Fortgeschrittene logikorientierte Programmierung 185.209 VL 2
Prolog und logikorientierte Programmierung
185.988 VO 1, 185.015 LU 2

- Mittwoch, 3. Juni 2009, 17h00
- Beispiele bis ⇈74.
- Inhalt:
  - Programmieren zweiter Ordnung
  - Meta-Programmierung
  - Coroutining
  - Suchverfahren
  - Programmtransformation

Mengenausdrücke, Programmieren zweiter Ordnung

Prologziel beschreibt Lösungsmenge (Lösungssequenz) implizit.

kind_von(joseph_II, maria_theresia).
kind_von(marie_antoinette, maria_theresia).
kind_von(maria_theresia, karl_VI).
← kind_von(Kind, maria_theresia).

Wieviele Kinder hat Maria Theresia?

Aggregationen

- Anzahl
- Summe
- Durchschnitt
- Maximum

Explizite Darstellung der Lösungsmenge, z.B. als Liste.

(Umständliche) Berechnung mittels Iteration:

```
kinder_von(Kinder, Elternteil) ←
    kinder_von(Kinder, [], Elternteil).

kinder_von(Kinder, Kinder, Elternteil) ←
    \+ ( kind_von(Kind, Elternteil), \+ member_of(Kind, Kinder) ).
% Alle Kinder von Elternteil kommen in Kinder vor
kinder_von(Kinder, Kinder0, Elternteil) ←
    kind_von(Kind, Elternteil), % ◊ nichtdet.
    \+ member_of(Kind, Kinder0), % ◊ quadratisch
    kinder_von(Kinder, [Kind|Kinder0], Elternteil).

← kinder_von(Kinder, maria_theresia).
% Kinder = [marie_antoinette, joseph_II].
% Kinder = [joseph_II, marie_antoinette]. % ◊ alle Permutationen

← kinder_von([], Person). % Wer hat keine Kinder?
% Ähnliche Probleme wie Negation
```

setof(Lösungsschema, Ziel, Lösungsmenge)

Welche Kinder hat Maria Theresia? (Wenn Sie überhaupt Kinder hat.)

$\leftarrow$ setof(Kind, kind_von(Kind,maria_theresia), Kinder).
% Kinder = [joseph_II, marie_antoinette].

- Kinder sortierte Liste
- $\not\leftarrow$ setof(.., .., []).
- VAR(Lösungsschema) $\subseteq$ VAR(Ziel)
- VAR(Lösungsschema) kommen nur in Ziel u. Lösungsschema vor
- VAR(Lösungsschema) werden nie gebunden
- Alle Lösungen für Ziel müssen variablenfrei sein
- VAR(Ziel) – VAR(Lösungsschema) liefern unabhängige Lösungen

Wer hat welche Kinder? (Wenn überhaupt)

$\leftarrow$ setof(Kind, kind_von(Kind,Person), Kinder).
% Kinder = [joseph_II, marie_antoinette], Person = maria_theresia.
% Kinder = [maria_theresia], Person = karl_VI.

Ausblenden von Argumenten:

Welche Eltern gibt es? (Kinder uninteressant)

← setof(Person, Kind↑kind_von(Kind,Person), Eltern).
% entspricht
elternteil(Person) ←
    kind_von(_Kind, Person).
← setof(Person, elternteil(Person), Eltern).

Welche Großeltern gibt es?

← setof(Person, Enkel↑Kind↑ (kind_von(Enkel, Kind), kind_von(Kind, Person)) , Großeltern).
% entspricht
großelternteil(Person) ←
    kind_von(_Enkel, Kind),
    kind_von(Kind, Person).
← setof(Person, großelternteil(Person), Eltern).

Ausschließen von Duplikaten:

← setof(t, kind_von(Kind,Person), _Lösungsmenge).
← kind_von(Kind,Person).

Alle Bindungen nach außen sichtbar.

Grenzen von setof/3**:**

Welche Ziele gibt es, sodaß ...

← setof(Kind, (Ziel, arg(1,Ziel,Kind)), [joseph_II, marie_antoinette]).

Anwendungen:

- Aggregationen
- Umformung Fakten nach Bäumen
- Suche

Metacall

Ein Ziel kann auch eine Variable sein. Muß zum Zeitpunkt des Aufrufs gebunden sein.

