Logical Programming

Niklas de Bruyn

Malte Schulze Balhorn

Robin Thielking

Logical Programming

„Algorithm = Logic + Control.“ -*R. Kowalski*

Logical Programming

„Algorithm = Logic + Control.“ -*R. Kowalski*

Logik	Steuerung
- Beschreibt "was" getan werden muss ⇒ Deklarativer Teil	- Legt fest, "wie" die angestrebte Lösung gefunden werden soll ⇒ Prozeduraler Teil

Warum Logical Programming?

⇒ Ansatz ist die Trennung beider Konzepte
⇒ Im Gegensatz zur imperativen Programmierung, ist der Entwickler hier grundsätzlich nur zu einer logischen Spezifikation verpflichtet

Sprache der Logik

Alphabet besteht aus zwei disjunkten Teilmengen an Symbolen:

- Logischen Symbolen:
  - Logische Konnektive: ∧, ∨, ¬, →, ↔
  - Quantifikatoren: ∀, ∃
  - Aussagenkonstanten: true, false
  - Punktuationssymbole: "("; ")"; ","
  - Variablen: X, Y, Z, …
- Nicht-logischen Symbolen: Funktions- und Prädikatszeichen mit Stelligkeit

<u>Hinweis:</u>
⇒ Funktionssymbole meist *f, g, h,* …

⇒ Prädikatsymbole meist *p, q, r,* …

Terme & Formeln

- Term als grundlegender Ausdruck, welcher durch die Anwendung eines Funktions- oder Operatorzeichen entsteht
- Konstrukte (Strings, Binärbäume, Listen, …) können auch als Terme betrachtet werden

Beispiel:

Seien *a*, *b* Konstanten; *X*, *Y* Variablen und *f*, *g* haben jeweils eine Stelligkeit von 2

Dann sind *f(a, b)* und *g(a, X)* aber auch *g(f(a, b), Y)* Terme.

- Formeln werden genutzt um Eigenschaften von Termen auszudrücken

Beispiel:

>(X,Y) ∧ >(Y,Z) → >(X,Z)

Logikprogramm

- Formeln können sehr komplex aufgebaut werden und schwer zu beweisen

⇒ Daher eingeschränkte Form: **Klauseln**

Es gibt zwei Arten von Klauseln:

- Fakten ⇒ cold. , male(homer) , father(homer,bart)
- Regeln ⇒ bigger(X,Y) :- X > Y , parents(F,M,C) :- father(F,C), mother(M,C)

Mit Queries können Fragen an die Umgebung gestellt werden.
⇒ ?- 1 < 2

⇒ ?- father(F,C)

Logikprogramm

?- father(homer,bart).
yes

?- father(homer,C).
C = bart

?- father(F,C).
C = bart
F = homer
yes;

C = lisa
F = homer
yes;

Angenommen:

father(homer,bart).
father(homer,lisa).

parents(F,M,C) :- father(F,C), mother(M,C).
siblings(A,B) :- parents(F,M,A), parents(F,M,B).
brother(A,B) :- siblings(A,B), male(B).

- Ursprünge liegen in den ersten Ideen zur Unifikation von ***Gödel*** und **Herbrand** in den 1930er
- ***Robinson*** entwickelte in den 1960er ein formales Unifikationsverfahren
- In den 1970er wird die automatische Deduktion als Berechnungsmodell entdeckt

Logische Variable

- Logische Variable kann einen Wert aus einer gegebenen Menge annehmen
  - Kann nur einmal gebunden werden. Wenn die Variabel X einmal zur Konstanten a gebunden ist, kann sie nicht mehr zu b gebunden werden
  - Kann teilweise definiert sein: X kann zu f(Y,Z) gebunden sein und Y und Z können erst später gebunden bzw. definiert werden, z.B. Y zu a und Z zu g(W), was X zu f(a, g(W)) machen würde
  - Bindungen sind bidirektional: Wenn X zu f(Y) gebunden ist es trotzdem erlaubt X zu f(a) zu binden durch die Bindung Y zu a. X zu f(Y) bleibt trotzdem bestehen

Substitution

- Eine Substitution ist eine Funktion von Variablen zu Termen

- sind unterschiedliche Variablen
- X_i ist unterschiedlich von t_i für alle i

Beispiel: Substitution

- $\vartheta = \{X/a, Y/f(W)\}$ und $g(X, W, Y)$
- $g(X, W, Y)\vartheta$ ergibt $g(a, W, f(W))$

Beispiel: Substitution

- Substitutionen werden gleichzeitig angewendet
- $\sigma = \{Y/f(X), X/a\}$ und $g(X, Y)$
- $g(X, Y)\sigma$ ergibt $g(a, \mathrm{f(X)})$ und nicht $g(a, \mathrm{f(a)})$

