Lambdas und Schleifen in monotonen Logikprogrammen

Ulrich Neumerkel
Technische Universität Wien
Institut für Computersprachen
ulrich@mips.complang.tuwien.ac.at

Zusammenfassung

Lambda-Abstraktionen und Schleifenkonstrukte sind in der Logikprogrammierung bisher kaum aufgenommen worden. Diese Zurückhaltung hängt unmittelbar mit den algebraischen Eigenschaften der verwendeten Konstrukte zusammen. Die bisher vorgeschlagenen Ansätze verletzen grundlegende Eigenschaften wie die der Monotonie, wodurch nicht nur eine deklarative Betrachtungsweise verunmöglicht wird und diagnostische Verfahren stark eingeschränkt werden, sondern auch effiziente Implementierungen behindert werden. Mit der vorgestellten Umsetzung von Lambda-Ausdrücken werden die meisten Mängel unmittelbar vermieden. Es eröffnen sich nun neue Möglichkeiten monotone Schleifenkonstrukte einzubinden.

1 Einführung

Das Programmieren höherer Ordnung in purem, monotonen Prolog stellt seit jeher ein wenig beachtetes Gebiet dar. Es ist zwar prinzipiell auch in Prolog möglich, ähnlich der Funktionalen Programmierung vorzugehen, dennoch fanden diese Konstrukte bisher keinen Anklang. Mit ein Grund liegt darin, dass selbst Lambda-Abstraktionen nicht angenommen wurden. Man hat geradezu den Eindruck, dass das Lambda in Prolog *littera non grata* ist. Sehr ähnlich verhält es sich auch mit Schleifenkonstrukten. Auch hier zieht man es in Prolog vor, selbst einfache Schleifen als direkt rekursive Prädikate anzuschreiben, statt sich einer kompakteren Schreibweise zu bedienen. Die damit verbundenen Nachteile, wie etwa die Verwendung von sonst nicht weiter benötigten Namen nimmt man billigend in Kauf. Die bisherigen Versuche, endlich robuste Lambda-Abstraktionen und Schleifenkonstrukte zu entwickeln sind letztlich fehlgeschlagen, weil die Monotonieeigenschaften unnotwendigerweise verletzt werden und die Notwendigkeit eines komplexeren Übersetzungsvorgangs erforderlich ist. Rekursive monotone Programme werden durch diese Schleifenkonstrukte in Programme umgewandelt, die gelegentlich die Monotonieeigenschaft verletzen. Gerade in der Constraintprogrammierung ist dies ein nicht zu vernachlässigendes Risiko. Dadurch werden etwa die Ergebnisse diagnostischer Verfahren verfälscht. Bevor wir diese neuen Lambda-Ausdrücke vorstellen, betrachten wir eine Form der Monotonieeigenschaft genauer.

2 Verallgemeinerungen

Anhand eines Beispiels der klassischen Programmierung höherer Ordnung wie sie schon seit langem in Prolog bekannt ist [1] verbunden mit CLPFD-Constraints [8] suchen wir Verallgemeinerungen eines monotonen Programms. Die Beispiele sind direkt in SWI-Prolog ausführbar.

```
maplist(_Pred, []).
maplist(Pred, [E|Es]) :-
   call(Pred, E),
   maplist(Pred, Es).

?- maplist(#\=(1), [2,1,3]).  % Entspricht 1 #\= 2, 1 #\= 1, 1 #\= 3
false.
```

In diesem Programm scheitert das Ziel — angezeigt durch `false`. Um zu verstehen warum, ist es naheliegend dazugehörige Verallgemeinerungen zu betrachten. So scheitert bereits ein Ziel, bei dem `2` und `3` durch Variable ersetzt wurden.

```
?- maplist(#\=(1), [X,1,Y]).  % Entspricht 1 #\= X, 1 #\= 1, 1 #\= Y
false.
```

Wie weit wir mit derartigen Verallgemeinerung gehen dürfen, ohne mit Nichttermination in Berührung zu kommen, lässt sich oft durch Terminationsanalysen [6] bestimmen. Üblicherweise wird hier laut Terminationsinferenz die Länge der Liste endlich bleiben müssen. In diesem konkreten Fall könnten wir aber noch weiter gehen, womit wir allerdings nun möglicherweise Nichttermination in Kauf nehmen müssen.

```
?- maplist(#\=(X), [_,X|_]).
false.
```

Eine weitere Verallgemeinerung der übriggebliebenen Liste führt nun zu konkreten Lösungen. Den Term `#\=(X)` weiterzuverallgemeinern führt lediglich zu einem Instanzierungsfehler. Wir haben offenbar die maximal erreichbare Verallgemeinerung erreicht. Offenbar liegt die Ursache für das Scheitern der Anfrage in dem Umstand, dass ein Wert in der Liste ungleich zu sich selbst sein soll. Diese Technik kann als eine Adaptierung von Slicing-Techniken [9] gesehen werden.

