Compiler construction lecture 14

Logic programs

Chapter 8

Logic programs

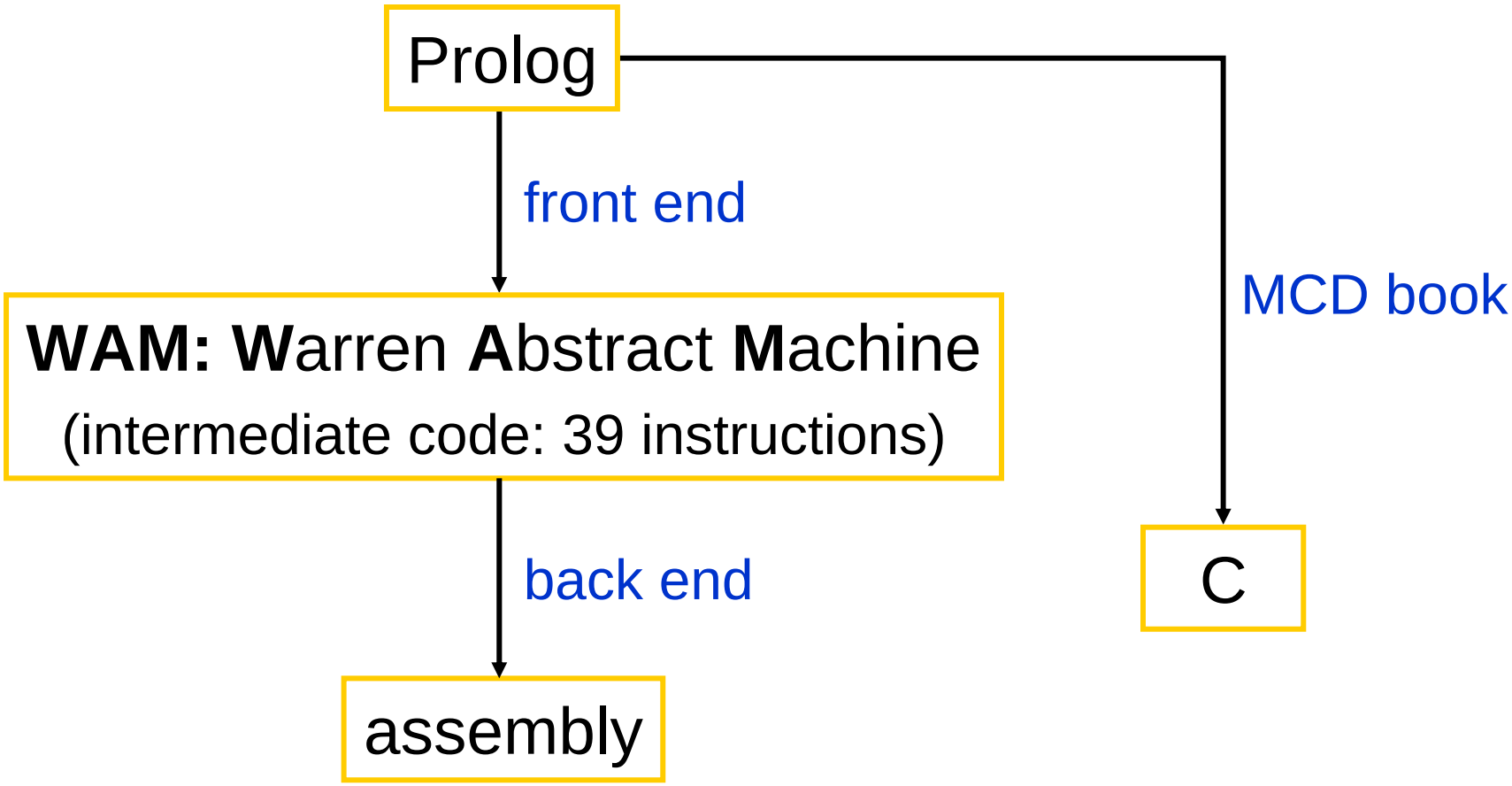
- languages:

  Prolog, Mercury, Gödel

- features:
  - high abstraction level: relations
  - computation by querying
  - unification, backtracking

compiler must work harder!!