Komposition von Substitution

- Aus zwei Substitutionen lässt sich eine Komposition bilden

$$\vartheta = \vartheta_1 \vartheta_2$$

  - ϑ_2 auf ϑ_1 anwenden
  - Um alle Paare aus ϑ_2 erweitern
  - Doppelte Bindung aus ϑ_2 entfernen
  - Alle Paare $\mathrm{X_i}/\mathrm{t}\vartheta_2$ mit $\mathrm{X_i} = \mathrm{t}\vartheta_2$ entfernen
- Es gilt $(\mathrm{E}\vartheta_1)\vartheta_2 = \mathrm{E}(\vartheta_1\vartheta_2) = \mathrm{E}(\vartheta)$

Komposition von Substitutionen

- $\vartheta_1 = \{X/f(Y), W/a, Z/X\}$ und $\vartheta_2 = \{Y/b, W/b, X/Z\}$
- $\vartheta = \{X/f(b), W/a, Z/Z, Y/b, W/b\}$
- $\vartheta = \vartheta_1\vartheta_2 = \{X/f(b), W/a, Y/b\}$

Substitutionen

- $\vartheta \leq \sigma$ bedeutet ϑ is more general
- Es gibt eine Substitution γ so dass $\gamma\vartheta = \sigma$

Most General Unifier

- $\mathrm{f(X)} = \mathrm{f(g(Y))}$ ist lösbar mit der Substitution $\vartheta = \{X/g(Y)\}$
- ϑ ist ein unifier
- Aber $\sigma = \{X/g(f(Z)), Y/f(Z)\}$ ist auch ein unifier
- Es gilt aber $\vartheta \leq \sigma$ und auch für alle anderen unifier
- ϑ ist der most general unifier (m.g.u.)

Unification Algorithmus

- Input: Menge an Gleichungen $E = \{s_1 = t_1, \ldots, s_n = t_n\}$
- Output: Fehler oder Menge an Gleichungen $\{X_1 = r_1, \ldots, X_m = r_m\}$
- Ergibt den m.g.u. $\vartheta = \{X_1/r_1, \ldots, X_m/r_m\}$

Unification Algorithmus

- Wähle zufällig eine Gleichung

1. $f\,(l_1,\ldots,l_k) = f\,(m_1,\ldots,m_k)$ => lösche Gleichung aus E und füge $l_1 = m_1, \ldots, l_k = m_k$ hinzu
2. $f\,(l_1,\ldots,l_k) = g\,(m_1,\ldots,m_k)$ wenn f und g ungleich sind => Fehler
3. $X = X$ => lösche Gleichung aus E
4. $X = t$ => wende Substitution $\{X/t\}$ auf alle Gleichungen in E an, wenn X nicht in t ist

Unification Algorithmus

5. $X = t$ => Fehler wenn X in t ist
6. $t = X$ => wenn t keine Variable ist, dann lösche die Gleichung füge $X = t$ zu E

- Wiederhole bis alle Gleichungen durch sind

Beispiel: Unification Algorithmus

- $E = \{ f(X, b) = f(g(Y), W), h(X, Y) = h(Z, W) \}$
- $\mathrm{E}_1 = \{X = g(Y), b = W, h(X, Y) = h(Z, W)\}$ (1.)
- $\mathrm{E}_2 = \{X = g(Y), b = W, h(g(Y), Y) = h(Z, W)\}$ (4.)
- $\mathrm{E}_3 = \{X = g(Y), W = b, h(g(Y), Y) = h(Z, b)\}$ (6. und 4.)

- $\vartheta_1 = \{X/g(Y), W/b\}$ ist bereist m.g.u. für die erste Gleichung

Beispiel: Unification Algorithmus

- $\mathrm{E}_4 = \{X = g(Y), W = b, g(Y) = Z, Y = b\}$ (1.)
- $\mathrm{E}_5 = \{X = g(Y), W = b, Z = g(Y), Y = b\}$ (6.)
- $\mathrm{E}_6 = \{X = g(b), W = b, Z = g(b), Y = b\}$ (4.)

