Prolog as description and implementation language in computer science teaching

paper by Henning Christiansen
presented by John Gallagher
Outline of the talk

• The problem of teaching theoretical C.S. in an interdisciplinary context
• Relevant properties of Prolog
• Conceptual model: abstract machines, plus Prolog for abstract syntax, semantics, compilation, etc.
• Example: What the teacher can squeeze out of an interpreter for machine language written in Prolog
• Other applications of the principles
• A sample course schedule
• Experiences & conclusion
The task: Teach C.S. students about

- Programming languages [basic theoretical understanding]
- Theoretical understanding of computing machines and computer languages
- (Idea of) precise semantics, 1–1 expressions–to–meanings relation
- Which `expressions`, which `meanings`?
- Interpreters, compilers, etc. as practical programming techniques
Conditions for teaching C.S. at Roskilde Univ.

• Interdisciplinary studies
  • Natural Science, Humanities, Social Science students, all in the same lecture rooms & exercise sessions
  • Mathematical backgrounds: From almost nothing to excellent

• 50% student project works from day 1
  • Few nominal course hours for "core material"
  • Students competent learners

• Teacher's considerations to the extreme
  • Address all students, inspire all students
  • Maintain substance, avoid "superficial populism"
  • Selection of protypical topics & examples very critical
Need for pedagogical framework & tools

• Overall theoretical model
• Precise (formal) description language
• Dynamic models: test/exercise/experiment

Our choice

• Simple model of abstract machines
• Prolog as
  • Description language
  • Experimentation tool ("dynamic model")
  • Example programming language ≠ Java
Why Prolog?

Familiar properties that we can rely on

- Core language has equiv. declarative and procedural semantics
- Used in disciplined way, reasonable semantics preserved for larger subset
- Easy to learn: students can write interesting programs after one day
- Compare with Java to show that prog. lang. can be elegant and highly expressive
  (however, for the price of loosing robustness and security,...)

More specifically for our goals...
Prolog terms as repr. of abstr. syntax trees

Emphasize structure *and* textually pleasing:

\[
a := 221; \ b := 493;
\]
\[
while(a =\ not= b, \ if(a>b, \ a := a-b, \ b := b-a))
\]
works provided :- op(..., ..., :=)

Structurally inductive definitions straightfw'd:

\[
stmnt(\text{while}(C, S), \ ...) :-
\]
\[
\text{condition}(C, \ ...), \ stmnt(S, \ ...), \ ...
\]

Such definitions are

- formal
- executable
- easily accessible with modest experience of Prolog
Other properties of Prolog

• Aux. pred's for symbol tables etc. easy to supply
• Easy to add pragmatic issues to spec's
  • e.g. turn interpreter into tracer or debugger
  • add error messages (instead of just "no")
• Interactive Prolog environment invites to incremental development
  • Type in and test spec. rule by rule during your lecture!
  • Exercises of modifying or extending spec. work well
Abstract machines as super-concept

**Def.**: Abstract machine characterized by

- an *input language* (some set of phrases)
- *memory* which at any time keeps values...
  (no explicit output component, part of "visible" mem.)
- A *semantic function*: Phrase x Mem -> Mem

**Examples:**
machine (language), pocket calculator, general prog.lang., database transaction system, program modules, general user interfaces
Defining interpreters in Prolog

- Abstract syntax given by rules of form
  \[ \text{cat}_0(\text{op}(T_1, \ldots, T_n)) :\text{=} \text{cat}_1(T_1), \ldots, \text{cat}_n(T_n). \]
- Each syn.tree has its own semantic relation
  e.g. State x State.

  \[ x := x + 1 \Rightarrow \{ \ldots, < [x=7], [x=8] >, \ldots, \]
  \[ < [x=117, y=4], [x=118, y=4] >, \ldots \} \]

- Def. interpreter given by rules extending syntax:

  \[ \text{cat}_0(\text{op}(T_1, \ldots, T_n), SRel_0) :\text{=} \]
  \[ \text{cat}_1(T_1, SRel_1), \ldots, \text{cat}_n(T_n, SRel_n), \ldots \text{ compose} \ldots . \]

An example to see how it works in practice...
Def. interp. for simple machine language

Program ≈ sequence of instr. ≈ Prolog list:

\[
[ \text{push}(2), \text{store}(t), \\
7, \text{fetch}(x), \ldots \\
equal, \text{n_jump}(7)]
\]

