Benutzerdefinierte Constraints

M. Anton Ertl Andreas Krall

Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, 1040 Wien, Österreich
{anton,andi}@mips.complang.tuwien.ac.at

Tel.: (+43-1) 58801 {4459,4462}
Fax.: (+43-1) 505 78 38

Zusammenfassung

Um kombinatorische Suchprobleme zu lösen, werden Konsistenztechniken in Logik-Programmiersprachen verwendet. Bei vielen dieser Probleme reichen allerdings die vordefinierten Constraints nicht aus. Leider Constraints, die mit lookahead- oder forward-Deklarationen definiert werden, oft sehr langsam. In dieser Arbeit stellen wir einen neuen Ansatz für benutzerdefinierte Constraints vor. Benutzerdefinierte Constraints sind normale Prolog-Prädikate mit einer zusätzlichen Constraint-Deklaration. Sie ermöglichen eine genaue Kontrolle über Laufzeit und Suchraumbeschränkung. Damit lassen sich gewaltige Laufzeitverbesserungen gegenüber lookahead-Deklarationen erzielen.

1 Einführung

Viele Probleme wie z.B. Mittelzuteilung und Scheduling können mittels Konsistenztechniken in Logik-Programmiersprachen gelöst werden [VH89]. Dazu werden Bereichsvariablen und Constraints zu Prolog hinzugefügt. Bereichsvariable sind logische Variable, denen eine endliche Menge von Werten zugeordnet ist. Constraints sind Prädikate, die inkonsistente Werte vom Bereich der Argumente entfernen.

Nehmen wir zum Beispiel die Variablen `X` mit dem Bereich $\{1, 2, \ldots 6\}$ und `Y` mit dem Bereich $\{4, 5, \ldots 9\}$. Das Constraint `X #> Y` reduziert die Bereiche sofort auf $\{5, 6\}$ und $\{4, 5\}$. Üblicherweise wird ein Constraint wieder ausgewertet, wenn sich der Bereich der Argumente ändert. Wird in diesem Beispiel die Variable `Y` mit 5 belegt, wird das Constraint aufgeweckt und reduziert den Bereich von `X` auf den Wert 6.

Das Problem mit eingebauten Constraints ist, daß sie zwar konjunktiv, aber nicht disjunktiv zu mächtigeren Constraints kombiniert werden können. Disjunktionen haben wie in Prolog einen Wahlpunkt zur Folge. Dadurch müssen die Berechnungen nach dem Wahlpunkt mehrfach ausgeführt werden, was zu einem exponentiellen Aufwand führt. Um dieses Problem zu lösen, hat Van Hentenryck einen Mechanismus vorgeschlagen, um neue Constraints in der Logik-Programmiersprache zu definieren: forward- und lookahead-Deklarationen. Diese Prädikate testen die Vereinbarkeit von allen möglichen Kombinationen der Werte der Variablenbereiche. Diese Methode ist sehr langsam und die mit ihr definierten Constraints oft unbrauchbar, da der Aufwand für die Constraintberechnung größer ist als der, der durch die Beschränkung des Suchraumes gewonnen wird.

Wir schlagen daher in dieser Arbeit einen flexibleren und effizienteren Ansatz für benutzerdefinierte Constraints vor.

2 Benutzerdefinierte Constraints

Syntaktisch sehen benutzerdefinierte Constraints wie ein normales Prolog Prädikat aus (sie haben auch die selbe deklarative Semantik), nur haben sie zusätzlich eine Constraint-Deklaration:

`:- constraint` C=*Head* `trigger` $Goal_1$ `satisfied` $Goal_2$.

Das Prädikat, das als Constraint deklariert wird, wird durch *Head* spezifiziert. $Goal_1$ gibt an, wann dieses Constraint aufgeweckt wird. $Goal_2$ gibt an, wann dieses Constraint vollständig erfüllt ist[1]. $Goal_1$ und $Goal_2$ sind metalogische Ziele. C wird in $Goal_1$ verwendet, um das Constraint zu bezeichnen.

