

Programming in Intuitionistic Logic

Alwen Tiu

Australian National University

ANU Logic Summer School
Lecture 3, December 12, 2007

Abstract Logic programming

- What is a *logic programming language*? How is it different from *theorem proving*?
- The premise behind logic programming is that the operational aspects of programming constructs coincides with the logical interpretation. For example: in Prolog, when a user query a goal $A \wedge B$, given a program P , he/she expects the interpreter to attempt proving A and B (with the same program P).
- Note that the program P is consulted only when the goal is *atomic*, i.e., no logical connectives appear in it.
- We take the abstract view of logic programming as *goal-directed proof search*.

Programming = Logic + ...

- In early days of logic programming, Bob Kowalski wrote the following equation

$$\text{Algorithm} = \text{Logic} + \text{Control}$$

- This has been greatly elaborated in subsequent years to something like

$$\text{Programming} = \text{Logic} + \text{Control} + \text{I/O} + \text{Modules}$$

$$+ \text{Concurrency} + \text{Object-oriented} + \dots$$

- An important research goal in the specification of programming language is to try to achieve

$$\text{Programming} = \text{Logic}$$

This would require a rethink on both the 'programming' part and the 'logic' part. Not all programming activities require logic. And we shouldn't just stick to classical logic.

Specifying computation as proof search

- In specifying computation in a logic programming language, we specify
 - ▶ a signature Σ containing the *non-logical constants* that the computation will involve.
 - ▶ a *logic program* P , which is a multiset of Σ -formulas that specifies the meaning of the constants in Σ , and
 - ▶ a *query or goal* G , which is a Σ -formula.
- Computation is the process of attempting to prove the sequent $\Sigma : P \longrightarrow G$. If successful, the resulting proof could be returned, e.g., in forms of *answer substitutions*.

Problems with proof search

Given a sequent, there are potentially many directions to explore:

- We can use the cut rule.
- Structural rules: unrestricted contraction/weakening.
- Left/right introduction rules.
- Finding instantiations of variables in quantifier rules.
- Initial rule.

Some obvious reduction in search space:

- By cut-elimination, we need not consider the cut rule.
- Structural rules can be absorbed into initial and introduction rules.

More difficult problems:

- Variable instantiations: use *unification* algorithm.
- Left/right choices of intro rules: *goal-directed* proof search.

An idealized interpreter

An idealized interpreter has three components: signature Σ , a set of Σ -formulas \mathcal{P} (program) and a Σ -formula G (goal). The *state* of this idealized interpreter is denoted by the sequent $\Sigma : \mathcal{P} \longrightarrow G$. Desirable operational behaviors of this interpreter:

AND Reduce $\Sigma : \mathcal{P} \longrightarrow B_1 \wedge B_2$ to $\Sigma : \mathcal{P} \longrightarrow B_1$ and $\Sigma : \mathcal{P} \longrightarrow B_2$.

OR Reduce $\Sigma : \mathcal{P} \longrightarrow B_1 \vee B_2$ to either $\Sigma : \mathcal{P} \longrightarrow B_1$ or $\Sigma : \mathcal{P} \longrightarrow B_2$.

INST Reduce $\Sigma : \mathcal{P} \longrightarrow \exists_{\tau} x. B$ to $\Sigma : \mathcal{P} \longrightarrow B[t/x]$, for some Σ -term t .

AUGMENT Reduce $\Sigma : \mathcal{P} \longrightarrow B_1 \supset B_2$ to $\Sigma : \mathcal{P}, B_1 \longrightarrow B_2$.

GENERIC Reduce $\Sigma : \mathcal{P} \longrightarrow \forall_{\tau} x. B$ to $\Sigma, c : \tau : \mathcal{P} \longrightarrow B[c/x]$, where c is a “new constant”.

TRUE The $\Sigma : \mathcal{P} \longrightarrow \top$ is provable immediately.

Note: these reductions do not consider the logic program or the signature at all.
Behaviors of logical connectives cannot be modified by logic programs

Completeness problems

We cannot use arbitrary formulas as programs if we want to retain the goal-directed search behavior.

The following sequents have no goal-directed proofs:

- $\Sigma : p \vee q \longrightarrow q \vee p$
- $\Sigma : (ra \wedge rb) \supset q \longrightarrow \exists_i x (rx \supset q)$

What restrictions should be made about program formulas to guarantee completeness of goal-directed proof search?

