Teaching Prolog and CLP

Ulrich Neumerkel

Institut für Computersprachen
Technische Universität Wien
www.complang.tuwien.ac.at/ulrich/

I The "magic" of Prolog — Common obstacles

II Reading of programs

III Course implementation

Part I
Common obstacles

- The “magic” of Prolog
  - puzzling procedural behaviour
  - voracious systems
- Previous skills and habits
- Prolog’s syntax
- Naming of predicates and variables
- List differences

Syllabus

- Training (project oriented) vs. teaching (concept oriented)

  — Larger projects do not work well

- full Prolog vs. pure Prolog

  — pure Prolog + CLP(FD)

Basics:

- Basic reading skills for understanding Prolog programs
- Avoiding common mistakes, develop coding style

Previous skills to build on

- Programming skills
- Mathematical skills
- Language skills

Previous programming skills

- Bad programming habits — the self-taught programmer
  Severe handicap: Edit-Compile-Run-Dump-Debug
  "Let the debugger explain what the program is doing"
- assertions? invariants? test cases before coding?
  (Eiffel, but also C `<assert.h>`).

Mathematical skills

- mathematical logic as prerequisite
- calculational skills (e.g. manipulating formulæ)
- syntactic unification (equational, Martelli/Montanari)

Language skills

- Many difficulties of Prolog are clarified reading programs in plain English.
- E.g. quantification problems in negation:

  female(Female) $\leftarrow$
      \+ male(Female).

  Everything/everyone, **really** *everything/everyone that/who is not male is female. Therefore: Since a chair/hammer/the summer isn't male it's female*

- Detect defaulty data structure definitions

  is_tree(_Element).   % *Everything is a tree.*
  is_tree(node(L, R)) $\leftarrow$
      is_tree(L),
      is_tree(R).

Prolog's Syntax, Difficulties

Minor typos make a student resort to bad habits

Prolog's syntax is not robust: "male(john)." is a goal or fact.

father_of(Father, Child) ←
    child_of(Child, Father),
    male(Father), % !
male(john).

1. Redesign Prolog's syntax. (Prolog II)
2. Subset of existing syntax. Spacing and indentation significant (GUPU).
   (a) Each head, goal in a single line.
   (b) Goals are indented. Heads are not indented.
   (c) Only comma can separate goals (i.e. *no* disjunction)
   (d) Different predicates are separated by blank lines.

⇒ more helpful error messages possible

Names of predicates

key to understanding — assignments for finding good names

Misnomers

- action/command oriented prescriptive names: append/3, reverse/2, sort/2
  quick fix: use past participle, sometimes noun
- leave the argument order open: child/2, length/2
- pretend too general or too specific relation: reverse/2, length/2
- tell the obvious: body_list//1

Finding predicate names

1. Start with intended types: type1_type2_type_3(Arg1, Arg2, Arg3)

   "child of a person" : person_person/2

2. If name too general, refine

   person_person ⇒ child_person/2

   list_list/2 ⇒ list_reversedlist/2

3. Emphasize relation *between* arguments

   - shortcuts like prepositions: child_of/2
   - past participles alone: list_reversed/2

"length of a list": list_number/2 ⇒ list_length/2
"append": list_list_list/3⇒ list_list_appended/3⇒ list_listdiff(X,Z,Y)⇒ list//1
"sorting": list_list/2 ⇒ list_sortedlist/2 ⇒ list_sorted/2⇒ list_ascending/2

Problem: High arities yield long names

- try to avoid high arities: DCGs, group arguments in meaningful structures, e.g. Latitute, Longitude ⇒ Position
- omit less important arguments *at the end*, name ends with underscore: country_(Country, Region, Population, ...)
- put the less important arguments at the end

Type definitions

Convention: is_type(Type) or type(Type)

- documentation purpose
- serve as template for predicates defined over data structures

O'Keefe-rules

- unsuitable (for beginners)
- deal with procedural aspects
- inputs and outputs: atom_chars vs. atom_to_chars

Variable names

Lack of type system makes consistent naming essential

- for lists: [Singularform|Pluralform] , e.g. [X|Xs]
- naming void variables in the head: member(X,[X|_]) _Xs instead of _
- state numbering (e.g. list differences) instead of Xin, Xout, Xmiddle

Understanding differences

– misleading name: “difference list”
  instead : difference, list difference, difference of lists

– differences too early

+ use grammars first: less error-prone, powerful, compact (string notation)