Wie funktioniert ein benutzerdefiniertes Constraint? Jedesmal, wenn das Constraint aufgeweckt wird, wird das Constraint-Prädikat ausgeführt und alle Lösungen des Prädikats werden gesammelt. Für jede Bereichsvariable, die in den Argumenten des Constraints vorkommt, wird die Vereinigungsmenge der Bereiche von allen Zwischenlösungen berechnet. Der Bereich der Variable wird dann auf diese Menge reduziert. Die Reduktion des Bereichs ist der einzige Effekt des Constraints. Wahlpunkte und Variablenbindungen, die während der Berechnung des Constraints angelegt werden, zeigen außerhalb keine Wirkung. Nach der Bereichsreduktion wird $Goal_2$ ausgeführt. Falls dieses Prädikat erfolgreich ist, ist das Constraint erfüllt und muß nicht mehr aufgeweckt werden. Falls dieses Prädikat fehlschlägt, ist das Constraint noch nicht vollständig erfüllt und muß später noch einmal aufgeweckt werden. Falls das Constraint-Ziel keine Lösungen hat, schlägt das Constraint fehl.

[1]Ein Constraint ist erfüllt, wenn es nicht mehr aufgeweckt werden muß

Ein benutzerdefiniertes Constraint wird nicht in der normalen Umgebung ausgeführt. Constraints, die außerhalb dieses Constraints aufgerufen wurden, werden innerhalb des Constraints nicht aufgeweckt, d.h., das Constraint arbeitet so, als ob äußere Constraints nicht existierten. Während des Sammelns der Lösungen werden unvollständig erfüllte (floundering) Constraints nicht berücksichtigt und die Bereiche werden so verwendet wie sie sind.

Die Funktionsweise wird an folgendem Beispiel gezeigt: das Constraint `max(X,Y,Z)` soll gelten, falls `Z` das Maximum von `X` und `Y` ist. Es kann folgendermaßen definiert werden:

```
:- constraint C=max(X,Y,Z)
     trigger trigger_minmax(X,C), trigger_minmax(Y,C),
             trigger_minmax(Z,C)
     satisfied one_var([X,Y,Z]).
max(X,Y,Z) :- X#>=Y, X=Z.
max(X,Y,Z) :- X#<Y,  Y=Z.
```

Wenn `max(X,Y,Z)` mit den Variablenbereichen

$$\mathtt{X} \in \{2,3,5\}, \quad \mathtt{Y} \in \{0,\ldots,4\}, \quad \mathtt{Z} \in \{2,4,6,8\}$$

aufgeweckt wird, werden folgende zwei Lösungen erzeugt:

$$\mathtt{X} = \mathtt{Z} = 2, \quad \mathtt{Y} \in \{0,\ldots,2\}$$

und

$$\mathtt{X} \in \{2,3\}, \quad \mathtt{Y} = \mathtt{Z} \in \{2,4\}$$

Die Vereinigungsmenge der Bereiche ergibt:

$$\mathtt{X} \in \{2,3\}, \quad \mathtt{Y} \in \{0,1,2,4\}, \quad \mathtt{Z} \in \{2,4\}$$

Die Wirkung dieses Constraints ist die Reduktion der Variablenbereiche auf diese Vereinigungsmenge.

Die Constrainterfüllung wird mit `one_var(T)` überprüft, das erfüllt ist, falls `T` höchstens eine freie Variable enthält, in diesem Beispiel also nicht. Da es häufig verwendet wird, ist es vordefiniert.

Die trigger-Prädikate funktionieren ähnlich wie `freeze/2` von Prolog II. Es ist immer erfolgreich und hat die Wirkung, daß das Constraint aufgeweckt wird, wenn sich der Variablenbereich in einer bestimmten Weise ändert.

`trigger_minmax(V,C)` bedeutet, daß das Constraint `C` aufgeweckt wird, falls sich das Minimum oder das Maximum des Bereichs von `V` ändert. Es gibt vordefinierte trigger-Prädikate, mit denen man andere trigger-Prädikate definieren kann.

Die deklarative Semantik eines Prädikats wird durch die Constraint-Deklaration nicht verändert.

C	ben.-def.	forward	Auswahl
8.5ms	9.3ms	14.5ms	30.3ms

Tabelle 1: Five houses timings

```
?- [A,B,C] in 1..N, max(A,B,C).
```

domain size `N`	ben.-def.	lookahead
20	0.87ms	2360ms
50	1.12ms	36500ms
100	1.55ms	292000ms

Tabelle 2: `max/3` timings (one activation)

forward- und lookahead-Deklarationen können mit userem Mechanismus emuliert werden.