Compiler structure for logic programming



Prolog – a short intro

Relations

- facts
- rules

```
% parent/2
parent(dick,koen).
parent(lia,koen).

parent(koen,allard).
parent(koen,linde).
parent(koen,merel).
```

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

Queries

- verify facts
- infer facts

```
?- grandparent(lia,X).
X=allard;

X=linde;

X=merel;

no
```

Prolog – terminology

Relations

- facts
- rules

} clauses

Queries

- `?- grandparent(lia,X).`

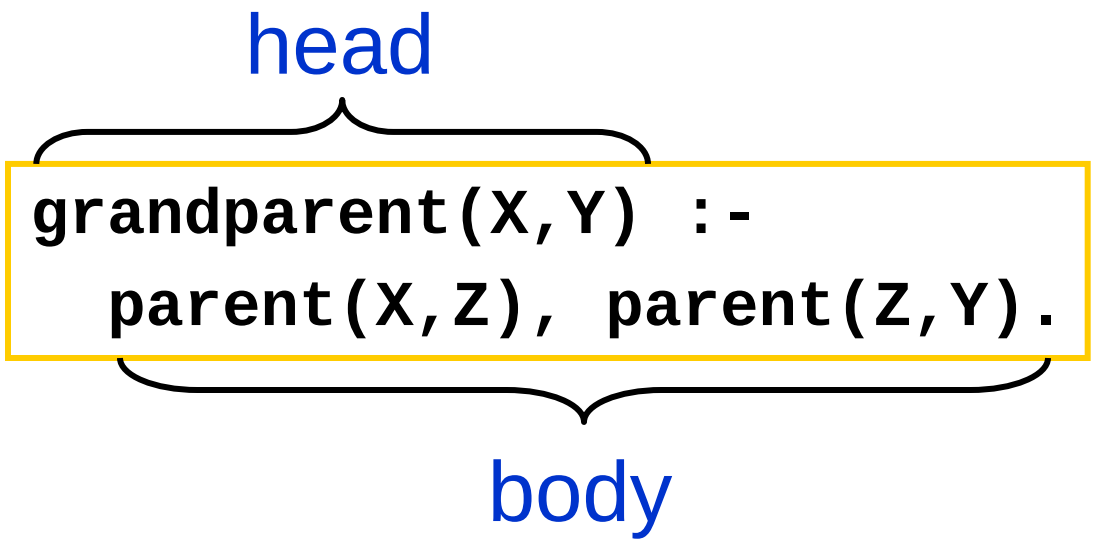
goal

unbound variable

Prolog – terminology

Relations

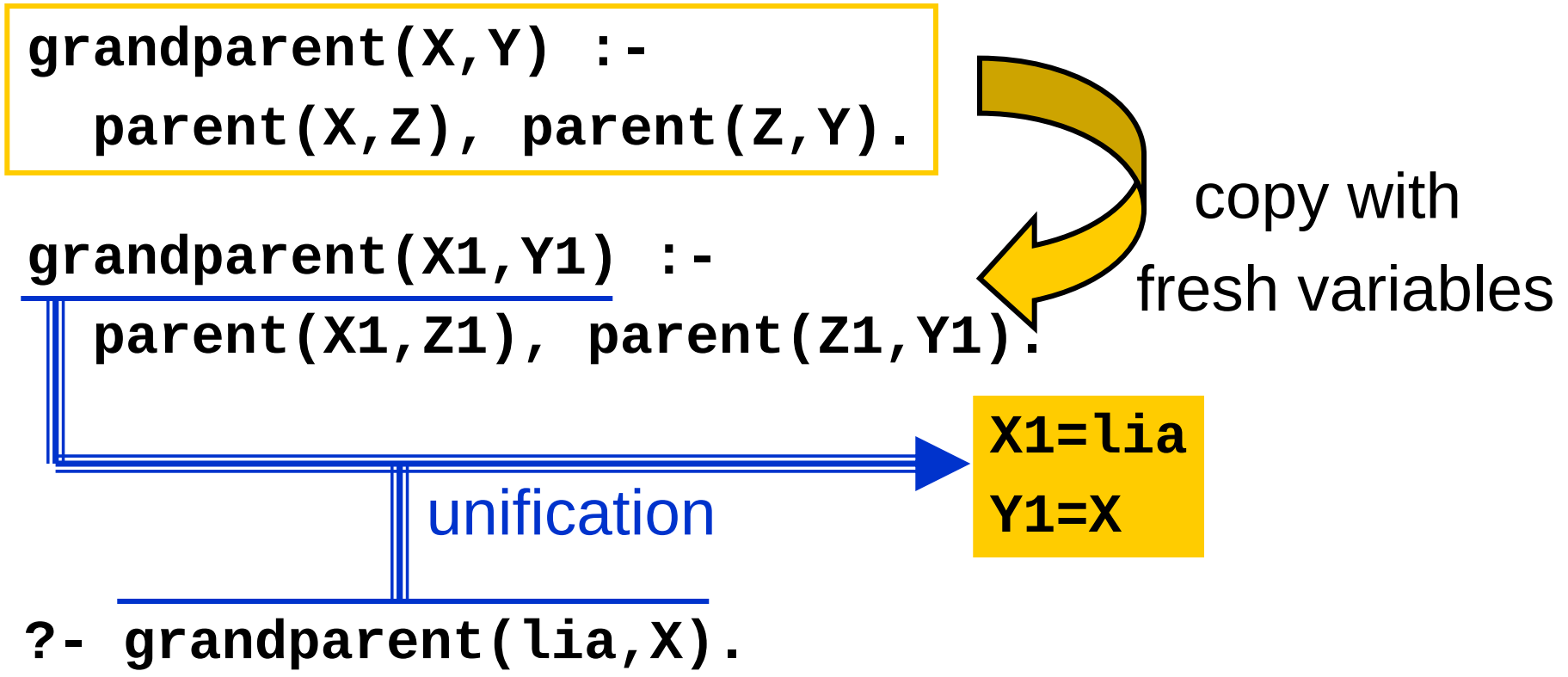
- facts
- rules



Queries

- **?- grandparent(lia,X).**

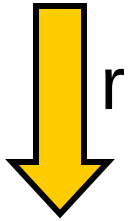
The inference mechanism



The inference mechanism

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

```
grandparent(lia,X) :-
