Programmiersprachen

Logik-Programmierung

Berthold Hoffmann

Studiengang Informatik
Universität Bremen

Wintersemester 2004/2005
(Vorlesung am 7. Februar 2005)

Logik-Programmierung (7. Februar 2005)

1 SLD-Resolution
2 Logik-Programmierung
3 Prolog
4 Erweiterungen von PROLOG

Interpretationen von Hornklauseln

- Zielklauseln behaupten Nicht-Existenz
$$\begin{aligned} & \{\neg G_1, \ldots, \neg G_n\} \\ \Leftrightarrow \quad & \forall x_1 \cdots x_n \neg G_1 \vee \cdots \vee \neg G_n \\ \Leftrightarrow \quad & \forall x_1 \cdots x_n \neg (G_1 \wedge \cdots \wedge G_n) \\ \Leftrightarrow \quad & \neg (\exists x_1 \cdots x_n (G_1 \wedge \cdots \wedge G_n)) \end{aligned}$$
- Programmklauseln sind Schlussregeln.
$$\begin{aligned} & \{P, \neg Q_1, \ldots, \neg Q_n\} \\ \Leftrightarrow \quad & \forall x_1 \cdots x_n (P \vee \neg Q_1 \vee \cdots \vee \neg Q_n) \\ \Leftrightarrow \quad & \forall x_1 \cdots x_n (P \vee \neg (Q_1 \wedge \cdots \wedge Q_n)) \\ \Leftrightarrow \quad & \forall x_1 \cdots x_n (P \leftarrow (Q_1 \wedge \cdots \wedge Q_n)) \end{aligned}$$
- Einelementige Programmklauseln definieren Fakten.
$\{P\} \Leftrightarrow \forall x_1 \cdots x_n P$

Breitensuche *versus* Tiefensuche

- Breitensuche:
  - Resolviere alle Literale der Zielklausel, bevor ein Resolventen-Literal weiter resolviert wird.
- Tiefensuche:
  - Resolviere alle Resolventenliterale eines Literals der Zielklausel (und deren Resolventen), bevor ein weiteres Literal der Zielklausel untersucht wird.

Satz

- *Breitensuche ist vollständig für Hornklauseln.*
- *Tiefensuche ist nicht vollständig für Hornklauseln.*

SLD-Resolution

"Geordnete" lineare Resolution mit Tiefensuche

- Klauseln sind Listen
- Programme sind Klausel-Listen
(geordnet, mit Doubletten)

Definition (SLD-Resolution)

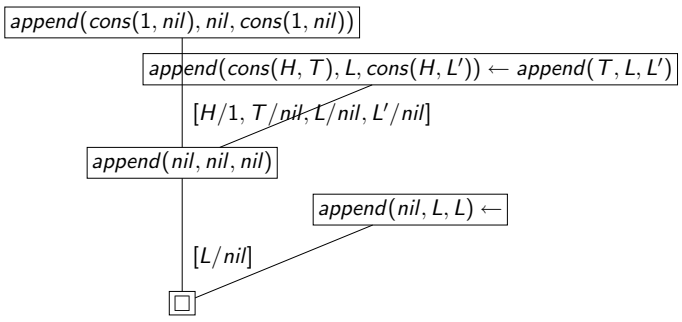
- *Eingabe*: Eine Zielklausel und ein Programm
- Jeder Schritt resolviert die erste Zielklausel mit der ersten passenden Programmklausel
- Passt keine Programmklausel, geschieht *backtracking*:
  - Resolutionsschritte werden rückgängig gemacht.
  - Die nächsten Programmklauseln werden resolviert.
  - Solange bis alle Möglichkeiten erschöpft sind.

Ein Beispielprogramm

Beispiel (Prädikat zur Listenverkettung)

- Faktum
$$append(nil, L, L) \leftarrow$$
- Programmklausel (*definite clause*)
$$append(cons(H, T), L, cons(H, L')) \leftarrow append(T, L, L')$$
- Anfrage (*goal*)
$$\neg append(cons(1, nil), nil, cons(1, \ nil))$$

SLD-Resolution am Beispiel



Weitere Anfragen und Antworten

- $append(cons(1, nil), cons(2, X), Y)$
yes. $Y = cons(1, cons(2, X))$
- $append(X, Y, cons(1, nil))$ yes.
$X = nil, Y = cons(1, nil)$;
yes. $X = cons(1, nil), Y = nil$;
no.
- $append(X, nil, X)$
yes. $X = nil$;
yes. $X = cons(H, nil)$;
...

