Effiziente Prolog-Techniken

Übersicht

- Wie werden Prolog-Programme effizienter?
  - Differenzlisten
  - First-Argument Indexing
  - Endrekursion
  - Partielle Evaluation
  - Weitergehende Optimierungen
    - Funktionsweise der WAM ausnutzen
    - Einbinden von externen Routinen

Ziel

- Kennen der Techniken, zumindest der einfacheren

Differenzlisten

Jede beliebige Liste kann als Differenz zweier Listen dargestellt werden:

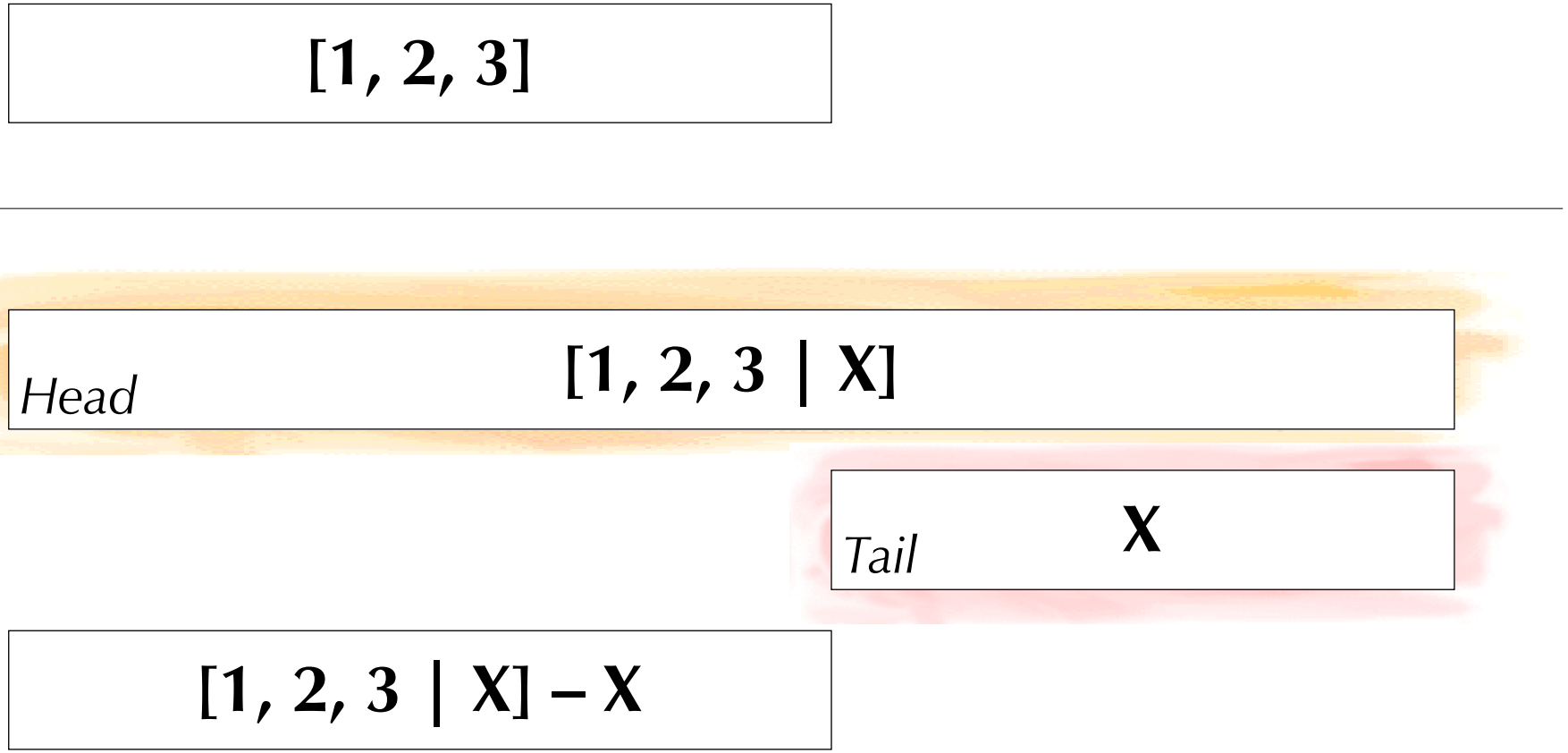
- [1, 2, 3] darstellbar als [1, 2, 3, 4, 5] – [4, 5]
- [1, 2, 3] darstellbar als [1, 2, 3, bla] – [bla]
- [1, 2, 3] darstellbar als [1, 2, 3] – [ ]

Allgemeiner:

- [1, 2, 3] darstellbar als [1, 2, 3 | X] – X

Differenzlisten

Grafische Veranschaulichung:



Differenzlisten

```
Head, Tail
[1, 2, 3 | X], X
```

zwei Terme

```
Head - Tail
[1, 2, 3 | X] - X
```

als Struktur mit einem (Infix-)Operator

Darstellung in Prolog:

- mit zwei Termen: etwas effizienter
- mit Operator: leichter lesbar
  - die Wahl des Operators ist beliebig, solange immer derselbe verwendet wird
  - auch Präfix- oder Postfix-Operatoren wären zulässig

Differenzlisten

Listen können leicht in Differenzlisten mit denselben Elementen umgewandelt werden:

`[a,b,…]` ⇒ `[a,b,…|X] - X`

`[]` ⇒ `X - X`

Differenzlisten

Warum ersetzt man überhaupt Listen durch die etwas komplizierteren Differenz-Listen?

- manche Operationen stellen mit normalen Listen einen gewissen Aufwand dar
  - → langsam
  - → grösserer Speicherplatzbedarf
- diese Operationen können mit Differenzlisten effizienter ausgeführt werden

Differenzlisten: Beispiel

Listen verketten (mit gewöhnlichen Listen):

Abbr.

```
append([], L2, L2).
```

Rekurs. Fall

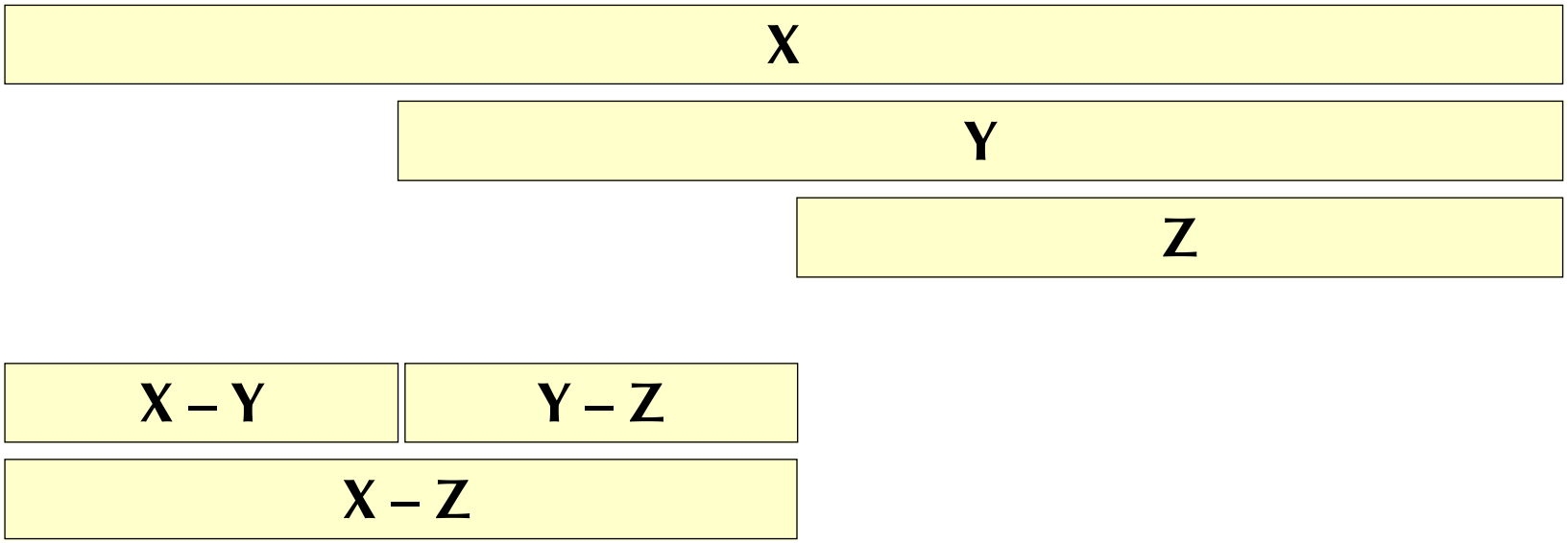
```
append([First1|Rest1],
       L2, [First1|Result]) :-
  append(Rest1, L2, Result).
```

Die Laufzeit ist proportional zur Länge der ersten Liste

- rekursives Abarbeiten der ersten Liste
- für jedes Element der ersten Liste ein Schleifendurchgang

Differenzlisten: Beispiel

Listen verketten (mit Differenzlisten):