  parent(lia,Z1), parent(Z1,X).
```



replace goal

```
?- parent(lia,Z1), parent(Z1,X).
```

The inference mechanism

```
% parent/2
parent(dick,koen).
parent(lia,koen).

parent(koen,allard).
parent(koen,linde).
parent(koen,merel).
```

unification

Z1=koen

```
?- parent(lia,Z1), parent(Z1,X).
```

The inference mechanism

```
% parent/2
parent(dick,koen).
parent(lia,koen).

parent(koen,allard).
parent(koen,linde).
parent(koen,merel).
```

```
?- parent(lia,koen), parent(koen,X).
X=allard;
X=linde;
X=merel;
```

Exercise (5 min.)

Specify Prolog rules for the binary relations

- **`offspring`** (nakomeling)
- **`spouse`** (echtgenoot)

using the **`parent`** relation.

The bait hides the hook

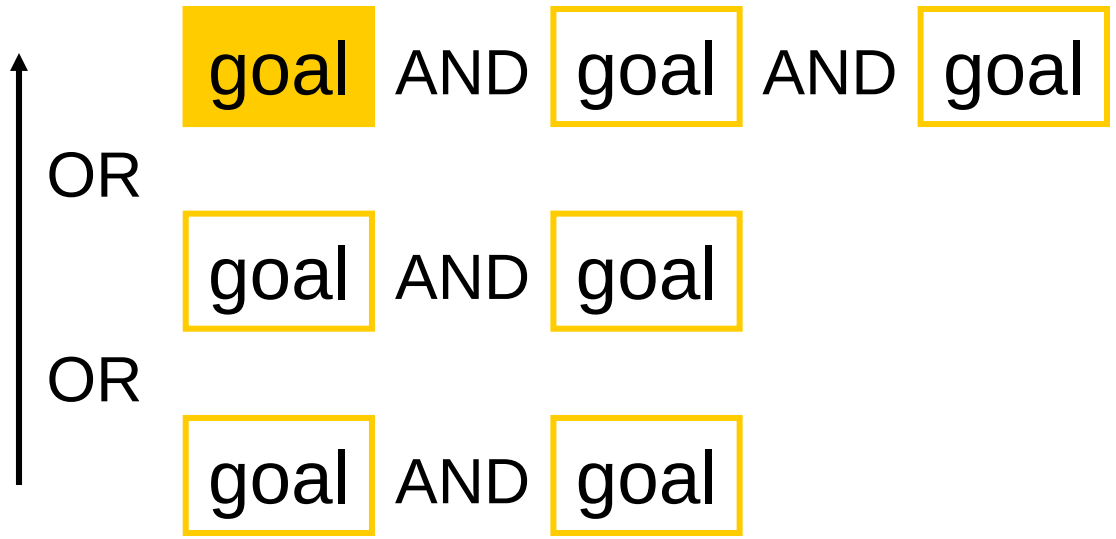
```
parent(dick,koen).
parent(lia,koen).

parent(koen,allard).
parent(koen,linde).
parent(koen,merel).
```

