaufwand dieser Verfahren legt nahe, daß ein direktes Ausführen der Programmfragmente noch vor diesen eingesetzt werden sollte.

Ein anderer Ansatz wurde von Mesnard [10] realisiert. Hier werden die beweisbaren Terminationsbedingungen als boolsche Ausdrücke dargestellt, die in CLP(B) gelöst werden. Durch diese Darstellung könnte es möglich sein, diesen Ansatz mit unseren direkt zu verbinden, indem die Variablen über die Programmpunkte einbezogen werden. Dadurch ließe sich auch die Generierung der Programmfragmente verbessern und beschleunigen. Es würden nur mehr Programmfragmente erzeugt werden, für die es einen Terminationsbeweis nicht gibt. Die in [8] vorgestellte Erweiterung scheint unsere Vermutung zu bestätigen. Dort wird ebenfalls eine große Menge ähnlicher Programme auf Termination geprüft—die Programme unterscheiden dort in der Reihenfolge der Ziele. In zukünftigen Arbeiten werden wir uns daher mit der Integration von Mesnards Ansatz beschäftigen.

Reduktion des Lösungsraums. In dem eingangs angeführten Prädikat vorfahre_von/2 müssen letztlich alle Kombinationen der Fakten kind_von/2 auf Termination geprüft werden. Es muß hier noch immer abhängig von der Anzahl der Fakten ein Suchraum von 2^n Programmfragmenten betrachtet werden. Durch unsere Analyse kann gegenwärtig nur sichergestellt werden, daß die erste Klausel von vorfahre_von/2 nie, und die zweite Klausel immer benötigt wird. Auch die gegenwärtigen Terminationsanalysen können in diesem Programm die Termination nicht völlig automatisch beweisen. Ein neuerer Ansatz [4] könnte in der Lage sein, die fehlende Information (Bestimmung einer entsprechenden Norm) zu liefern. Techniken wie sie für Datalog entwickelt wurden, könnten hier nur für Datalogprogramme verwendet werden. Kämen jedoch allgemeine Terme in den Argumenten von kind_von/2 vor, wären diese Verfahren nicht mehr verwendbar.

Eine weitergehende Überlegung wäre die folgende durch Entfaltung entstandene Variante von vorfahre_von/2 zu betrachten. Damit diese Variante nicht terminiert, muß stets das zweite Ziel kind_von/2 erfüllt sein. Für ein konkretes Faktum muß es daher ein entsprechend unifizierbares Nachfolgerfaktum

geben. Durch diese Einschränkung ergeben sich in unserem Beispiel statt 511 möglicher Programmfragmente nur 4. Inwiefern eine derartige Technik allgemeiner anwendbar wäre, wurde noch nicht untersucht.

```
vorfahre_von(Vorfahre, Nachfahre) ←
    kind_von(Person, Vorfahre),
    kind_von(Person2, Person),
    vorfahre_von(Person2, Nachfahre), false.
← vorfahre_von(Vorfahre, Nachfahre), false.
```

Argument slicing. Alle bisherigen Slicing-Techniken für Logikprogramme [23, 7, 17] betrachten auch die Argumente von Prädikaten. Nicht benötigte Argumente werden ausgeblendet. Während dies die Lesbarkeit eines Programmfragmentes erhöhen mag, können dadurch —direkt oder indirekt— keine weiteren Klauseln oder Ziele ausgeschlossen werden. Aus diesem Grund haben wir diese Technik bisher nicht einbezogen. Es erscheint jedoch möglicherweise interessant, dies nach allen anderen Verfahren zu Verbesserung der Darstellung eines Programmfragmentes durchzuführen.

Beweis der Nichttermination. Durch unseren Ansatz ist es zwar möglich, die Ursachen für Nichttermination auf einige Programmfragmente zu lokalisieren, dennoch kann es vorkommen, daß ein Programmfragment zwar terminiert, aber einfach zuviel Rechenzeit benötigt. Die Nichttermination des Programms müßte bewiesen werden. Arbeiten zum Thema *loop detection* [5, 19, 6, 20] könnten dafür verwendet werden. Da es jedoch (genauso wie bei Termination) unmöglich ist, die gesamte Menge der nichtterminierenden Programme zu bestimmen, müssen letztlich auch jene Programmfragmente angezeigt werden, für die keine Schleife entdeckt werden konnte. Der Beweis der Nichttermination wäre dennoch nützlich, um bei einem gefundenen Programmfragment anmerken zu können, daß dieses Programm tatsächlich nicht terminiert.

Literatur

[1] J. Beck, D. Eichmann. Program and interface slicing for reverse engineering. Conf. on Software Engineering, IEEE 1993.

[2] L. Byrd. Understanding the control flow of Prolog programs. Logic Programming Workshop, Debrecen, Hungary, 1980.

[3] M. Carlsson, G. Ottosson, B. Carlson. An Open-Ended Finite Domain Constraint Solver. PLILP, 1997.

[4] St. Decorte, D. De Schreye. Demand-driven and constraint-based automatic left-termination analysis for Logic Programs. ICLP, 1997.

[5] A. van Gelder. Efficient Loop Detection in Prolog Using the Tortoise-and-hare Technique. JLP 1987(4). Siehe dazu auch [19, 6]

[6] A. van Gelder. Van Gelder's Response. JLP 14(1&2): 185 1992.