– differences presented as incomplete data structures — “holes”

+ motivate differences with ground lists

+ differences are not specific to lists, describe state

Part II
Reading of programs

$$Algorithm = Logic + Control$$

Family of related reading techniques

Focus on distinct (abstract) parts/properties of the program

- informal reading in English
- declarative reading
- (almost) procedural reading
- termination reading
- resource consumption

Informal reading

use English to

- focus the student's attention on the meaning of program
- avoid operational details
- clarify notions
- clarify language ambiguities
- clarify confusion of "and" and "or"

ancestor_of(Ancestor, Person) ←
    child_of(Person, Ancestor).

Someone is an ancestor of a person if he is the parent of that person.
Alternatively: *Parents are ancestors.*

ancestor_of(Ancestor, Descendant) ←
    child_of(Person, Ancestor),
    ancestor_of(Person, Descendant).

Someone is an ancestor of a descendant if he is the parent of another ancestor of the descendant.

Alternatively: *Parents of ancestors are ancestors*

Reading complete predicates is often too clumsy:

Someone is an ancestor of a descendant, (either) if he is the parent of that descendant, **or** *if he is the parent of another ancestor of the descendant.* (unspeakable)

Alternatively: *Parents* **and** *their ancestors are ancestors*. (too terse)

Informal reading is intuitive but limited to small programs.
⇒ Extend informal reading to read larger programs

Declarative reading of programs

- consider only parts of program at a time
- cover the uninteresting/difficult parts (~~like this~~)
- shortens sentences to be read aloud

Conclusion reading

Read clause in the direction of the rule-arrow (body to head).

Analysis of clauses

Read single clause at a time. Add remark: *But there may be something else.*

ancestor_of(Ancestor, Person) ←
  child_of(Person, Ancestor).

~~ancestor_of(Ancestor, Descendant)~~ ←
  ~~child_of(Person, Ancestor)~~,
  ~~ancestor_of(Person, Descendant)~~.

Someone is an ancestor of a person if he is the parent of that person. (But there may be other ancestors as well).

Alternatively: *At least parents are ancestors.*

~~ancestor_of(Ancestor, Person)~~ ←
    ~~child_of(Person, Ancestor)~~.
ancestor_of(Ancestor, Descendant) ←
    child_of(Person, Ancestor),
    ancestor_of(Person, Descendant).

Someone is an ancestor of a descendant if he's the parent of another person being an ancestor of the descendant. But ...

At least parents of ancestors are ancestors.

Erroneous clauses

For error location it is not necessary to see the whole program

ancestor_of_too_general(Ancestor, Person) ←
    child_of_too_general(Ancestor, Person).
~~ancestor_of_too_general(Ancestor, Descendant)~~ ←
    ~~child_of_too_general(Person, Ancestor)~~,
    ~~ancestor_of_too_general(Person, Descendant)~~.

Analysis of the rule body

- goals restrict set of solution
- cover goals to see generalized definitions

father(Father) ←
    male(Father),
    ~~child_of(_Child, Father)~~.

Fathers are at least male. (But not all males are necessarily fathers)

father_toorestricted(franz) ←
    ~~male(franz)~~, % Body is irrelevant to see that definition is too restricted.
    ~~child_of(_Child, franz)~~.

Searching for errors

If erroneous definition is

1. too general. Use: Analysis of clauses to search too general clause
2. too restricted. Use: Analysis of the rule body

Reading method leads to analgous writing style.

Procedural reading of programs

- special case of the declarative reading
- uncover goals in strict order
- look at variable dependence
  - first occurrence of variable — variable will always be free
  - further occurrence — connected to goal/head

1. ancestor_of(Ancestor, Descendant) $\leftarrow$ % $\Longleftarrow$ head never fails
   ~~child_of(Person, Ancestor)~~,
   ~~ancestor_of(Person, Descendant)~~.

2. ancestor_of(Ancestor, Descendant) $\leftarrow$
   child_of(Person, Ancestor), % $\Longleftarrow$
   ~~ancestor_of(Person, Descendant)~~.

$\Rightarrow$ Ancestor can influence child_of/2. Descendant doesn't.
Person will be always free. Descendant only influences ancestor_of/2.