Ähnliche Arbeiten sind [VHD91, Frü92, SH92, LPW92, Neu90, CMG82, Nai86, Sid93].

3 Ergebnisse

Wir haben benutzerdefinierte Constraints implementiert. Dazu haben wir das Aristo-System, ein WAM basiertes Logik-Programmiersystem mit Constraints, erweitert. Die Benchmark-Programme wurden auf einer DecStation 5000/125 (25MHz R3000) ausgeführt.

Ein Benchmark ist das Fünf-Häuser-Rätsel [VH89]. Wir verglichen vier Ansätze: ein in der Programmiersprache C implementiertes Constraint, unsere benutzerdefinierten Constraints, ein mit forward-Deklaration definiertes Constraint und den normalen Auswahlmechnismus von Prolog. Die Lösung benötigt ein Constraint `plusorminus/3`, für das wir die vier verschiedenen Implementierungsvarianten testeten. Das Problem ist sehr einfach, daher sind die Unterschiede nicht sehr groß. Die Zeiten, um eine Lösung zu finden, sind in der Tabelle 1 enthalten.

Wir verglichen auch die Zeiten eines Aufrufes von `max/3` mit einer lookahead Version von `max/3` (siehe Tabelle 2). Bei beide Versionen wird der Suchraum in gleicher Weise beschnitten. Wie erwartet ist die lookahead-Version um einen quadratischen Faktor langsamer. Die Version mit benutzerdefinierten Constraints ist zwischen 2700 und 188000 Mal schneller. Bereichgrößen wie in diesem Beispiel sind typisch für `max/3`. Der Sinn dieses Beispiels ist nicht, den Effizienzgewinn zu zeigen, sondern zu demonstrieren, daß lookahead-Deklarationen in der Praxis oft viel zu langsam sind, während benutzerdefinierte Constraints noch erfolgreich verwendet werden können.

4 Zusammenfassung

Wir haben einen Mechanismus für benutzerdefinierte Constraints vorgestellt. Er setzt sich aus Mechanismen zum Aufrufen von Constraints und zum Zusammenfassen der Lösungen eines Constraint-Prädikats zusammen. Syntaktisch gesehen sind benutzerdefinierte Constraints normale Prolog-Prädikate mit einer zusätzlichen Constraint-Deklaration.

Benutzerdefinierte Constraints ermöglichen eine genaue Kontrolle über Suchraumbeschrankung und Laufzeit. Die deklarative Bedeutung des Programms ist dieselbe wie ohne Constraint-Deklaration. Mit unserem Mechanismus können forward- und lookahead-Deklarationen emuliert werden. Er ist einfach zu implementieren.

Literatur

[CMG82] K. L. Clark, F. G. McCabe, and S. Gregory. IC-Prolog language features. In K. L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 253–266. Academic Press, London, 1982.

[Frü92] Thom Frühwirth. Constraint simplification rules. Technical Report ECRC-92-18?, ECRC, 1992.

[LPW92] Thierry Le Provost and Mark Wallace. Domain independent propagation. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 1004–1011, ICOT, Japan, 1992. Association for Computing Machinery.

[Nai86] Lee Naish. *Negation and Control in Prolog.* Springer LNCS 238, 1986.

[Neu90] Ulrich Neumerkel. Extensible unification by metastructures. In *Meta-90*, Leuven, 1990.

[SH92] Gregory Sidebottom and William S. Havens. Hierarchical arc consistency for disjoint real intervals in constraint logic programming. *Computational Intelligence*, 8(4):601–623, 1992.

[Sid93] Greg Sidebottom. Compiling constraint logic programming using interval computations and branching constructs. Technical report, Simon Fraser University, 1993.

[VH89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series. MIT Press, Cambridge, Massachusetts, 1989.

[VHD91] Pascal Van Hentenryck and Yves Deville. The cardinality operator: A new logical connective for constraint logic programming. In *Eighth International Conference on Logic Programming (ICLP-8)*, pages 745–759. MIT Press, 1991.