[7] T. Gyimóthy and J. Paakki. Static slicing of logic programs. AADEBUG 95.

[8] S. Hoarau, F. Mesnard. Inferring and Compiling Termination for Constraint Logic Programs, LOPSTR 1998.

[9] N. Lindenstrauss, Y. Sagiv. Automatic Termination Analysis of Logic Programs, 63-77, ICLP 1997. Siehe auch http://www.cs.huji.ac.il/~naomil/

[10] F. Mesnard. Inferring Left-terminating Classes of Queries for Constraint Logic Programs: JICSLP 1996 7–21, 1996.

[11] L. Naish. Types and the Intended Meaning of Logic. in F. Pfenning (ed.) Types in Logic Programming, 1992.

[12] U. Neumerkel. Mathematische Logik und logikorientierte Programmierung, Skriptum zur Laborübung, 1993-1997.

[13] U. Neumerkel. Teaching Prolog and CLP. Tutorial. PAP Paris, 1995 und ICLP Leuven, 1997.

[14] U. Neumerkel. GUPU: A Prolog course environment and its programming methodology. Proc. of the Poster Session at JICSLP (Fuchs, Geske Eds.), GMD-Studien Nr. 296, 1996 Bonn.

[15] J. Noyé, M. Ducassé. Logic programming environments: Dynamic program analysis and debugging (with M. Ducassé). JLP 19,20: 351-384 (1994).

[16] L. Plümer. Termination Proofs for Logic Programs, LNAI 446, 1990.

[17] St. Schoenig, M. Ducassé. A Backward Slicing Algorithm for Prolog. SAS 1996: 317-331

[18] T. Vasak, J. Potter. Characterization of Termination Logic Programs, IEEE SLP, 1986.

[19] D. Skordev. Short Note: On Van Gelder's Loop Detection Algorithm. 181-183 JLP 4(1), 1992.

[20] D. Skordev, An Abstract Approach to Some Loop Detection Problems. Fundamenta Informaticae 31(2): 195-212, 1997.

[21] M. Weiser. Programmers Use Slices When Debugging. CACM 25(7): 446-452 (1982)

[22] M. Weiser. Program Slicing. IEEE TSE 10(4): 352-357 (1984).

[23] J. Zhao, J. Cheng, K. Ushijima. Literal Dependence Net and Its Use in Concurrent Logic Programming Environment: Workshop on Parallel Logic Programming FGCS, 127-141, Tokyo, 1994.

A Failure-slices für perm/2

```
% Ursprüngliches Programm          % Minimales Programmfragment          % Zweite Lösung
perm([], []). % P1                ~~perm([], []) ← false~~.            ~~perm([], []) ← false~~.
perm(Xs, [X|Ys]) ← % P2           perm(Xs, [X|Ys]) ←                   perm(Xs, [X|Ys]) ←
   del(X, Xs, Zs), % P3              del(X, Xs, Zs), false,               del(X, Xs, Zs),
   perm(Zs, Ys). % P4                ~~perm(Zs, Ys)~~.                    perm(Zs, Ys), false.

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

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

```
← 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]. % siehe oben
% FVect = s(1,0,1,1,0,1,0,0,0), Weights = [2,2,4]. % siehe oben
% 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].

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]),
   P0 = 1, P8 = 0,
   % Sicher immer terminierend
   ImmerTermPerm ⇔ ( ¬P2 ∨ ImmerTermDel ) ∧ ( ¬P3 ∨ 0 ),
   ImmerTermDel ⇔ ( ¬P6 ∨ 0 ),
   % R1:
   ¬P2 ⇒ ¬P3, ¬P3 ⇒ ¬P4, ¬P6 ⇒ ¬P7,
   % R2:
   /*perm/2:*/ ¬P0 ∧ ¬P3 ⇒ ¬P1 ∧ ¬P2, /*del/3:*/ ¬P2 ∧ ¬P6 ⇒ ¬P5 ∧ ¬P6,
   % R3:
   /*perm/2:*/ ¬P1 ∧ ¬P4 ⇒ ¬P8 ∧ ¬P4, /*del/3:*/ ¬P5 ∧ ¬P7 ⇒ ¬P3 ∧ ¬P7,
   % R4:
   /*perm/2:*/ ¬P1 ⇒ ¬P4, /*del/3:*/ ¬P5 ⇒ ¬P7,
   % L1:
   /*perm/2:*/ ¬P1 ∧ ¬P2 ⇒ ¬P3 ∧ ¬P0, /*del/3:*/ ¬P5 ∧ ¬P6 ⇒ ¬P2 ∧ ¬P7,
   % L2:
   ¬P4 ∧ ImmerTermPerm ⇒ ¬P3, ¬P8 ∧ ImmerTermPerm ⇒ ¬P0,
   ¬P3 ∧ ImmerTermDel ⇒ ¬P2, ¬P7 ∧ ImmerTermDel ⇒ ¬P6,
   % L3:
   ¬P8 ⇒ ¬P1 ∧ ¬P4, ¬P3 ⇒ ¬P5 ∧ ¬P7,
   % Gewichtungen
   NPreds #= min(1,P1+P2) + min(1,P5+P6),
   NClauses #= P1+P2+P5+P6,
   NPoints #= P0+P1+P2+P3+P4+P5+P6+P7+P8.
```