Answers

Interpreting logic programs

- goal list stack

  interpreter operates on the topmost goal list



Interpreting logic programs

- goal list stack

  interpreter operates on the topmost goal list

- display goal

  `<<(“X”,X)`

- unify request

  `[ P ?= Q ]`

- unify result

  `[ P == Q ]`

```
[gp(lia,X) == gp(lia,X)],
     pa(lia,Z1), pa(Z1,X),
                 <<(“X”,X).
```

Interpreter instructions

Switch on topmost leftmost goal

- **Unify**

  `[P ?= Q]` stack all possible unifications

- **Match**

  `[P == Q]` remove goal

- **Succeed**

  `<<("X",X)` print + pop

- **Attach clauses**

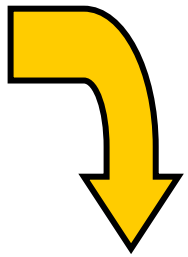
  `default` stack unifications for all clauses

Attaching clauses

- For each clause C, create a copy with fresh variables and stack a unification request with a copy of first goal T

clause head **:-** clause body

first goal, other goals.



[first goal **?=** clause head**]**, clause body, other goals.

Example

```
pa(dick,koen)
pa(lia,koen)
pa(koen,allard)
gp(X,Y) :- pa(X,Z), pa(Z,Y)
```

Interpreter actions

```
gp(lia,X),<<(“X”,X)
```

Example

```
pa(dick,koen)
pa(lia,koen)
pa(koen,allard)
gp(X,Y) :- pa(X,Z), pa(Z,Y)
```

Interpreter actions

- attach

```
[gp(lia,X) ?= pa(dick,koen)],<<(“X”,X)
[gp(lia,X) ?= pa(lia,koen)],<<(“X”,X)
[gp(lia,X) ?= pa(koen,allard)],<<(“X”,X)
[gp(lia,X) ?= gp(X1,Y1)],
pa(X1,Z1),pa(Z1,Y1),<<(“X”,X)
```

Example

```
pa(dick,koen)
pa(lia,koen)
pa(koen,allard)
gp(X,Y) :- pa(X,Z), pa(Z,Y)
```

Interpreter actions

- attach
- unify (4x)

```
[gp(lia,X) == gp(lia,X)],
   pa(lia,Z1),pa(Z1,X),<<("X",X)
```

Example

```
pa(dick,koen)
pa(lia,koen)
pa(koen,allard)
gp(X,Y) :- pa(X,Z), pa(Z,Y)
```

Interpreter actions

- attach
- unify (4x)
- match

```
pa(lia,Z1),pa(Z1,X),<<(“X”,X)
```

Example

```
pa(dick,koen)
pa(lia,koen)
pa(koen,allard)
gp(X,Y) :- pa(X,Z), pa(Z,Y)
```

Interpreter actions

- …
- attach

```
[pa(lia,Z1)?= pa(dick,koen)],pa(Z1,X),<<(“X”,X)
 [pa(lia,Z1)?= pa(lia,koen)],pa(Z1,X),<<(“X”,X)
[pa(lia,Z1)?= pa(koen,allard)],pa(Z1,X),<<(“X”,X)
      [pa(lia,Z1) ?= gp(X2,Y2)],
         pa(X2,Z2),pa(Z2,Y2),pa(Z1,X),<<(“X”,X)
```

