Slicing zur Fehlersuche in (Constraint-) Logikprogrammen

Stefan Kral, Fred Mesnard, Ulrich Neumerkel
Technische Universität Wien, Université de la Réunion

Zusammenfassung

Wir stellen drei Verfahren zur Fehlersuche in Logikprogrammen vor, die durch Slicing-Techniken den für ein unerwartetes Phänomen verantwortlichen Bereich einschränken. Als Erklärung des Fehlers wird ein Programmfragment (slice) erzeugt, das modifiziert werden *muß*, um den Fehler zu beheben. Im Gegensatz zu den üblichen Techniken zur Fehlersuche (Tracing und deklaratives Debugging [6]) benötigt unser Ansatz keinerlei Interaktion mit dem Benutzer. Er ist daher besonders für Anfänger geeignet und wurde in eine Programmierumgebung integriert. Da unser Ansatz ohne den üblichen Begriff des Beweisbaumes auskommt, eignet er sich auch zur Fehlersuche von Constraint-Programmen. Insbesonders kann auch die Ursache für das Scheitern eines Zieles in Constraint-Programmen lokalisiert werden.

Beim Erlernen einer Programmiersprache nimmt die Fehlersuche einen prominenten Platz ein. Durch sie kann im Idealfall das Verständnis der Eigenschaften einer Programmiersprache vertieft werden. Bei Logik-Programmiersprachen wie Prolog sind die gegenwärtigen Techniken zur Fehlersuche jedoch kaum geeignet, das Sprachverständnis zu fördern. Die üblichen Techniken beruhen darauf, den Programmablauf in einer —gelegentlich etwas gerafften, aber letztlich immer noch prozeduralen Form— darzustellen. Derartige Ansätze eignen sich kaum, ein vertiefendes Verständnis der deklarativen Eigenschaften zu fördern.

Das von Shapiro entwickelte Algorithmic Debugging [6] versucht, die Fehlerursache unter Zuhilfenahme eines Orakels einzugrenzen. Dabei nimmt meist der Benutzer die Rolle des Orakels ein, das entscheidet, ob Ziele im Ableitungsbaum wahr sein sollen oder nicht. Die Richtigkeit der Diagnose hängt also hier von der Richtigkeit der Benutzerantworten ab. Gerade bei Anfängern ist es naheliegend, daß nicht alle Antworten für das Orakel richtig sein werden. Zur eigentlichen Fehlersuche gesellt sich so die Fehlersuche im Diagnoseprozess. Zudem sind viele der zu beantwortenden Fragen sehr komplex und schwer zu lesen. Bei Constraint-Systemen, die keine vollständigen Gleichungslöser besitzen, ist eine lesbare Darstellung ausgeschlossen. Im Gegensatz dazu benötigen unsere Verfahren keinerlei zusätzliche Benutzereingabe. Die erzeugten Erklärungen werden direkt aus dem gegebenen Programm und einer Anfrage erzeugt.

Die drei vorgestellten Verfahren basieren auf einigen (informellen) Lesarten [2], die über die allgemein übliche Trennung zwischen deklarativer und prozeduraler Sichtweise hinausgehen. Insbesonders wurde die deklarative Lesart verfeinert, um ihre analytische Erklärungskraft zu steigern. Die Grundidee beruht darauf, nur ein gewisses Fragment eines Programms auf einmal zu betrachten. Gegenwärtig werden solche Programmfragmente durch zwei zueinander komplementäre Transformationen erhalten: durch Generalisierung und Spezialisierung. Bei der Generalisierung werden Klauseln durch Wegstreichen eines Ziels verallgemeinert. Bei der Spezialisierung kommt ein neues Ziel zu einer Klausel hinzu. Ist dieses Ziel das Ziel false/0, so kann bei deklarativen Lesarten die gesamte Klausel, bei prozeduralen Lesarten, alle

Ziele hinter dem neu eingefügten gelöscht werden. Ein besonderer didaktischer Vorteil unserer Lesarten liegt darin, daß sie sich verhältnismäßig einfach erklären lassen. Das Zudecken läßt sich etwa auch von Hand an der Tafel ohne weitere Hilfsmittel veranschaulichen.

Unsere Verfahren eignen sich für pure Logik- und constraintlogische Programme. Bei der Verwendung von Negation sind unsere Verfahren nur mehr eingeschränkt einsetzbar. Prolog-Programme mit Seiteneffekten und anderen außerlogischen Konstrukten können nicht analysiert werden. Analog zu Shapiros drei Diagnosen, erklären auch wir die Fehlerfälle Unerwartetes Scheitern, Unerwartete Lösung und Nichttermination.