← Ziel, Ziel = true. % Fehler
← Ziel = true, Ziel.
← call(Ziel), Ziel = true. % Fehler
← Ziel = true, call(Ziel).

Könnte in Prolog definiert werden:

```
call(Ziel) ←
    var(Ziel),
    fehler('Metacall mit freier Variable').
call(true).
call(nat_nat_summe(A, B, C)) ←
    nat_nat_summe(A, B, C).
call(kind_von(Kind, Person)) ←
    kind_von(Kind, Person).
... .
```

Verwendung:

Meist für allgemeine Prädikate, wie z.B. setof/3.

```
für_alle(Generator, Test) ←
    \+ ( Generator, \+ Test ).
```

```
mapcar(_Schema, [], []).
mapcar(Schema, [X|Xs], [Y|Ys]) ←
    copy_term(Schema, X↑Y↑Ziel), % explizite Instanzierung
    Ziel,
    mapcar(Schema, Xs, Ys).

← mapcar(X0↑X1↑ (X1 is X0 + 1), [1,2,3], [2,3,4]).
← mapcar((A-_)↑A↑true, [a-1,b-2], [a,b]).

apply(P, ZArgs) ←
    P =.. PArgs,
    phrase((liste(PArgs),liste(ZArgs)),Args),
    Ziel =.. Args,
    Ziel.

call(Cont, X1) ←
    apply(Cont,[X1]).
call(Cont, X1,X2) ←
    apply(Cont,[X1,X2]).
call(Cont, X1,X2,X3) ←
    apply(Cont,[X1,X2,X3]).
...
```

```
maplist(_P, [], []).
maplist(P, [X|Xs], [Y|Ys]) ←
    call(P, X, Y),
    maplist(P, Xs, Ys).

pair_left(A-_,A).

inc(X0, X1) ←
    X1 is X0 + 1.

← maplist(inc, [1,2,3], [2,3,4]).
← maplist(pair_left, [a-1,b-2], [a,b]).

maplist(_P, []).
maplist(P, [X|Xs]) ←
    call(P, X),
    maplist(P, Xs).

← Xs = [X,Y,Z], maplist(dif(a),Xs).
```

Wiederverwendbarkeit

- Formatierte Daten
- Programmtext
- Termdefinitionen is_, Datenstrukturen
- Prädikate, Prozeduren
- Abstrakte Datentypen (z.B. make/next/done)
- Spezialisierte Sprachen
  - Protokolle, z.B. IDL
  - Masken- Formularbeschreibungssprachen, SGML, HTML
  - Grammatiken z.B. DCGs
    abstrahiert Listendifferenzenpaar

Meta-Programmierung

Programme als Daten

- Parser
- Compiler
- Übersetzer (z.B. FORTRAN $\rightarrow$ C)
- Makro-Prozessoren
- *pretty printer*, *cross referencer*
- Interpreter
- Debugger
- Optimierer
- Programmtransformatoren
- Programme zur Fehlersuche (z.B. *lint*)
- Allgemeine Spracherweiterungen — *domain specific languages*

Spracherweiterungen mittels Meta-Interpreter

Ausgehend von meta-zirkulärem Interpreter

Voraussetzungen für Meta-Programmierung

- Definition einer Datenstruktur zur Darstellung eines Programms

  Wünschenswert: Definition = Sprachdefinition

  - Abstrakter Syntaxbaum (AST), is_ Definition

- Verhältnis Anzahl der Sprachmittel & Komplexität der Syntax zu Mächtigkeit der Sprachmittel ausgeglichen

  - Maschinensprachen: einfache Syntax, einfache Semantik
  - Prozedurale Sprachen: komplexe Syntax, meist kein AST, viele Sprachelemente
  - Smalltalk, LISP, Prolog: AST als Term, wenige Sprachelemente