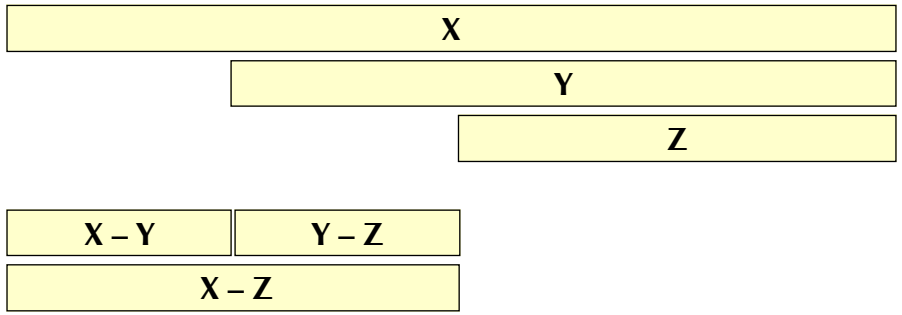

```
append_dl(X-Y, Y-Z, X-Z).
```

Differenzlisten: Beispiel

Listen verketten (mit Differenzlisten):

```
?- append_dl([1,2,3|A]-A,
             [4,5|B]-B,
             Result).

Result = [1,2,3,4,5|B]-B,
A = [4,5|B]-B
```




```
append_dl(X-Y, Y-Z, X-Z).
```

Die Laufzeit ist konstant

- keine Rekursion/Schleife, sondern eine einzige Term-Unifikation

First-Argument Indexing

Der Prolog-Interpreter muss bei jedem Schritt feststellen, welcher Klauselkopf mit einem Ziel unifiziert werden kann.

```
?- person(kevin).
yes.
```

```
person(hans).
person(klara).
person(sabrina).
person(kevin).

hund(fido).

weiblich(klara).
weiblich(sabrina).

maennlich(hans).
maennlich(kevin).
maennlich(fido).
```

First-Argument Indexing

Anstatt das gesamte Prolog-Programm zu durchsuchen, indizieren die meisten Prolog-Interpreter die Klauselköpfe *(Hashing)*.

- Als Schlüssel dient eine Kombination aus
  - Funktor
  - und dem ersten Argument
- Daher sind häufig solche Programme effizienter, welche die »bekannte« Information im *ersten* Argument (evtl. auch im Funktor) enthalten
  - wenn oft nach der Farbe von bestimmten Tieren gesucht wird:
    `farbe(frosch, gruen)`
  - wenn oft nach den Tieren mit einer bestimmten Farbe gesucht wird:
    `farbe(gruen, frosch)`

Endrekursion

Bestimmen der Länge einer Liste, rekursiv:

```
length([_ | Xs], N) :-
  length(Xs, N_minus_1),
① N is N_minus_1 + 1.
```

Diese Rekursion ist recht speicher- und rechenintensiv.

- denn bei jedem Aufruf muss sich Prolog Information merken, um anschliessend wieder zu ① zu gelangen.
  - sogenannter *Stack Frame*
- Speicherplatzbedarf wächst linear mit der Listenlänge

Endrekursion

Unter bestimmten Umständen muss sich der Prolog-Interpreter jedoch nichts merken:

- der rekursive Aufruf entsteht durch das *letzte* Ziel in der Klausel,
- alle anderen Ziele der Klausel haben keine weiteren Lösungen,
- und für das Prädikat gibt es keine alternativen Klauseln

Viele Prolog-Implementation erkennen diese Situation (sog. Endrekursion, *tail recursion*) und optimieren:

- die Rekursion wird durch einen Sprung ersetzt
- es wird kein *Stack Frame* angelegt

Endrekursion

Bestimmen der Länge einer Liste, end-rekursiv:

- diese Version ist effizienter
- läuft unabhängig von der Listenlänge in konstantem Speicherplatz

```
length(List, Length) :-
  length(List, 0, Length).

length([], Result, Result).

length([_|L], N0, Result) :-
  N1 is N0 + 1,
  length(L, N1, Result).
```

Akkumulator *enthält Zwischenresultat*
(Anzahl bisher angetroffener Elemente)

Partielle Evaluation

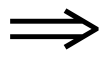
Eine weitere Technik zur Effizienzsteigerung heisst *Partielle Evaluation*:

- einige Prolog-Resolutionsschritte werden im Voraus vorgenommen und brauchen daher keine Rechenzeit mehr

```
person(hans).
person(klara).
person(sabrina).

weiblich(klara).
weiblich(sabrina).

frau(X) :-
  person(X),
  weiblich(X).
```



```
person(hans).
person(klara).
person(sabrina).

weiblich(klara).
weiblich(sabrina).

frau(klara).
frau(sabrina).
```

frau/1 besitzt dieselbe Lösungsmenge, ist aber schneller

Weitergehende Optimierungen

Ein Programm verbraucht 90% der Ausführungszeit in 10% seines Befehlscodes.