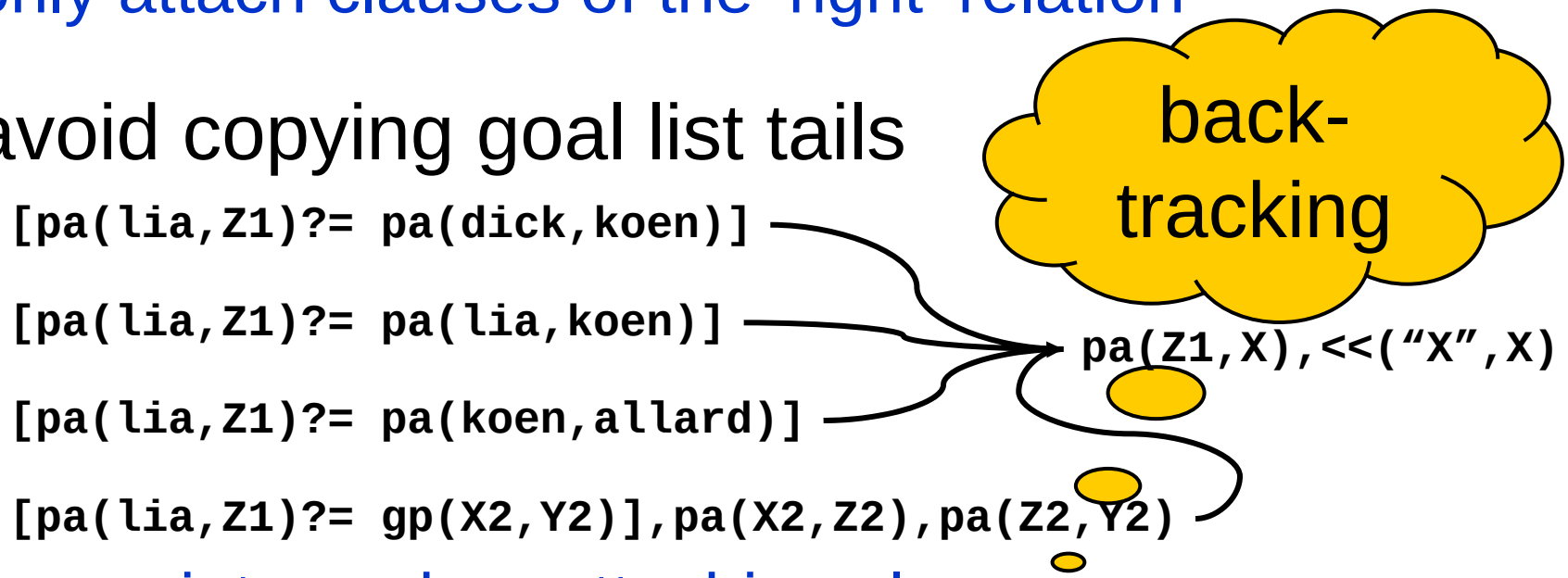
Optimizations

- avoid redundant goal lists

```
[gp(lia,X) ?= pa(koen,allard)],<<("X",X)
```

only attach clauses of the 'right' relation

- avoid copying goal list tails



use pointers when attaching clauses

Unification

dereference variables first

- two constants [success|fail]
- constant and (unbound) variable [success]

  the variable is bound to the constant

- two (unbound) variables [success]

  the head variable is bound to the goal variable

- lists, structures, and sets

  pair-wise unification of terms

Unification – implementation

```
struct variable {
        char *name;
        struct term *term;
};
```

```
struct list {
        struct term *hd;
        struct term *tl;
};
```

```
typedef enum {Is_Const,Is_Var,Is_List} TermType;
typedef struct term {
        TermType type;
        union {
                char *const;              /* Is_Const */
                struct variable var;      /* Is_Var */
                struct list list;         /* Is_List */
        } term;
} Term;
```

```
Term *deref(Term *t) {
   while (t->type == Is_Var && t->term.var.term != 0) {
      t = t->term.var.term;
   }
   return t;
}
```

```
int unify_terms(Term *goal_arg, Term *head_arg) {
   Term *goal = deref(goal_arg);
   Term *head = deref(head_arg);

   if (goal->type == Is_Var || head->type == Is_Var) {
      return unify_unbound_variable( goal, head);
   } else {
      return unify_non_variables( goal, head);
   }
}
```

Unification – implementation

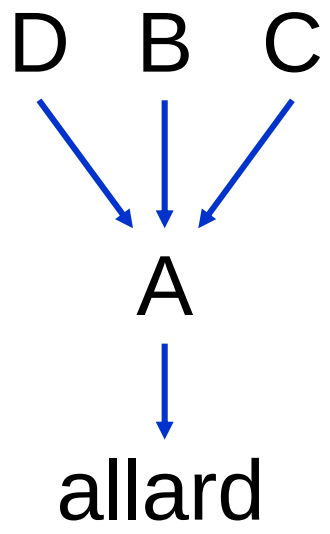
```
int unify_non_variables(Term *goal, Term *head) {
   if (goal->type != head->type) return 0;

   switch (goal->type) {
     case Is_Const:
       return strcmp(goal->term.const,
                     head->term.const) == 0;
     case Is_List:
       return unify_terms(goal->term.list.hd,
                          head->term.list.hd)
           && unify_terms(goal->term.list.tl,
                          head->term.list.tl);
   }
}
```

Unification – implementation

```
int unify_unbound_variable(Term *goal, Term *head) {
   if (goal == head) return 1;     /* identical vars */

   if (head->type == Is_Var) {
      trail_binding(head);
      head->term.var.term = goal;
   } else {
      trail_binding(goal);
      goal->term.var.term = head;
   }
   return 1;
}
```

(un)binding variables

- unify_terms(A, B)
- unify_terms(B, C)
- unify_terms(B, D)
- unify_terms(allard, A)



- undoing the most recent binding (backtracking) is easy

zero out the pointer in the variable

Compiling Prolog

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

- generate C

```
int grandparent(Term *X, Term *Y) {
    Term *Z = new_Var("Z");

    return parent(X,Z) && parent(Z,Y);
}
```

we must backtrack!