Termination

- often considered weak point of Prolog
- nontermination is a property of
  a general purpose programming language
- only simpler computational models guarantee termination
- floundering is also difficult to reason about
- pretext to stop declarative thinking, usage of debuggers etc.
- difficult to understand by looking at Prolog's precise execution (tracing)

Notions of Termination

T1: $\leftarrow$ Goal1. terminates
T2: $\leftarrow$ Goal2. terminates
T3: $\leftarrow$ Goal1, Goal2. terminates

Existential termination: $\leftarrow$ Goal. finds an answer substitution

Difficult to use / analyze:

- clause order significant
- T1 and T2 $\not\Rightarrow$ T3 (loops "on backtracking")
- T3 $\Rightarrow$ T1

Universal termination: $\leftarrow$ Goal. terminates iff $\leftarrow$ Goal, false. finitely fails

Easier to analyze:

- clause order not significant
- T1 and T2 $\Rightarrow$ T3 (no surprise on backtracking)
- T3 $\Rightarrow$ T1

Properties of universal termination

1. Adding clause does not affect nonterminating goals.

   $\leftarrow$ Goal. nonterm. for P $\Rightarrow$ $\leftarrow$ Goal. nonterm. for P $\cup$ {C}

2. For many interesting programs P (e.g. binary clauses and facts):

   $\leftarrow$ Goal. nonterm. for P $\Leftrightarrow$ $\leftarrow$ Goal. nonterm. for P $\cup$ {C}, C is a fact

Methods for termination reading

- reading a predicate:

  hide clauses, if simpler predicate does not terminate, also the original predicate does not terminate (by 1)

- reading single clause:

  H $\leftarrow$ G1, ..., Gi, false. nonterm. $\Rightarrow$ H $\leftarrow$ G1, ..., Gi, ...,Gn. nonterm.

Termination reading is very fast in location possibilities for nontermination.
Unfortunately (in most cases) no replacement for termination proof.

Example termination reading: append/3

- cover some (irrelevant) clauses: esp. facts, non recursive parts

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

  - reduced predicates terminates iff original terminates
  - **The** misunderstanding of append/3
    rôle of fact append([], Xs, Xs)
    often called "end/termination condition"
    But: append([], Xs, Xs) has no influence on termination!

- cover variables handed through (Ys): ~~append([], Xs, Xs)~~.
  append([X|Xs], ~~Ys,~~ [X|Zs]) ←
      append(Xs, ~~Ys,~~ Zs).

- cover head variables (approximation): ~~append([], Xs, Xs)~~.

  append([$\cancel{X}$ |Xs], ~~Ys,~~ [$\cancel{X}$ |Zs]) ←
      append(Xs, ~~Ys,~~ Zs).

Resulting predicate:
appendtorso([_X|Xs], [_Z|Zs]) ←
    appendtorso(Xs, Zs).

- if appendtorso/2 terminates, append/3 will terminate
- appendtorso/2 never succeeds
- only a safe approximation
  ← append([1|_], _, [2|_]).
  ← appendtorso([1|_], [2|_]).
  appendtorso/2 does not terminate while append/3 does

Example termination reading: append3/4

$$\begin{array}{ll} \text{append3A(As, Bs, Cs, Ds)} \leftarrow & \text{append3B(As, Bs, Cs, Ds)} \leftarrow \\ \quad \text{append(As, Bs, ABs),} & \quad \text{append(ABs, Cs, Ds),} \\ \quad \text{append(ABs, Cs, Ds).} & \quad \text{append(As, Bs, ABs).} \end{array}$$

append3A(As, Bs, Cs, Ds) ←
    append(As, Bs, ABs), % ⟸ terminates only if As is known
    ~~append(ABs, Cs, Ds)~~.
similarily append3B/4: terminates only if Ds is known

- only a part of predicate was read — second goal was not read
- it was not necessary to imagine Prolog's precise execution
- no "magic" of backtracking, unifying etc.
- a tracer/debugger would show irrelevant inferences of second goal
- solution:

Fair enumeration of infinite sequences

- termination reading is about termination/non-termination only
- in case of non-termination, fair enumeration still possible
- much more complex in general
- order of clauses significant
- e.g. unfair if two independent infinite sequences

  list_list(Xs, Ys) ←
      length(Xs, _),
      length(Ys, _).
- explicit reasoning about alternatives (backtracking)
- use *one* simple fair predicate (e.g. *one* length/2) instead
- learn the limits, but don't go to them