Defining interpreter consists of rules of form

\[
\text{sequence}([\text{FirstInstr}|\text{Rest}], \text{Prog}, \text{Stack, VarBinds, Stack}_\text{Fin, VarBinds}_\text{Fin}):- \\
\ldots \text{transform} \ldots, \\
\text{sequence}(\text{Contin, Prog, NewStack, NewVarBinds, Stack}_\text{Fin, VarBinds}_\text{Fin}).
\]

Most often \textit{Contin} = \textit{Rest}; argument \textit{Prog} holds the whole program to provide contextual meanings of labels
Def. interp. for simple machine language, \textit{cnst}

Standard instructions, no surprises:

\begin{verbatim}
sequence([add|Cont], Prog, [X,Y|S], B, Sn, Bn):-
    YplusX is Y + X,
    sequence(Cont, Prog, [YplusX|S], B, Sn, Bn).
\end{verbatim}

Jumps: Non-standard continuation

\begin{verbatim}
sequence([jump(E)|_], P, S, B, Sn, Bn):-
    append(_, [E|Cont], P),
    sequence(Cont, P, S, B, Sn, Bn)).
\end{verbatim}
Def. interp. for simple machine language, \textit{cndt}

Standard instructions, no surprises:

\begin{verbatim}
sequence([add|Cont], Prog, [X,Y|S], B, Sn, Bn):-
    YplusX is Y + X,
    sequence(Cont, Prog, [YplusX|S], B, Sn, Bn).
\end{verbatim}

Jumps:

\begin{verbatim}
sequence([jump(E)|_], P, S, B, Sn, Bn):=
    append(_, [E|Cont], P),
    sequence(Cont, P, S, B, Sn, Bn).
\end{verbatim}

A little extra, making interp. into tracer adding this initial rule:

\begin{verbatim}
sequence([Inst|_],_,_,_,_,_,_):=
    write(Inst), write(' '), fail.
\end{verbatim}
Exercises to def. int. for machine lang,

- Extend language and interpreter with instructions for subroutines
- Write Prolog program that checks labels used correctly (and what does this mean?)
- Write optimizer that replace patterns of instructions by others
Time for reflection:

• How long time will it take (lecture+exercises)
• How many important points were made explicit and clear by this minimalist example?
• Was it difficult for the students to follow?
Other notions presented in similar way

• Semantics of While-programs by def. interp.
• Compilation: As above but with semantics expressed by code sequences.
  Example: Compiling While-programs into our machine lang.
• Recursive procedures & type-checking *(details later)*
• Interactive LISP environment modelled with `assert`
• Vanilla self-interpreter for Prolog extended into tracer and debugger
• Turing machines (with transition function as facts)
A successful learning-by-doing approach to recursion and typechecking

• Informal introduction to Pascal-like language; recursive Quicksort as prototypical test program
• Informal explanation to type-requirements, type-checking, and stack-based impl. of recursion
• Students' task in 1 week on 50% time:
  • Write type-checker in Prolog
  • Write defining interpreter and test on Quicksort program
  • Document solution in short report
• All students succeeded in time
• Positive comments from students: Aha experience
• No difference between math and no-math students
Sch'le of 7.5ECTS course, 10 course days + hw

2. Prolog workshop II: Lists, operators, assert/retract, cut, negation-as-failure.
4. Language and meta-language, Prolog as meta-language.
   Semantics of sequential and imperative languages; compilation
5. Declarations, type checking, recursive procedures
6. Do-it-your-self recursive procedures, interpreter and type checker
7. Discussing solution to above. Turing-machines, decidability and computability, T.-universality, Halting problem, TMs in Prolog
8. Extra theme: Constraint Logic Programming, CHR
9. Traditional syntax analysis, FSA and recursive descent
10. Phases of traditional compiler, dissect impl. of Datalog in Java.

Evaluation of the course
Experience

• Lively interaction with students in lectures and exc's
• Equally successful and effective for all students in our heterogeneous audience
• Difficult material becomes accessible for non-math inclined students (usually judged unaccessible)
• Much easier to learn and master than, say, domain theory — but recursive def's in Prolog essentially express the same
• No loss for math. inclined students, in later project work they may read Winskel's book on semantics or A&S&U's book on compiling — very quickly!
Conclusion

• This Prolog-based teaching methodology is highly effective for university students of C.S. (in Roskilde University's special context)

• May also be proposed for
  • Homogeneous audiences with solid math background — as intro to hard-core theoretical CS
  • IT-related univ. edications with otherwise no C.S. theoretical elements
  • Younger students, high school??