- $\vartheta = \{X/g(b), W/b, Z/g(b), Y/b\}$ ist der m.g.u. für E
- Üblicherweise wird für jede Gleichung in E der m.g.u. gefunden und dann wird die Komposition daraus gebildet

Herbrand-Universum

- Menge in der Prädikatenlogik
  - Das Alphabet ist nicht konstant
  - Nicht-Logiksymbole haben keine festgelegte Semantik -> ein "+" muss nicht Addition bedeuten
  - Es gibt keine Typen

Deklarativ oder Prozedural

- Man kann Logikaussagen Deklarativ oder Prozedural betrachten z.B. $H :- \mathrm{A_1}, \mathrm{A_2}, \ldots, \mathrm{A_n}$. LP-Sprachen betrachten es Prozedural
- Deklarativ: Wenn $\mathrm{A_1}, \mathrm{A_2}, \ldots, \mathrm{A_n}$ gilt, dann gilt H
- Prozedural: Um H zu berechnen musst du erst $\mathrm{A_1}, \mathrm{A_2}, \ldots, \mathrm{A_n}$ berechnen
  - H kann man als äquivalent einer Funktion sehen
  - $\mathrm{A_1}, \mathrm{A_2}, \ldots, \mathrm{A_n}$ sind die Instruktionen der Funktion

SLD Resolution

- Basiert auf dem Resolutionskalkül
  - Benutzt das Widerlegungsverfahren, um zu zeigen, dass die Negation einer Aussage nicht gegeben ist
- Grundlage für Prolog

Beispiel: SLD Resolution

- Gegeben ist folgendes Programm:
  - (1) ancestor(X, Y) ← parent(X, Y).
  - (2) ancestor(X, Z) ← parent(X, Y) ∧ ancestor(Y, Z).
  - (3) parent(X, Y) ← mother(X, Y).
  - (4) parent(X, Y) ← father(X, Y).
  - (5) father(emil, julia).
  - (6) mother(birgit, emil)

- Und folgende Anfrage:
  - ancestor(birgit, julia)

Beispiel: https://users.informatik.uni-halle.de/~brass/lp21/print/c5_sldre.pdf

Beispiel: SLD Resolution

- Durch (2) ergibt sich
  - parent(birgit, Y) ∧ ancestor(Y, julia)
- Durch (3) ergibt sich
  - mother(birgit, Y) ∧ ancestor(Y, julia)
- (5) mit {Y/emil}
  - ancestor(emil, julia)
- (1)
  - parent(emil, julia)

Beispiel: SLD Resolution

- (4)
  - father(emil, julia)
- Durch (6) ergibt sich die leere Klausel □
- Negation wurde widerlegt, also ist birgit ein ancestor von julia

Erweiterungen von Prolog

- **Erweitert:** Praktische Features über reine Logik hinaus (I/O, Arithmetik, Bibliotheken).
- **Reihenfolge:** Code-Ordnung bestimmt Ausführung und Ergebnisse.
- **Kontrollstrukturen:** Cut, If-Else, Negation, Arithmetik – nicht rein logisch.
- **Unterschied zur Theorie:** Prolog ist logik-inspiriert, aber technisch, unvollständig und prozedur

Arithmetik in Prolog

- Integer & Float verfügbar
- + – * // Operatoren
- Vergleichsoperatoren < <= >= =:=
- Auswertungsoperator **is**

Typische Stolperfallen

- Gleichsetzung speichert nur den Ausdruck
- Beispiel: „X = drei plus fünf“ ergibt den Ausdruck „drei plus fünf“

- Auswertungsoperator berechnet den Wert
- Beispiel: „X is drei plus fünf“ ergibt den Wert acht

Kontrollstrukturen

- Cut-Operator: !
- Disjunktion: G1 ; G2
- If-then-else: B -> C1 ; C2
- Implementiert über Cut
- Negation als Fehlschlag: not(G) bzw. \+ G

Prolog & Datenbanken

- **Fakten als gespeicherte Daten**
  Beispiel: „Direktflug von Bologna nach Paris.“
- **Regeln als berechnete Beziehungen**
  Beispiel: „Ein Flug existiert auch, wenn es einen Zwischenstopp gibt.“
- **Entspricht Tabellen und Views**
  Fakten = Tabelle, Regel = View.
- **Datalog als Prolog-Variante für Datenbanken**
  Ohne Funktionssymbole, wie klassische Datenbanken.
- **Rekursion ermöglicht komplexe Abfragen**
  Beispiel: „Alle Städte, die man von Bologna aus erreichen kann.“

Vorteile & Nachteile des Logik-Paradigmas

Vorteile

Deklarative Programmierung

Bi-direktionale Regeln

Nachteile

Ineffizientes Backtracking

Keine starken Typen

Aufgabe

```
parent(anna, ben).
parent(ben, clara).
parent(ben, hans).
parent(clara, dora).

ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

```
?- ancestor(anna, dora).
```

Aufgabe

```
parent(anna, ben).
parent(ben, clara).
parent(ben, hans).
parent(clara, dora).

ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

---

?- ancestor(anna, dora).
```

»

„anna ist parent von ben“
„ben ist parent von clara“
„clara ist parent von dora“

Also ist anna → ben → clara → dora eine Kette.

Antwort:
Ja, die Anfrage ist wahr, weil anna eine Vorfahrin von dora ist.

Weitere interessante Links

- SWI-Prolog's features
- Projog - Prolog interpreter for Java