Resource consumption

- analytical vs. empirical
- *Do not try to understand precise execution!*
- prefer measuring over tracing
- abstract measures often sufficient
  - inference counting: similar to termination reading

    list_double(Xs, XsXs) $\leftarrow$
        append(Xs, Xs, XsXs).
    $\leftarrow$ length(XsXs, N), list_double(Xs, XsXs).

    list_double(Xs, XsXs) $\leftarrow$
        append(Xs, Ys, XsXs),
        Xs = Ys.
    $\leftarrow$ list_double(Xs, XsXs).
  - size of data structures: approx. proportional to execution speed

Reading of definite clause grammars

```
nounphrase ⟶        % A noun phrase consists of
    determiner,     % a determiner followed by
    noun,           % a noun followed by
    optrel.         % an optional relative clause.
```

Declarative reading of grammars

```
nounphrase ⟶        % A noun phrase (at least)
    determiner,     % starts with a determiner
    noun,           % —
    optrel.         % ends with an optional relative clause
```

Procedural reading of grammars

Take implicit argument (list) into account

```
list([]) ⟶              list_(Xs, Ys, Zs) ⟶      append3(As, Bs, Cs, Ds) ←
   [].                     list(Xs),                 phrase(list_(As, Bs, Cs), Ds).
list([X|Xs]) ⟶             list(Ys),
   [X],                    list(Zs).
   list(Xs).
```

Writing of programs

1. find types (is_-predicates)
2. find relations and good names
3. write down example goals that should suceed/fail/terminate
4. define the actual predicate

CLP(FD)

- map problem into integers
- difficult to test

Structure of CLP(FD) programs

1. domains with domain_zs(Min..Max,Zs)
2. relations
3. additional constraints (redundant, reducing symmetries)
4. labeling labeling_zs(Labelingmethods,Zs)

- define a *single* predicate for 1-3 e.g. krel_vars(Desc, Vars)
- always separate labeling completely

```
rel(Desc) ←
    krel_vars(Desc, Zs),
    labeling_zs([ff], Zs).
```

- frequent error: early labeling

```
list_sum([E|Es],S0) ←
    S0 #= S1 + E,
    E in 1..10, % !
    labeling_zs([], [E]), % !
    list_sum(Es,S1).
```

- frequent error: not all variables are labelled, display constraint store

Termination in CLP programs

- complex programs are difficult to test: labeling takes a lot of time

```
↚ rel(Desc), false. % often too expensive
↚ krel_vars(Desc,_), false. % faster termination test
```

- goal reordering:

```
n_factorial(0,1).
n_factorial(N0, F0) ←
    N0 #>= 1, N0 #= N1 + 1, n_factorial(N1, F1),
    F0 #= N0 * F1. % !
```

Part III
Course implementation

- 2nd year one semester course, 2hrs/week (effectively: 9×5hrs work)
- nine weeks (example groups) about 80 small assignments

Course contents

- Basic elements (queries, facts, rules) and declarative reading
- Procedural reading, termination reading
- Terms, term arithmetic, lists
- Grammars
- CLP(FD)
- List differences (*after grammars*), general differences

Cursory at end: meta-logical & control (error/1, var/1, nonvar/1, cut), negation, term analysis, is/2-arithmetic

Topics not covered

(*): covered in an advanced course (3hrs)

1. setof(Template, Goal, Solutions) (*)
   "answer substitutions" vs. "list of solutions" confusing — quantification tricky
2. meta interpreters (*) — *program* = *data* too confusing, defaultyness of vanilla
   instead use pure meta interpreters "in disguise" (e.g. regular expressions)
3. meta call (*)
4. explicit disjunction (*) — meaning of alternative clauses must be understood first
5. if then else (*) — leads to defaulty programming style
   if used, restrict condition to var/nonvar and arithmetical comparison
6. data base manipulation (*) — difficult to test — if used, focus on setof/3-like usage
7. advanced control (*) — reasoning about floundering difficult
8. extra logical predicates
9. debuggers, tracers — reason for heavy usage of cuts

GUPU Programming Environment

Gesprächs**u**nterstützende **P**rogrammierübungs**u**mgebung

conversation supporting programming course environment

- specialized for Prolog courses
- uses clean subset of Prolog, no side effects
- comfortable querying and testing
- viewers for graphical display of answer substitutions

Further information

- Guided tour: http://www.complang.tuwien.ac.at/ulrich/gupu/
- Demo Friday 9h00 at the

    8th Workshop on Logic Programming Environments