Uniform provability

- A cut-free intuitionistic proof is a *uniform proof* if every sequent in the proof with a non-atomic succedent is the conclusion of a right-introduction rule.
- Let \vdash be a provability relation in some logic. Let \mathcal{D} be a set of formulas denoting program clauses and let \mathcal{G} be a set of formulas denoting goal formulas in an intended logic programming. The triple $\langle \mathcal{D}, \mathcal{G}, \vdash \rangle$ is an *abstract logic programming language* if and only if for every finite subset \mathcal{P} of \mathcal{D} and for every $G \in \mathcal{G}$, $\Sigma; \mathcal{P} \vdash G$ if and only if $\Sigma : \mathcal{P} \longrightarrow G$ has a uniform proof.

Horn clauses

Syntax of first-order Horn clauses:

$$\begin{aligned} G &::= \top \mid A \mid G \wedge G \mid G \vee G \mid \exists_{\tau} x. G \\ D &::= A \mid G \supset A \mid D \wedge D \mid \forall_{\tau} x. D \end{aligned}$$

D -formulas are called *program clauses* and G -formulas are called *goal formulas*. Let \vdash_{IL} denote intuitionistic provability. Then $\langle \mathcal{D}, \mathcal{G}, \vdash_{IL} \rangle$ is an abstract programming language.

Actually, for this fragment, classical and intuitionistic provability coincides.

Backchaining

$$\frac{\Sigma : \mathcal{P} \xrightarrow{D} A}{\Sigma : \mathcal{P} \longrightarrow A} \textit{decide}$$

$$\frac{}{\Sigma : \mathcal{P} \xrightarrow{A} A} \textit{initial}$$

$$\frac{\Sigma : \mathcal{P} \xrightarrow{D_1} A}{\Sigma : \mathcal{P} \xrightarrow{D_1 \wedge D_2} A} \wedge_L$$

$$\frac{\Sigma : \mathcal{P} \xrightarrow{D_2} A}{\Sigma : \mathcal{P} \xrightarrow{D_1 \wedge D_2} A} \wedge_L$$

$$\frac{\Sigma : \mathcal{P} \longrightarrow G \quad \Sigma : \mathcal{P} \xrightarrow{D} A}{\Sigma : \mathcal{P} \xrightarrow{G \supset D} A} \supset_L$$

$$\frac{\Sigma \vdash t : \tau \quad \Sigma : \mathcal{P} \xrightarrow{D[t/x]} A}{\Sigma : \mathcal{P} \xrightarrow{\forall \tau x. D} A} \forall_L$$

The right-intro rules with backchaining rules are complete for intuitionistic logic (for the horn fragment).

Hereditary harrop formulas

$$\begin{aligned} G &::= \top \mid A \mid G \wedge G \mid G \vee G \mid \exists_{\tau} x. G \mid D \supset G \mid \forall_{\tau} x. G \\ D &::= A \mid G \supset D \mid D \wedge D \mid \forall_{\tau} x. D \end{aligned}$$

$\langle \mathcal{D}, \mathcal{G}, \vdash_{IL} \rangle$ is an abstract programming language.

The presence of implication in HH-goals allows one to encode the notion of *modules*.

Higher-order Horn clauses

We can extend both Horn and hereditary Harrop fragments with higher-order quantifications, but the shape of program clauses must be restricted: Let \mathcal{H}_2 denote the set of λ -normal terms that do not contain \supset or \perp . Let A_r denote a rigid atom in \mathcal{H}_2 .

$$\begin{aligned} G &::= \top \mid A \mid G \wedge G \mid G \vee G \mid \exists_{\tau} x. G \mid D \supset G \mid \forall_{\tau} x. G \\ D &::= A_r \mid G \supset A_r \mid D \wedge D \mid \forall_{\tau} x. D \end{aligned}$$

Notice that the head of program clauses must be a *rigid* atomic formula in \mathcal{H}_2 . Both higher-order horn clauses and higher-order hereditary formulas are abstract programming languages.

Reasoning about logic programs

Cut and cut elimination can be used to relate provability in logic programming languages.

Suppose \vdash_L denote provability in the fragment of language under consideration. Let $\vdash_{L'}$ be a superset of \vdash_L (e.g., a richer logic with induction, etc.).

Suppose we can prove in L' that $P' \vdash_{L'} P$ for some program P . Then we know that whenever $P \vdash_{L'} G$, by cut we have $P' \vdash_{L'} G$, and by cut elimination, we have $P' \vdash_L G$.

What this means is that a proof of, say, program equivalence in the richer logic L' coincides with the operational equivalence of the logic programs.

EXAMPLES

- Modular programming
- Abstract data types
- Higher-order programming
- Hypothetical judgments
- Encoding operational semantics of languages