Logik-Programmierung (7. Februar 2005)

1. SLD-Resolution
2. Logik-Programmierung
   - Prädikate
   - Variablen
   - Programmieren
3. Prolog
4. Erweiterungen von PROLOG

Kern des logischen Programmierens

- Prädikate definieren Relationen (die Datenbasis).
- Anfragen haben keine, eine, mehrere oder unendlich viele Antworten.
- sie werden nacheinander aufgezählt (durch *backtracking*).
- Anfragen erlauben vielseitige Verwendung von Prädikaten.

Beispiel (Benutzung von *append*)

- $append(\ell_1, \ell_2, \ell_3)$ prüft, ob $\ell_3 = \ell_1 +\!\!+ \ell_2$ ist (Test).
- $append(\ell_1, \ell_2, R)$ berechnet $R = \ell_1 +\!\!+ \ell_2$ (funktional).
- $append(P, S, \ell_3)$ berechnet alle Aufspaltungen von ℓ_3.
- $append(P, S, C)$ zählt alle $(\ell_1, \ell_2, \ell_3) \in R_{append}$ auf.

Terme

- Werte sind Terme über Atomen und Funktoren.
- Terme können nicht instanziierte Variablen enthalten ("Unbekannte").
- Dies gilt auch für Eingabe und Ausgabe.

Beispiel (Weitere Benutzungen von *append*)

- $append(\ell, X, Y)$ berechnet aus den "Eingaben" ℓ und X die (einzige) Ergebnisliste $Y = \ell +\!\!+ X$.
- Vorsicht: $append(X, \ell, Y)$ berechnet nicht $Y = X +\!\!+ \ell$. Weshalb?

Funktionen

- Alle Funktoren in Termen sind frei (wie im Herbrand-Universum).
- Funktoren sind Konstruktoren (wie in HASKELL)
- Andere Funktionen werden als Prädikate definiert.
  - *append* definiert die Listenverkettung "++".
  - Aufrufe von *append* können nicht geschachtelt werden (wie die von "++").

Wahrheitswerte

- Der Typ $\mathcal{W} = \{false, true\}$ wird nicht direkt dargestellt.
- Boole'sche Operationen werden als Prädikate realisiert
  - Gelingen (*success*) bedeutet "*true*"
  - Fehlschlag (*failure* bedeutet "*false*"
- Prinzip der *negation by failure*

$$p(\cdots) \leftarrow \cdots \wedge \neg ordered(L) \wedge \cdots$$

  Wenn $ordered(L)$ fehlschlägt, gelingt $\neg ordered(L)$.
- Annahme der abgeschlossenen Welt (*closed world assumption*)
  - Was nicht ableitbar ist, gilt nicht.
  - Regeln müssen die Welt vollständig modellieren.

Logische Variablen

- Zustände: gar nicht, teilweise oder vollständig instanziiert.
- Einmalige "Zuweisung" durch Unifikation.
- Seiteneffekt auf alle teilweise instanziiertenVariablen, in denen die Variable vorkommt.
- "Freimachen" und Überschreiben nur durch *backtracking*.

Beispiel (Weitere Benutzungen von *append*)

- $append(\ell, X, Y)$ berechnet aus den "Eingaben" ℓ und X die (einzige) Ergebnisliste $Y = \ell +\!\!+ X$.
- Vorsicht: $append(X, \ell, Y)$ berechnet nicht $Y = X +\!\!+ \ell$. Weshalb?

guess & verify

- *guess*: Generiere eine Menge von möglichen Antworten
- *verify*: Wähle die relevante(n) Antwort(en) aus

$$compute(X) \leftarrow guess(X) \wedge verify(X)$$

Beispiel (Ultimatives Sortieren)

$$\begin{aligned}
sort(L, L') &\leftarrow permute(L, L') \wedge ordered(L') \\
permute([], []) &\leftarrow \\
permute([H|T], T2) &\leftarrow permute(T, T1), nsert(H, T1, T2) \\
nsert(X, L, L1) &\leftarrow app(P, S, L), app(P, [X|S], L1) \\
ordered([]) &\leftarrow \\
ordered([X]) &\leftarrow \\
ordered([F, S|T]) &\leftarrow F \leq S, ordered([S|T])
\end{aligned}$$

Logik-Programmierung (7. Februar 2005)

1. SLD-Resolution
2. Logik-Programmierung
3. Prolog
   - Geschichte
   - Werte
   - Zahlen
   - Listen
   - Programmieren
4. Erweiterungen von PROLOG

Entstehung

Entwerfer Alain Colmerauer und P. Roussel, Marseille 1972

Motivation
- Verarbeitung natürlicher Sprache
- Definition von Programmiersprachen (ALGOL-68)