- Faustregel (»90/10-Leistungsregel«) aus der Computerarchitektur
- gilt für typische in imperativen Sprachen geschriebene Programme, lässt sich aber vermutlich auch auf logische Programmiersprachen übertragen

  Quelle: John L. Hennessy/David A. Patterson: Rechnerarchitektur. Analyse, Entwurf, Implementierung, Bewertung. Göttingen: Vieweg, 1994.
  (deutsche Übersetzung von »Computer Architecture. A Quantitative Approach«)

Es lohnt sich, bei Effizienzproblemen als erstes »die kritischen 10%« zu suchen und diese zuerst zu optimieren.

Weitergehende Optimierungen

Moderne Prolog-Systeme übersetzen (compilieren) Prolog in einfache Befehle für eine Abstrakte Maschine.

- »Warren Abstract Machine« (WAM)
  - Hassan Aït-Kaci: Warren's Abstract Machine. A Tutorial Reconstruction. Logic Programming. Cambridge, MA: MIT Press, 1991.
  - David H. D. Warren: An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Menlo Park, Calif., August 1983.
- verbreitete Technik
  - für andere Programmiersprachen: P-Code für Pascal · Byte-Code für Java
  - diese abstrakten Befehle werden von einem geeigneten Programm interpretiert
  - gelegentlich werden in einem zusätzlichen Compilationsschritt Instruktionen der abstrakten Maschine in »echte« Befehle für einen bestimmten Prozessor umgewandelt
    - Java: Just-in-Time-Compiler · Prolog: SICStus für SPARC, MIPS, 68020

Weitergehende Optimierungen

Wer die Funktionsweise der WAM kennt, kann Prolog-Programme so schreiben, dass sie effizienter abgearbeitet werden.

- Beispiel: Compiler für HPSG-artige Grammatiken erzeugt Prolog-Code, der im Hinblick auf die WAM optimiert ist.
  - Bob Carpenter/Gerald Penn: Compiling Typed Attribute-Value Logic Grammars. In: Harry Bunt/Masaru Tomita [ed.]: Current Issues in Parsing Technologies, vol. 2. Dordrecht: Kluwer Academic Publishers, 1995.

Weitergehende Optimierungen

Eine andere Möglichkeit ist das Einbinden externer Routinen in Prolog.

- Professionelle Prolog-Systeme erlauben, ein Prädikat in einer anderen Programmiersprache (z.B. C) zu schreiben
- Auch die Umkehrung (Aufruf von Prolog-Prädikaten aus C heraus) ist manchmal möglich

Die effizienz-kritischen Teile eines Programms können so mit maschinennahen Programmiersprachen geschrieben werden, ohne völlig auf Prolog zu verzichten.

A: Effiziente Prolog-Techniken

Programmiertechniken der Computerlinguistik 2 · Sommersemester 1999

1. First-Argument-Indexing

Verbessere die Effizienz des folgenden Programms, indem Du das von den meisten Prolog-Interpretern verwendete *First-Argument-Indexing* berücksichtigst.

```
lebendig(A) :-
  geburt(_, A),  % das Geburtsjahr ist uns egal
  \+ tod(_, A).  % das Todesjahr auch

geburt(1978, bello).
geburt(1989, fido).
geburt(1947, hans).
geburt(1868, hermine).
geburt(1975, kevin).
geburt(1945, klara).
geburt(1972, sabrina).
geburt(1990, schnurrli).

tod(1981, bello).
tod(1905, hermine).
```

2. Warteschlangen

Eine Warteschlange (engl. *Queue*) ist eine Datenstruktur, bei der Elemente hinten angefügt und vorne entfernt werden. Bei einem Keller/Stack werden hingegen Elemente sowohl vorne eingefügt als auch vorne entnommen.

Eine Warteschlange besitzt somit eine sogenannte *First-In-First-Out*-Strategie (FIFO), während ein Keller nach dem Prinzip *Last-In-First-Out* (LIFO) arbeitet.

Implementiere Warteschlangen in Prolog mittels Differenzlisten:

a) Schreibe ein Prolog-Prädikat `empty_queue/1`, das überprüft, ob eine Warteschlange leer ist. Wenn das Argument eine freie Variable ist, soll diese Variable an eine leere Warteschlange gebunden werden.

b) Schreibe eine Prolog-Klausel `enqueue/3`, die ein Element (1. Argument) in eine Warteschlange (2. Argument) einfügt und die neue Schlange als 3. Argument zurückgibt.

c) Schreibe ein Prolog-Prädikat `dequeue/3`, das ein Element von einer Warteschlange entfernt und die modifizierte Schlange zurückliefert.

d) Schreibe eine Implementation mit gewöhnlichen Listen. Warum ist diese weniger effizient?

Beispiel für die Anwendung dieser Prädikate:

```
?- empty_queue(Initial),
   enqueue(one, Initial, Q1),
   enqueue(two, Q1, Q2),
   dequeue(FirstElement, Q2, Q3),
   dequeue(SecondElement, Q3, Final),
   empty_queue(Final).

FirstElement = one, SecondElement = two.
```