Derartige Verallgemeinerungen sind nur in monotonen Logikprogrammen sinnvoll. Sobald wir uns auf die monotone Eigenschaft des Programms nicht mehr verlassen können, können derartige Verallgemeinerungen keine Aussagen mehr über das Programm tätigen. Es wäre nun wünschenswert, dass diese Monotonieeigenschaft auch für weitere Sprachkonstrukte gilt. Insbesondere für Lambda-Abstraktionen und den damit verbundenen Schleifenkonstrukten.

Traditionelle Lambda-Abstraktionen, wie sie aus der Funktionalen Programmierung bekannt sind, eignen sich nicht direkt zu einer Darstellung in Prolog. Um unterscheiden zu können, ob eine Variable frei oder gebunden ist, bedarf es einer Analyse des Kontexts. Erst durch diese Analyse kann eine Entscheidung getroffen werden. Lambda-Abstraktionen konnten so bislang nicht einfach in einer Bibliothek zur Verfügung gestellt werden. Je nach Realisierung dieser Analyse kann das Ergebnis einer derartigen Analyse anders aussehen. Dies lässt sich durch die Einbettung eines Ziels nachvollziehen. So wird etwa ein `Ziel` durch `NV = Ziel, NV` ersetzt, und damit die Analyse zu einem späteren Zeitpunkt erst ausgeführt.

Unsere neue Realisierung von Lambda-Abstraktionen basiert ausschließlich auf Prädikatsdefinitonen. Es ist keinerlei Veränderung des Übersetzers erforderlich. Eine einfache Bibliothek ist ausreichend um in ISO Prolog verwendet werden zu können [5].

3 Lambdas in ISO Prolog

Lambda-Ausdrücke bestehen aus zwei Komponenten. Ein Teil dient der Umbenennung, der andere der Parameterübergabe.

3.1 Umbenennung

In einem parameterlosen Lambda-Ausdruck `\` werden nun sämtliche Variable als lokale Variable betrachtet. Bindungen von ihnen sind also nicht außen sichtbar.

```
?- X+Y = 1+2.
X = 1,
Y = 2.
```

```
?- \ ( X+Y = 1+2 ).
true.
```

Im Unterschied zum herkömmlichen Ansatz, dem Übersetzer die Einteilung der Variable in freie und gebundene zu überlassen, muss man nun selbst die Variablen einzeln deklarieren. Dazu wird `+\` verwendet. Werden sämtliche Variable deklariert, so erübrigt sich der Lambda-Ausdruck. Variable die undeklariert sowohl in einem Lambda-Ausdruck als auch außerhalb vorkommen, sollten idealerweise als Fehler gemeldet werden. Bei einfacher Interpretation führen sie zu unbeabsichtigten Bindungen.

```
?- X+\ ( X+Y = 1+2 ).
X = 1.
```

```
?- [X,Y]+\ ( X+Y = 1+2 ).
X = 1.
Y = 2
```

3.2 Parameter

Neben der Umbenennung von Variablen können nun Parameter explizit über `call/2..` übergeben werden. Für sich genommen ruft `call(Cont, Arg)` einen Term (Kontinuation) `Cont` mit noch einem weiteren Argument auf. Dieses vordefinierte Prädikat geht auf O'Keefe zurück [2]. Auf dieselbe Art und Weise können nun Parameter verwendet werden. Allerdings fehlt hier die Umbenennung. Man kann also noch die Bindung der Variable `Y` erkennen.

```
?- call(#=<(3),X).
X in 3..sup.

?- 3 #=< X.
X in 3..sup.
```

```
?- call(Y^(Y#>=3),X).
Y = X,
X in 3..sup.
```