- Erwünschter Detaillierungsgrad sollte Komplexität des Meta-Programms bestimmen.
  - Einfache Datenstrukturen
  - *reification*: explizites Ausprogrammieren von Teilen der Sprache
    * Unifikation
    * Bindungsumgebung für Variablen
    * Prozeduraufruf
    * *backtracking*
  - *absorption*: implizite Wiederverwendung von Sprachmitteln

vanilla meta-interpreter

```
mi(true).
mi((A,B)) ←
    mi(A),
    mi(B).
mi(Goal) ←
    clause(Goal,Body),
    mi(Body).

is_body(G) ← % ♢ Mehrdeutig
    is_goal(G).
is_body((A,B)) ←
    is_body(A),
    is_body(B).
```

explizit: Konjunktionen
implizit: Unifikation, *backtracking*

vanilla meta-interpreter II

```
mi(true).
mi((A,B)) ←
    mi(A),
    mi(B).
mi(g(Goal)) ←
    mi_clause(Goal,Body),
    mi(Body).

is_body(true).
is_body(g(G)) ←
    is_goal(G).
is_body((A,B)) ←
    is_body(A),
    is_body(B).
```

vanilla meta-interpreter III

```
mi_list([]).
mi_list([G|Gs]) ←
    mi_lclause(G,Hs),
    mi_list(Hs),
    mi_list(Gs).

is_body([]).
is_body([G|Gs]) ←
    is_goal(G),
    is_body(Gs).
```

Explizite Unifikation

```
mi_list([]).
mi_list([G|Gs]) ←
    mi_lclause(H,Hs),
    unify(G,H),
    mi_list(Hs),
    mi_list(Gs).
```

Linearer Meta-Interpreter

```
mi_applist([]).
mi_applist([G|Gs]) ←
    mi_lclause(G,Hs),
    append(Hs,Gs,Is),
    mi_applist(Is).
```

```
always_infinite ←
    always_infinite,
    fail.
```

Linearer Meta-Interpreter mit Listendifferenzen

```
mi_dllist([]).
mi_dllist([G|Gs]) ←
    mi_dlclause(G,Gs0,Gs),
    mi_dllist(Gs0).
```

```
mi_dlclause(h(X),[g(X)|Gs],Gs).
```

Resolution explizit

```
demonstrate(_Prog,Goals) ←
    empty(Goals).
demonstrate(Prog,Goals) ←
    select(Goal,Goals,RestGoals),
    member_of(Procedure,Prog),
    renamevars(Procedure,Goals,ProcedureR),
    parts(ProcedureR,Head,Body),
    match(Goal,Head,Sub),
    add(Body,RestGoals,InterGoals),
    apply(InterGoals,Sub,NewGoals),
    demonstrate(Prog,NewGoals).
```

Bindungsumgebung explizit

```
mi_be(true,E,E,N,N).
mi_be((A,B),E0,E,N0,N) ←
    mi_be(A,E0,E1,N0,N1),
    mi_be(B,E1,E,N1,N).
mi_be(g(Goal),E0,E,N0,N) ←
    mi_be_clause(Goal,N0,Head,Body),
    add_equation(Goal=Head,E0,E1),
    N1 is N0 + 1,
    mi_be(Body,E1,E,N1,N).
```

```
← if(Bedingung, Then, Else).

if(Bedingung, Then, _Else) ←
    Bedingung,
    Then.
if(Bedingung, _Then, Else) ←
    \+ Bedingung,
    Else.

← (Bedingung → Then ; Else).
(Bedingung → Then ; _Else) ←
    Bedingung,
    !,
    Then.
(_Bedingung → _Then ; Else) ←
    Else.
```

Verbesserte Berechnungsstrategien

Dynamisches Umordnen von Zielen. *coroutining*

Block Deklaration:

Ausführung wird verzögert, bis durch – gekennzeichnete Argumente gebunden sind.

```
← block freeze(-,?).
freeze(_Var,Ziel) ←
    Ziel.

← block is_list(-).
is_list([]).
is_list([_X|Xs]) ←
    is_list(Xs).

← block greater_than(-,?), greater_than(?,-).
greater_than(A,B) ←
    A > B.
← block smallerequal_than(-,?), smallerequal_than(?,-).
smallerequal_than(A,B) ←
    A =< B.
```

```
← block list_list_merged(-,?,-), list_list_merged(?,-,-).
list_list_merged([], Ys, Ys) ←
    is_list(Ys).
list_list_merged(Xs, [], Xs) ←
    is_list(Xs).
list_list_merged([H|Xs], [E|Ys], [H|Zs]) ←
    smallerequal_than(H,E),
    list_list_merged(Xs, [E|Ys], Zs).
list_list_merged([H|Xs], [E|Ys], [E|Zs]) ←
    greater_than(H,E),
    list_list_merged([H|Xs], Ys, Zs).

← list_list_merged(Xs,Ys,Zs).
% list_list_merged(Xs,Ys,Zs).
% Eine Lösung gefunden

← list_list_merged(Xs,Ys,Zs), Xs = "a".
% Xs = "a", list_list_merged("a",Ys,Zs).
% Eine Lösung gefunden

← list_list_merged(Xs,Ys,Zs), Xs = "ac", Ys = "bd".
% Xs = "ac", Ys = "bd", Zs = "abcd".
% Eine Lösung gefunden
```

← list_list_merged(Xs,Ys,Zs), Zs = "abcd".
% Xs = [], Ys = "abcd", Zs = "abcd".
% Xs = "abcd", Ys = [], Zs = "abcd".
% Xs = "a", Ys = "bcd", Zs = "abcd".
% Xs = "ab", Ys = "cd", Zs = "abcd".
% Xs = "abc", Ys = "d", Zs = "abcd".
% Xs = "abd", Ys = "c", Zs = "abcd".
% Xs = "acd", Ys = "b", Zs = "abcd".
% Xs = "ac", Ys = "bd", Zs = "abcd".
% Xs = "ad", Ys = "bc", Zs = "abcd".
% Xs = "bcd", Ys = "a", Zs = "abcd".
% Xs = "b", Ys = "acd", Zs = "abcd".
% Xs = "bc", Ys = "ad", Zs = "abcd".
% Xs = "bd", Ys = "ac", Zs = "abcd".
% Xs = "cd", Ys = "ab", Zs = "abcd".
% Xs = "c", Ys = "abd", Zs = "abcd".
% Xs = "d", Ys = "abc", Zs = "abcd".
% 16 Lösungen gefunden

← list_list_merged([1|Xs], [2|Ys], Zs).
% Zs = [1|_A], list_list_merged(Xs,[2|Ys],_A).
% Eine Lösung gefunden

← list_list_merged([1,3|Xs], [2|Ys], Zs).
% Zs = [1,2|_A], list_list_merged([3|Xs],Ys,_A).
% Eine Lösung gefunden

Verallgemeinerte Deklarationen

```
← when(Bedingung, Ziel).

is_bedingung(nonvar(_X)).
is_bedingung(ground(_X)).
is_bedingung(?=(_X,_Y)).
is_bedingung((A,B)) ←
    is_bedingung(A),
    is_bedingung(B).
is_bedingung((A;B)) ←
    is_bedingung(A),
    is_bedingung(B).

dif(X,Y) ←
    when(?=(X,Y), X \== Y).
```

Verzögerung von Zielen gelegentlich zu konservativ.

```
← greater_than(A,B), greater_than(B,A).
```

Vordefinierte Prädikate:

```
← frozen(Var, Ziel). % Var ist an Ziel gebunden
← call_residue(Ziel, ResidualeZiele).
```

Suchverfahren

- Zustand

  Grundterme

- Zustandsübergang

- Tiefensuche

  geringer Platzverbrauch

- Breitensuche

  findet Lösung mit kürzestem Pfad

  – Wave search
  – Inter-wave search

Heuristiken

- Wissen über Zustände (z.B. Straßenrichtungen)
- Optimale Lösung nicht erforderlich (unvollständige Lösung)
- Ausschließen unsinniger Übergänge
- Bewertungsfunktionen, evaluation function
  Bewertet einen konkreten Zustand
- Hill-climbing = Tiefensuche mit Bewertungsfunktion
  jeder Zwischenzustand ist bester momentaner Zustand
- Best-first = Breitensuche mit Bewertungsfunktion
- Kostenfunktion Bewertet bisherige Zustände
- Branch and bound (best cost) = Tiefensuche mit Kostenfunktion
  findet Lösung mit minimalen Kosten
- A* (best path) = Breitensuche mit Kostenfunktion und Bewertungsfunktion