Einsatzfelder
- künstliche Intelligenz
- insbesondere Wissensrepräsentation

Motto *algorithm* = *logic* + *control* (Kowalski)

Werte

- Werte sind Terme (Bäume) über Atomen und Variablen
- Atome sind
  - Zahlenliterale `0`, `1972`, `3.1415`
  - Bezeichner `lisp`, `hugo`,
  - Zeichenketten `'Hello World!'`, `'====='`, `'='`, `'+'`
- Variablen sind Bezeichner (groß geschrieben) `List`, `_`
- Terme sind Atome, denen eine geklammerte Liste von termen folgen kann
  - `'Hello World!'`
  - `tree(X, empty)`
  - `'+'(7,X)`
- Einige Atome sind Operatoren und können infix geschrieben werden
  - `7+X` ≡ `'+'(7,X)`

Typen – Fehlanzeige

- Werte sind generell ungetypt.
- Prädikate und Funktoren haben eine Stelligkeit

  $append/3 \quad permute/2 \quad sort/2 \quad ordered/1$

- Namen können mit verschiedenen Stelligkeiten benutzt werden

  $append/1 \quad append/2 \quad append/3$

- Das erste Auftreten eines Namens vereinbart in als Variable bzw. Atom
  - Vorsicht bei Tippfehlern!

    $append(nill, L, L1) \leftarrow$

    Diese Regel ist wohlgeformt!
  - Nur das Fehlen von Regeln für Prädikate wird bemerkt.

Arithmetik

- vordefinierte Operationen auf Zahlen: "+", "−", "*", "/", "mod"
  Aufrufe können geschachtelt werden (`3+(X*4)`)
- vordefinierte Vergleichsprädikate: "=", "\=", "<", ">", ">=", "=<"
- Ihre Argumente müssen Zahlen sein (also vollständig instanziiert)
- X **is** $\langle expr \rangle$ realisiert eine Zuweisung
  - X ist freie Variable
  - $\langle expr \rangle$ ist voll instanziierter Ausdruck

Arithmetik am Beispiel

Beispiel (Rechnen mit Zahlen)

- X **is** $13 + 13 \rightsquigarrow yes, X = 26$
- X **is** $26, X$ **is** $13 + 13 \rightsquigarrow yes$
- $13 < X, X$ **is** $13 + 13 \rightsquigarrow yes$
- $N > 0, N < (N + N) \rightsquigarrow no$
- $N < (N + N), N$ **is** $13 \rightsquigarrow no$
- N **is** $13, N < (N + N) \rightsquigarrow yes$

Listen

- Listen sind heterogen (weil ungetypt)
- vordefinierte Atome
  - die leere Liste "`[]`"
  - der Konstruktor *cons* heißt "`.`" oder "`|`"
  - Schreibweise "`[a, b, c]`"
    für "`'.'(a, '.'(b, '.'(c, [])))`"

Beispiel (Einige vordefinierte Prädikate auf Listen)

$$\begin{aligned} last(X, [X]) &\leftarrow \\ last(X, [H, Y]) &\leftarrow last(X, Y) \\ nextto(X, Y, [X|Y|R]) &\leftarrow \\ nextto(X, Y, [H|Z]) &\leftarrow nextto(X, Y, Z) \end{aligned}$$

Eingebaute Prädikate

Gleichheit ("==", "\==")

`X==X` ⇝ yes
`X==Y` ⇝ no
`[]==[]` ⇝ yes
`[1]==[X]` no
`[X]==[X]` yes

Unifikation("=", "\=")

`X=Y` ⇝ yes, X=Y
`[1]=[X]` yes, X=1
`[X]=[X]` yes
`[] =[X]` no

Selbst-modifizierende Programme

- Programme sind auch nur Terme
  - Die Zeichen "`:-`", "`;`", "`,`", "`.`" sind Atome (infix geschrieben)
  - Aus ihnen sind Programme aufgebaut
  - Prädikate können Programe aufbauen
- Vordefinierte Prädikate zur Manipulation der Datenbasis
  - `asserta` und `assertz` fügen Klauseln hinzu (vorne bzw. hinten).
  - `retract` entfernt eine Programmklausel.
- Programme können also lernen (und vergessen)

Programmieren mit Seiteneffekten

Beispiel (Warteschlangen)

```
setup (q(X,X)).
enter(A, q(X, Y), q(X, Z)) :- Y = [A|Z].
leave(A, q(X, Z), q(Y, Z)) :- X = [A|Y].
```