Backtracking in C: list procedures

Move the action into the procedure
"returning" a list of values

```
void two_values(void (*action)(int)) {
   (*action)(11);
   (*action)(13);
}
...
two_values(foo);
```

Compiling Prolog – 2nd try

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

- generate C

```
void gp(Term *X, Term *Y, Action goal_tail_list) {
   Term *Z = new_Var("Z");

   void gp_goal2(void) {
      pa(Z,Y,goal_tail_list);
   }

   pa(X,Z,gp_goal2);
}
```

Compiling Prolog – facts

```
parent(koen,allard).
```

```
void pa(Term *arg1, Term *arg2, Action goal_tail_list){
   Term *koen = new_Const(”koen”);
   Term *allard = new_Const(”allard”);

   void pa_arg2(void) {
      unify_terms(arg2,allard,goal_tail_list);
   }

   unify_terms(arg1,koen,pa_arg2);
}
```

Unification + backtracking

```
void unify_unbound_variable(
        Term *goal, Term *head, Action goal_tail_list) {
   if (goal == head) {                  /* identical vars */
      goal_list_tail();
   } else if (head->type == Is_Var) {
      head->term.var.term = goal;
      goal_list_tail();
      head->term.var.term = 0;          /* undo binding */
   } else {
      goal->term.var.term = head;
      goal_list_tail();
      goal->term.var.term = 0;          /* undo binding */
   }
}
```

Multiple clauses

```
parent(koen,allard).
parent(koen,linde).
parent(koen,merel).
```

```
void pa(Term *arg1, Term *arg2, Action goal_tail_list){
   void pa_clause1(void){ ... }
   void pa_clause2(void){ ... }
   void pa_clause3(void){ ... }

   pa_clause1();
   pa_clause2();
   pa_clause3();
}
```

Optimized clause selection in the WAM

```
parent(dick,koen).
parent(lia,koen).

parent(koen,allard).
parent(koen,linde).
parent(koen,merel).
```

```
parent(lia,X)
```

Take advantage of constants

- inside clauses
- passed as arguments

Optimizations on 1st argument

- inline expansion of **`unify_terms()`**
- lift type checking part common to all clauses
- for each constant: call matching clauses only

Optimized clause selection in the WAM

```
parent(dick,koen).
parent(lia,koen).

parent(koen,allard).
parent(koen,linde).
parent(koen,merel).
```

```
parent(lia,X)
```

```
L_constant:
switch (hash(arg1->term.const)) {
  case DICK:
    pa_clause1();
    goto L_fail;
  case LIA:
    pa_clause2();
    goto L_fail;
  case KOEN:
    pa_clause3();
    pa_clause4();
    pa_clause5();
    goto L_fail;
}
```

```
void pa_clause2(void) {
  Term *koen = new_Const(”koen”);

  unify_terms(arg2, koen,
              goal_tail_list);
}
```

Implementing the ‘cut’

```
spouse(X,Y) :-
   parent(X,Z),
   parent(Y,Z),
   X \= Y,
   !.
```

backtracking stops after reporting first couple

```
void spouse(Term *X, Term *Y,
            Action goal_tail_lst) {
   Term *Z = new_Var(”Z”);
   void spouse_goal3(void) {
      if (!equal(X,Y)) {
         goal_tail_lst();
         goto L_cut;
      }
   }
   void spouse_goal2(void) {
      pa(Y,Z,spouse_goal3);
   }
   pa(X,Z,spouse_goal2);
L_cut:;
}
```

Advanced stuff

- implementing **`assert`** and **`retract`** [8.4.5]
- compiled unification [8.5.1 – 8.5.3]
- read/write mode optimization [8.5.4]

Summary

Logic programming = inferring facts

- bind logic variables (unification)
- backtracking (vs. copying)

Implementation

- interpreter (attach, unify, match, succeed)
- compiler (Prolog -> C)
  - backtracking through list procedures
  - nested routines