Darstellung von beliebig langen Iterationen

```
make_p(N,N).              iteration_I(N) ←
                              make_p(N,S0),
done_p(1).                    iter_I(S0).

next_p(N0, N) ←           iter_I(S) ←
    N0 > 1,                   done_p(S).
    0 is N0 mod 2,        iter_I(S0) ←
    N is N0 // 2.             next_p(S0,S), % \+ done_p(S0)
next_p(N0, N) ←               iter_I(S).
    N0 > 1,
    1 is N0 mod 2,
    N is 3 * N0 + 1.
```

```
iteration_II(N) ←
    make_p(N, S0),
    iter_II(S0, S),
    done_p(S).

iter_II(S, S).
iter_II(S0, S) ←
    next_p(S0, S1),
    iter_II(S1, S).
```

```
iteration_II(N, IS) ←
    make_p(N, S0),
    iter_II(S0, S),
    show(S, IS).

show(S, ende(S)) ←
    done_p(S).
show(S, zw(S)) ←
    \+ done_p(S).

← iteration_II(27,IS).
% IS = zw(27).
% IS = zw(82).
% ...
% IS = zw(2). % IS = ende(1). % 112 Lösungen gefunden
```

Programmtransformation

- Verbesserung des Ressourcenverbrauchs
- erlauben klareren, kompakteren aber ineffizienteren Programmierstil
  - Bibliotheken
    Spezialisierung von allgemeinen aber ineffizienten Prädikaten
    z.B. make/next/done Schnittstelle, Metacalls
  - Meta-Interpreter
    ca. eine Größenordnung pro Sprachebene
    geschachtelte Meta-Interpreter
    oft Meta-Interpreter + Prolog (MI + Scheme) effizienter als spezielle Implementierung
  - dynamische Operationen werden statisch ausgeführt
- exekutierbare Spezifikationen
- Reduzieren Programme auf das „Wesentliche“

Compiler vs. Programmtransformationssystem — Unterschiede fließend. Compiler:

- übersetzen Hochsprache in maschinennähere Sprache
  Eiffel → C, C → asm, Prolog → WAM-code, WAM-code → Maschinencode.
- konzeptueller Aufbau: paßorientiert
  Programmtext → AST → Zwischencode → Maschinencode
  (Teile manchmal ähnlich Transformationssystem)
- meist hoher Aufwand für Syntax
- klare Algorithmen zur Übersetzung
- wenige Heuristiken bei Codegenerierung
- sehr lokale Optimierungen, meist z.B. Prozedur, Prädikats- Klauselebene
- geht kaum auf „Intention des Programmierers“ ein
- wenige Annotationen/Optionen zur Steuerung von Optimierungen (z.B. inline-, Register- Deklarationen)
- Algorithmen, Fehler, Endlosschleifen „bleiben erhalten“
- automatisch, keine Interaktion

Transformationssystem

- übersetzen meist in gleiche Sprache *source to source*
- konzeptueller Aufbau:
  - wenige Regeln
    beschreiben mögliche Äquivalenzumformungen
  - viele Strategien
    steuern Regelanwendung, können Korrektheit nicht beeinflussen

  Algorithmen = Regeln + Strategien
- große Programmteile werden auf einmal betrachtet, Programmstruktur wird stark verändert
- verändern (verbessern) u.U. Algorithmen
- oft interaktiv, Annotationen, Heuristiken

- Problemstellung
  - Transformation eines gesamten Programms
  - Spezialisierung für wenige nach außen sichtbare Prädikate
- Äquivalenz
  - Deklarativ
  - Prozedural
    - SLD-Ableitungsbäume
      - Endlosableitungen
      - Endliches Scheitern
    - Cuts
    - Seiteneffekte
    - alles Beobachtbare außer Ressourcenverbrauch

  Seiteneffekte behindern oft Transformationen

- Ebene der Transformationsregeln

  meist über „Kontrollstrukturen“

  – Ziele, Regeln, Prädikate
  – Terme

  ? ähnlich wie erfahrener Programmierer ?