Anfragen

```
?- setup(Q), enter(A,Q,Q1), enter(B,Q1,Q2),
   leave(X,Q2,Q3), leave(Y,Q3,Q4).
```

Q = q([A, B—X], [A, B—X])
Q1 = q([A, B—X], [B—X])
Q2 = q([A, B—X], X) X = A
Q3 = q([B—X], X) Y = B
Q4 = q(X, X)

Differenzlisten

Beispiel

Differenzlisten mit Inhalt [a,b]

```
dl([a,b],[])   dl([a,b|E],E)   dl([a,b,c|F],[c|F]).
```

Effiziente Verkettung von Differenzlisten

```
append_dl(X, Y, Z) :- X = dl(L,M), Y = dl(M,N), Z = dl(
```

Inhalt einer Differenzliste

```
contents(X, dl(L,E)) :- append(X, L, E). (*)
```

Anfragen

```
?- contents([a,b], dl([a,b,c],[c]))
?- contents([a,b], dl([a,b,c|F],[c|F]))
```


Beschneiden des Suchbaums (*cut*)

- Der Operator "!" kann auf rechte Regelseiten benutzt werden
- Für Literale links von "!" wird kein *backtracking* gemacht.
- Für die linke Seite, die "!" einführte, auch nicht.
- *green cut*: Abschneiden von erfolglosen Suchästen
- *red cut*: Abschneiden von erfolgreichen Suchästen

Beispiel

$$sort(L, L') \leftarrow permute(L, L') \wedge ordered(L')!$$

liefert höchstens die erste geordnete Permutation einer Liste.

Logik-Programmierung (7. Februar 2005)

1. SLD-Resolution
2. Logik-Programmierung
3. Prolog
4. Erweiterungen von PROLOG

Typisierung und Kapselung

Typen

- Terme werden polymorph getypt (wie in HASKELL)
- Funktions- und Prädikatsymbole haben Signaturen
- Manche Sprachen führen auch Untertypen ein

Moduln

- Moduln
- Schnittstellen
- getrennte Übersetzung
- veträglich mit `asserta` usw.?

Nichtdeterminismus und Datenfluss

Nichtdeterminismus

- Nichtdeterministische Prädikate haben $n \geqslant 1$ Lösungen
- Partielle Prädikate können fehlschlagen
- Deterministische Prädikate definieren Funktionen (Sie können wie in funktionalen Sprachen durch Gleichungen definiert werden.

Datenfluss

- Festlegen der Flussrichtung
  - Eingabe-Parameter müssen instanziiert sein
  - Ausgabe-Parameter dürfen nicht instanziiert sein
- Dann können Prädikate effizienter implementiert werden (sind aber nicht so flexibel benutzbar)

Constraints und Funktionale

Intervall-Arithmetik (*constraints*)

- unvollständig instanziierte Variablen können verglichen werden
- Rechnen mit eingeschränkten Wertebereichen

Funkionale (*constraints*)

- Prädikate können mit Funktionen parametrisiet werden.
- Variablen können für Funktionen oder Prädikate stehen.
- Unifikation ist im Allgemeinen nicht mehr entscheidbar.
- Im Allgemeinen gibt es keinen (allgemeinsten) Unifikator mehr

Moderne logische Sprachen

- CURRY – eine funktional-logische Sprache
  Kiel – Aachen – Münster – Portland
  CURRY = HASKELL + Unifikation + Nichtdeterminismus + *constraints*
- MERCURY – eine funktional-logische Sprache
  Melbourne
  MERCURY = PROLOG + Typen + Determinismus

(Mehr dazu auf den Webseiten)

Logik-Programmierung (7. Februar 2005)

1 SLD-Resolution
2 Logik-Programmierung
3 Prolog
4 Erweiterungen von PROLOG

Zusammenfassung

- Logik-Programmierung — der abstrakteste Programmierstil
- Auch der ineffizienteste
- Manche Anwendungen rechtfeetigen den *Overhead* trozdem

Nächstes Mal

- Ausblick: Zukunft der Programmiersprachen
- Ausblick: Implementierung von Programmiersprachen (Übersetzer + Praktikum im Sommer 2005)
- Auswertung der Lehrveranstaltung

Nachlese(n)

R. Kowalski.
Algorithm = Logic + Control.
Journal ACM 22: 424–436, 1979.

Ulf Nilsson, Jan Małuszyński.
Logic Programming, and PROLOG (2nd ed.).
URL www.ida.liu.se/~ulfni/lpp, 2000.
Vorher bei John Wiley & Sons, 1995.