Erst durch das gemeinsame Verwenden von `\` und `^` entstehen nun die eigentlichen Lambdas. Diese können nun auch mit globalen Variablen verwendet werden. Die `^`-Schreibweise geht auf Pereira zurück [3], der sie allerdings nicht zu einem vollständigen Lambdakonstrukt ausgebaut hat.

```
?- call(\Y^(Y#>=3),X).
X in 3..sup.

?- maplist(\Y^(Y#>=3),[X1,X2]).
X1 in 3..sup,
X2 in 3..sup.

?- maplist(Z+\Y^(Y#>=Z),[X1,X2]).
X2#>=Z,
X1#>=Z.
```

4 Realisierung

Zur einfachen Umbenennung verwenden wir lediglich das vordefinierte Prädikat `copy_term/2`. Umbenennungen können aber nicht nur bei einem einfachen Ziel (erste Zeile) vorkommen, sondern auch bei Konstrukten, die ein oder mehrere weitere Argumente benötigen — sogenannten, *continuations*. Die eigentlichen Parameter (`V1`, ...) werden unbehelligt weitergereicht.

```
\(FC) :- copy_term(FC,C),call(C).
\(FC, V1) :- copy_term(FC,C), call(C, V1).
\(FC, V1, V2) :- copy_term(FC,C), call(C, V1, V2).
    ...
```

Für Umbenennungen in Gegenwart globaler Variable benötigen wir eine Operatordeklaration. Die globalen Variablen `GV` werden nun nicht mehr kopiert, sondern nur mehr die restlichen Variablen in `FC`. Prozedural gesehen werden die `GV` kopiert und gleich wieder mit sich gleichgesetzt. Auch hier bleiben die Parameter unverändert.

```
:- op(201,xfx,+\).

+\(GV, FC) :- copy_term(GV+FC,GC+C), call(C).
+\(GV, FC, V1) :- copy_term(GV+FC,GV+C), call(C, V1).
+\(GV, FC, V1, V2) :- copy_term(GV+FC,GV+C), call(C, V1, V2).
     ...
```

Letztendlich fehlt noch die Parameterübergabe. Hier wird jeweils das erste Argument mit dem ersten Argument der Parameter verbunden. Diese Parameter stellen in gewisser Weise eine auscompilierten Stapel dar.

```
^(V1, Goal, V1) :- call(Goal).
^(V1, Goal, V1, V2) :- call(Goal, V2).
^(V1, Goal, V1, V2, V3) :- call(Goal, V2, V3).
    ...
```

5 Schluss

Wir haben einen neuen besonders einfachen Ansatz von Lambda-Ausdrücken für Prolog vorgestellt, der lediglich durch direkte Prädikatsdefinitionen darstellbar ist und dennoch Variablenquantifikationen so umsetzt, dass es zu keinen Monotonieverletzungen kommt. Alle verwendeten Mittel sind zwar schon seit langem bekannt, dennoch ist die konkrete Zusammenstellung bisher noch nicht so vorgekommen. Die vollständige Implementierung der Bibliothek `lambda`, die auch Constraints (eine implementierungsspezifische Erweiterung der Norm) kann sowohl in SWI-Prolog als auch YAP direkt verwendet werden. Sie befindet sich unter http://www.complang.tuwien.ac.at/ulrich/Prolog-inedit/lambda.pl

Literatur

[1] D. H. D. Warren. Higher-Order Extensions to Prolog - Are They Needed?, In Hayes, Michie and Pao, Machine Intelligence 10. Ellis Horwood, 1982. Originally: International Machine Intelligence Workshop, Cleveland, April 1981, DAI Research Paper 154.

[2] R. O'Keefe. The Craft of Prolog. MIT-Press. 1990.

[3] F. C. N. Pereira, St. M. Shieber Prolog and Natural-Language Analysis. (CSLI Lecture Notes 10, ISBN 0-937073-18-0), 1987. Digital version

[4] D. Cabeza, M. Hermenegildo, and J. Lipton Hiord: A Type-Free Higher-Order Logic Programming Language with Predicate Abstraction ASIAN 2004, LNCS 3321, pp. 93-198, 2004. PDF

[5] ISO/IEC 13211-1 Programming languages - Prolog - Part 1: General core. 1995.

[6] F. Mesnard, U. Neumerkel. Applying static analysis techniques for inferring termination conditions of logic programs. 8th Static Analysis Symposium (SAS'01), Paris 2001.

[7] U. Neumerkel, St. Kral. Declarative program development in Prolog with GUPU. 12th Workshop on Logic Programming Environments (WLPE), Copenhagen 2002.

[8] M. Triska, U. Neumerkel, J. Wielemaker. Better Termination for Prolog with Constraints. WLPE, Udine 2008.

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