- Repräsentation der Programmteile

  meist AST $\rightarrow$ AST

  Idealer Ansatz: Meta-interpreter zur Definition der Transformationsebenen

  Detaillierungsgrad und Datenstrukturen des Meta-interpreters bestimmen Transformationsregeln

fold/unfold-Transformationssystem

- unfold — Ersetzen eines Ziels durch entsprechende Definition
  (inspiriert durch Programmexekution)
- fold — Umkehrung von unfold
  Einschränkung: unfold führt zu ursprünglichem Programm
- definition
  Erfindung eines neuen Prädikats (meist einer Regel)

Probleme von Strategien:

- Heuristik zum „Erfinden“ einer Definition — Eureka
  - Definition verwendet nur bestehende Ziele
  - Definition besteht nur aus einer einzigen Regel
  - generalisiertes Ziel
  - Ziele mit gemeinsamer existentieller Variable
- Suchraum sehr groß
- Termination der Strategie

```
sum([], 0).
sum([I|Is], N0) ←
    sum(Is, N1),
    N0 is N1 + I.

prod([], 1).
prod([I|Is], N0) ←
    prod(Is, N1),
    N0 is N1 * I.

sumprod(Is, S, P) ← % EUREKA!
    sum(Is, S),
    prod(Is, P).

sumprod(Is, S, P) ←
    sum(Is, S), % unfold
    prod(Is, P).
```

```
sumprod([], 0, P) ←
    prod([], P). % unfold
sumprod([I|Is], S0, P) ←
    sum(Is, S1),
    S0 is S1 + I,
    prod([I|Is], P). % unfold

sumprod([], 0, 1).
sumprod([I|Is], S0, P0) ←
    sum(Is, S1),
    S0 is S1 + I, % reorder
    prod(Is, P1),
    P0 is P1 * I.
```

```
sumprod([], 0, 1).
sumprod([I|Is], S0, P0) ←
    sum(Is, S1), % fold
    prod(Is, P1),
    S0 is S1 + I,
    P0 is P1 * I.

sumprod([], 0, 1).
sumprod([I|Is], S0, P0) ←
    sumprod(Is, S1,P1),
    S0 is S1 + I,
    P0 is P1 * I.
```

bekannteste Strategie für fold/unfold:

partielle Evaluation

partial deduction, partial execution, mixed computation

- Vermeidung von großem Suchraum durch starke Anlehnung an Exekution
- Definition nur für ein einziges Ziel
- Arity raising
- gesteuert durch bekannte Daten

z.B. mixtus (in Umgebung integriert) $\leftarrow$ pe append([1,2,3],Ys,Zs).

Futamura-Projektionen

1. target = mix(interpreter, source)
2. compiler = mix(mix,interpreter)
3. compilergenerator = mix(mix,mix)

Wenn 2 und 3: selbstandwendbarer Partieller Evaluator
Komplexität vs. Verarbeitbarkeit

Geschichte von Prolog

Repräsentation von Wissen: Prozedural oder deklarativ?

1. Robinson 1965, Kowalski 1970, Colmerauer 1972

2. Backtracking-Parser, vW-Grammatiken, système Q, DCG-Kodierung

```
POURQUOI EST-CE QUE JE NE SUIS PAS DIEU?
LA MACHINE. - PARCE QUE JE SAIS RAISONNER.
ET QUE VOUS M'AVEZ DIT: 'JE SUIS HORACE.'
```

Metamorphosis grammar Colmerauer 1978

q ⟶	q([a,b]).
"ab".	
p ⟶	p(I) ←
q,	append(I0,I12,I),
r,	q(I0),
s.	append(I1,I2,I12),
	r(I1),
	s(I2).

Dataloggrammatiken

Eingabestring als Fakten	zeichen(a,1,2).
	zeichen(b,2,3).

```
q ⟶              q(X0,X) ←
    "ab".             zeichen(a,X0,X1),
                      zeichen(b,X1,X).

p ⟶              p(X0,X) ←
    q,                q(X0,X1),
    r,                r(X1,X2),
    s.                s(X2,X).
```

```
list_listdiff(L,Es0,Es) ←
    append(L,Es,Es0).
```

$$p(\overbrace{X0, X}^{I}) \leftarrow q(\overbrace{X0, X1}^{I0}), r(\overbrace{X1, X2}^{I1}), s(\overbrace{X2, X}^{I2}).$$