Unerwartetes Scheitern

Scheitert eine Anfrage unerwarteterweise, so ist das dazugehörige Programm zu speziell definiert. Scheitert nun auch eine Verallgemeinerung des Programms, muß sich bereits in dieser Verallgemeinerung ein Fehler befinden. Wir betrachten derzeit nur Verallgemeinerungen, die durch das Löschen von Zielen in Regeln entstehen, da bei komplexeren Verallgemeinerungen die textliche Entsprechung nicht mehr offensichtlich ist. Unser Verfahren versucht, minimale scheiternde Fragmente zu finden, bei denen durch jede weitere Verallgemeinerung die Anfrage erfüllt ist.

← bruder_von(B, P).	
	männlich(franz_I).
bruder_von(B, P) ←	männlich(joseph_II).
* ~~dif(B,P)~~,	männlich(leopold_II).
* ~~männlich(B)~~,	
* ~~kind_von(B,V)~~,	weiblich(maria_th).
männlich(V),	
* ~~kind_von(P,V)~~,	~~kind_von(joseph_II, maria_th).~~
* ~~kind_von(B,M)~~,	~~kind_von(joseph_II, franz_I).~~
* ~~kind_von(P,M)~~,	~~kind_von(leopold_II, maria_th).~~
weiblich(V).	~~kind_von(leopold_II, franz_I).~~

Das erzeugte Programmfragment reduziert das Programm auf einen kleinen Bereich, in dem sich zumindest ein Fehler befinden *muß*. Im angeführten Beispiel könnten neben der offenbar fehlerhaften Definition von bruder_von/2 auch die Prädikate männlich/1 und weiblich/1 —bei entsprechender Interpretation der Prädikatsnamen— zu speziell definiert sein. Auch die Anfrage selbst könnte der eigentliche Fehler sein.

In rekursiven Programmen terminieren viele durch * erhaltene Verallgemeinerungen nicht. Um solche Fragmente möglichst zu umgehen, verwenden wir eine constraintbasierte Terminationsinferenz [1], die im Gegensatz zu den üblichen Terminationsbeweisern, ganze Klassen von terminierenden Anfragen auf einmal bestimmt. Zudem versuchen wir die Anzahl der tatsächlich probeweise auszuführenden Fragmente zu minimieren.

Unerwartete Lösungen

Ist eine Anfrage, die keine Lösung finden sollte, dennoch erfüllt, muß die Programmdefinition zu allgemein sein. Das Programm wird gegenwärtig nur durch Einfügen des Ziels false/0 spezialisiert. Diese Ziele erlauben das Wegstreichen ganzer Klauseln. Detaillierte Erklärungen sind textlich nur wesentlich komplexer darstellbar.

⊬ kind_von(K, K).
~~kind_von(joseph_II, maria_theresia) ← **false**~~.
~~kind_von(joseph_II, franz_I) ← **false**~~.
kind_von(leopold_II, leopold_II).
~~kind_von(leopold_II, franz_I) ← **false**~~.

Nichttermination

Auch wenn Termination zu den prozeduralen Eigenschaften eines Logik-Programms zählt, lassen auch hier sich analog Programmfragmente erzeugt, die im Falle der existentiellen Nichttermination des Programms bereits diese Eigenschaft bedingen. Dazu werden an Programmpunkten Ziele false eingesetzt und wiederum minimale Fragmente erzeugt [5].

← vorfahre_von(Vorfahre, Nachfahre), **false**.

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

~~kind_von(joseph_II, maria_theresia) ← **false**~~.
~~kind_von(joseph_II, franz_I) ← **false**~~.
~~kind_von(leopold_II, leopold_II) ← **false**~~.
~~kind_von(leopold_II, franz_I) ← **false**~~.

Literatur

[1] F. Mesnard. Inferring Left-terminating Classes of Queries for Constraint Logic Programs: JICSLP 1996 7–21, 1996. http://www.complang.tuwien.ac.at/ulrich/cti/

[2] U. Neumerkel. Mathematische Logik und logikorientierte Programmierung, Skriptum zur Laborübung, 1993-1999.

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

[4] 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.

[5] U. Neumerkel, F. Mesnard. Localizing and explaining reasons for non-terminating logic programs with failure-slices. PPDP'99, LNCS 1702, pp. 328-341, 1999.

[6] E. Y. Shapiro. Algorithmic Program Diagnosis. POPL 1982, 299-308.