$$I = I0 + I1 + I2$$

$$I \ = X0 - X, I0 = X0 - X1, I1 = X1 - X2, I2 = X2 - X$$

$$X0 - X = (X0 - X1) + (X1 - X2) + (X2 - X)$$

$$p(\overset{X0}{\overrightarrow{X0, X) \leftarrow q(X}}0, X1), r(X1, X2), s(X2, X).$$

$$p(X0, X) \leftarrow q(X0, \overset{X1}{\overrightarrow{X1), r(X}}1, X2), s(X2, X).$$

$$p(\overset{X0}{\overrightarrow{X0, X) \leftarrow q(X}}0, \overset{X1}{\overrightarrow{X1), r(X}}1, \overset{X2}{\overrightarrow{X2), s(X}}2, X).$$

$$p(\overbrace{X0, X) \leftarrow q(X0}^{X0}, \overbrace{X1), r(X1}^{X1}, \overbrace{X2), s(X2}^{X2}, X).$$

Fehlervermeidung:

- Zusicherungen, entsprechen Integritätsbedingungen *integrity constraints* in DBS.
- Zusicherung, daß ein Ziel nie scheitert.

  ← op(900,fy,[@]).

  @ X ←
      if(X, true, fehler(gescheitert-X)).

- Typsysteme

  Prolog „Typ-los", dynamische Typen

  ← append([1],a,Xs).

  - Meta-Interpreter mit typprüfenden Erweiterungen
  - Wenn residuales Programm keine Ziele für Fehlermeldungen enthält, ist Programm statisch typsicher.

Übliche Typsysteme

- eigener Formalismus zur Beschreibung von Typen

  type list(T) $\longrightarrow$ [] ; [T | list(T)].
  pred append(list(B),list(B),list(B)).

  append([], Xs, Xs).
  append([X|Xs], Ys, [X|Zs]) $\leftarrow$
      append(Xs, Ys, Zs).

- Typen z.T. automatisch abgeleitet
- meist sehr schwache Typen

  z.B. unmöglich: Liste, die Wörter einer allg. Grammatik beschreibt
- sehr aufwendige statische Analyse
- statische Analyse sehr bald unentscheidbar

Typsystem von Naish

- Typdeklarationen werden mittels Prolog-Regeln dargestellt
- allgemein statisch unentscheidbar
- unentscheidbare Teile werden dynamisch überprüft

```
list_list_merge(A,B,C) type
    sorted(A),
    sorted(B),
    sorted(C).

append(A,B,C) type
    list_of(T,A),
    list_of(T,B),
    list_of(T,C).

list_of(_T,[]).
list_of(T,[E|Es]) ←
    is_a(T,E),
    list_of(T,Es).
```

```
is_a(boolean,B) ←
    is_boolean(B).
is_a(integer,I) ←
    integer(I).
is_a(list_of(T),Es) ←
    list_of(T,Es).

is_boolean(true).
is_boolean(false).
```

Typen vs. Zusicherungen

```
p_while(X,_) type
    invariant(X).
```

```
p_while(X,X) ←
    \+ cond(X).
p_while(X0,X) ←
    cond(X0),
    update(X0,X1),
    p_while(X1,X).
```

Beispiele zur partiellen Evaluation

```
append([], Xs, Xs).
append([X|Xs], Ys, [X|Zs]) ←
    append(Xs, Ys, Zs).

← suffix(Xs, Ys).
← präfix(Xs, Ys).
← element(X, Xs).
← letztes(X, Xs).

a(X,Y,Ys,Zs) ←
    append([X,Y],Ys,Zs).
```

Constraints

- Constraint Logic Programming
  - domain of computation (z.B. Ganzzahlen, Rationale Zahlen, Boolsche Werte) CLP(X): CLP($\Re$), CLP(Q), CLP(B), CLP(FD)
  - neue Relationen (Constraints) zwischen Werten. (z.B. $<$, $>$)
  - Gleichungslöser
- Prolog als Spezialfall, Terme, =/2, dif/2

\+ einfachere Problemformulierung

\+ oft effizienter, da weniger Lösungen interner Ziele

\- Unifikationsalgorithmen i.a. unentscheidbar/sehr komplex, daher oft nur Annäherungen statt vollständiger Unifikation.

Konsistenztechniken

- arch-consistency: rel(A,B) Entfernung inkonsistenter Werte zwischen A und B
- globale Konsistenz kann nicht zusgesichert werden (zu teuer)
- letztendlich müssen konkrete Werte eingesetzt werden, um globale Konsistenz zu erzwingen (labeling)

CLP(FD)

Finite Domains = Ganzzahlen (SICStus, SWI)

Relationen

X in Min..Max, #=, #\=, #<, #=<, #>=, #>

Arithmetische Ausdrücke

+,-,*,/,mod,min(_,_),max(_,_),abs(_)

← X #< Y.
% Y in 1..inf, X in 0..inf.
% Eine Lösung gefunden

← [X,Y,Z] ins 1..10, 2*X + 3*Y + 2 #< Z.
% Y = 1, X in 1..2, Z in 8..10.
% Eine Lösung gefunden

Labeling

indomain(V) labeling([],Vs) labeling([ff],Vs) labeling([ffc],Vs)

```
[] ins _ .
[X|Xs] ins Domain ←
    X in Domain,
    Xs ins Domain.
```

element(I, Ys, Y) I-tes Element in Ys ist Y

relation(X, Map, Y) X Map Y Map = [X1-{Y11,Y12,...}, ...]

Konsistenztechniken $\neq$ Unifikation

$\leftarrow$ A+B #= 1, A+B+C #= 2.
% A in 0..1, B in 0..1, C in 0..2.
% Eine Lösung gefunden
$\leftarrow$ A+B #= 1, A+B+C #= 2, labeling([],[A,B,C]).
% A = 0, B = 1, C = 1.
% A = 1, B = 0, C = 1.
% 2 Lösungen gefunden

(Unifikation würde C=1 finden).

$\leftarrow$ [A,B,C] ins 1..2, A #\= B, B #\= C, C #\= A.
% A in 1..2, B in 1..2, C in 1..2.
% Eine Lösung gefunden
$\not\leftarrow$ [A,B,C] ins 1..2, A #\= B, B #\= C, C #\= A, labeling([],[A,B,C]).

(Unifikation würde sofort scheitern).

Scheduling

```
    A2
   +|
  / |
A1  #
 \  |
  + |
   A3 --+ A5 --+ A6
          +
         /
       A4
```

```
tasks([A1,A2,A3,A4,A5,A6]) ←
    A1 #< A2,
    A1 #< A3,
    A4 #< A5,
    A3 #< A5,
    A5 #< A6,
    A2 #\= A3.
```

```
← tasks(Tasks), Tasks ins 1..4.
% Tasks = [1,_A,2,_B,3,4], _A in 3..4, _B in 1..2.
% Eine Lösung gefunden
← tasks(Tasks), Tasks ins 1..4, labeling([], Tasks).
% Tasks = [1,3,2,1,3,4].
% Tasks = [1,3,2,2,3,4].
% Tasks = [1,4,2,1,3,4].
% Tasks = [1,4,2,2,3,4].
% 4 Lösungen gefunden
```

```
← tasks(Tasks), Tasks ins 1..5.
% Tasks = [_A,_B,_C,_D,_E,_F], _B in 2..5, _A in 1..2, _C in 2..3, _E in 3..4, _D in 1..3, _F in 4..5.
% Eine Lösung gefunden

← tasks(Tasks), Tasks ins 1..5, labeling([], Tasks).
% Tasks = [1,2,3,1,4,5].
% Tasks = [1,2,3,2,4,5].
% Tasks = [1,2,3,3,4,5].
% ...
% 36 Lösungen gefunden
```

```
jobs_kosten(Aufgaben,Kosten) ←
    Aufgaben = [M1,M2,M3],
    Aufgaben ins 1..5,
    all_different(Aufgaben),
    element(M1, [3,2,6,8,9], K1),
    element(M2, [4,6,2,3,2], K2),
    element(M3, [6,3,2,5,2], K3),
    K1+K2+K3 #= Kosten.

← jobs_kosten(Aufgaben,Kosten), Kosten #< 8.
% Aufgaben = [_A,_B,_C], Kosten in 6..7, _A in(1..2)∨(5..5), _B in 2..5, _C in(1..3)∨(5..5).
% Eine Lösung gefunden
```