CSci 8980, Fall 2012
Specifying and Reasoning About Computational Systems
Foundations of Logic Programming

Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota

Lectures in Fall 2012

Logic Programming and Rule-Based Specifications

Horn clause logic possesses two features that are important to encoding rule based specifications

- First-order terms generalize traditional abstract syntax

$$[\alpha_1 \rightarrow \alpha_2] \implies arr([\alpha_1],[\alpha_2])$$
$$[(e_1\ e_2)] \implies app([e_1],[e_2])$$

- The logical structure of Horn clauses mirrors closely the structure of (syntax-directed) rules

$$\frac{\Gamma \vdash m : a \rightarrow b \qquad \Gamma \vdash n : a}{\Gamma \vdash (m\ n) : b}$$

$$\Downarrow$$

```
of(Gamma,app(E1,E2),Ty2) :-
      of(Gamma,E1,arr(Ty1,Ty2)), of(Gamma,E2,Ty1).
```


An Inadequacy of Traditional Abstract Syntax

No support is provided for binding notions

E.g., consider the mini-ML abstraction (`fn` *x* : *ty* => *e*)

The best possible representation in the first-order setting:

abst(x,[ty],[e])

Unfortunately, there are several problems with such a representation

- In a typeless setting, we have no capability for enforcing well-formedness constraints
- The representation is too sensitive to the name used for the abstracted variable
- No mechanism for ensuring scoping properties will be respected

Leaving such issues to be dealt with by the programmer seriously compromises the logical use of the encoding

Subtleties of Scoping

Ensuring that scoping properties are respected can be tricky

For example, consider the following encoding of type checking

```
of(Gamma,var(X),Ty) :- member((X:Ty),Gamma).
of(Gamma,app(E1,E2),Ty2) :-
     of(Gamma,E1,arr(Ty1,Ty2)),of(Gamma,E2,Ty1).
of(Gamma,abst(X,Ty1,E),arr(Ty1,Ty2)) :-
     of([(X:Ty1)|Gamma],E,Ty2).
```

This program seems to follow naturally from the typing rules

Unfortunately, it is *incorrect*

For example, the following query will succeed

```
?- of([],
      abst(x,int,abst(x,arr(int,int)),
                      app(var(x),var(x))),
      Ty).
```

Providing a Logical Treatment of Binding

We would like to provide a general, logic supported way to deal with binding constructs

More specifically, we desire the following

- data structures that allow relevant binding properties to be captured
- logical mechanisms for manipulating such data structures
- devices for reasoning about binding related computations

Dealing with the first two aspects requires a richer language than first-order Horn clause logic

To describe such a language we must first reconsider the foundations of logic programming

Towards a Foundation for Logic Programming

We will use the formalism of the sequent calculus in developing such a foundation

The sequent calculus framework provides a flexible mechanism for characterizing various logical systems

A characterization in this style, in fact, allows us also to talk about the *behaviour* of such systems

- it admits a *proof search* interpretation that gives a handle on the computational structure of logics
- it is a good means for presenting and establishing meta-theoretic and computational properties of logics
- it is applicable to a wide variety of logics, e.g. also to higher-order logics that we will want to discuss soon

Sequent calculi will be used for developing specification logics now and logics for reasoning about them later

The Constituents of a Sequent Calculus

The two components determining a sequent calculus are *sequents* and *rules*

- Sequents are the basic units of assertion

  Formally, they are pairs of *multisets* of (first-order) formulas that we will write as $\Gamma \vdash \Delta$

  - Γ is called the left-hand side or antecedent
  - Δ is called the right-hand side or succedent

  The formulas in Γ should be though of as *assumptions* and those in Δ as *possible conclusions*

- The rules are the devices by which we *derive* sequents

  We will typically partition the rules into three categories: *structural*, *identity* and *operational*

We will assume our logics use the following logical symbols: $\bot$, $\top$, $\vee$, $\wedge$, $\supset$, $\exists$ and $\forall$

The Structural Rules

There are two kinds of such rules: *contraction* and *weakening*

$$\frac{\Gamma, B, B \vdash \Delta}{\Gamma, B \vdash \Delta}\ cL \qquad \frac{\Gamma \vdash \Delta, B, B}{\Gamma \vdash \Delta, B}\ cR$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, B \vdash \Delta}\ wL \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, B}\ wR$$

Some comments on these rules:

- Gentzen's original formulation used lists instead of multisets so required an extra *interchange* rule
- Conversely, if we use sets instead of multisets, then we can drop the contraction rules
- Contraction rules are the main source of undecidability in (*cut-free*) sequent calculi

The Identity Rules

There are two rules of this kind

$$\frac{}{B \vdash B}\ \mathit{init} \qquad \frac{\Gamma_1 \vdash \Delta_1, B \quad \Gamma_2, B \vdash \Delta_2}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \mathit{cut}$$

Some comments on these rules

- The *init* rule can be restricted to the setting where B is atomic
- The *cut* rule is essential for mathematical reasoning but is problematic for analysis of provability, proof search and computation
- A key result for all the logics that we will use: the *cut* rule is eliminable or, conversely, admissible

  Thus, we will work with cut-free proofs but use cut when we need to in meta-theoretic arguments about provability

The Operational Rules

These rules *introduce* logical symbols into formulas

$$\frac{}{\Gamma \vdash \Delta, \top}\ \top R \qquad \frac{}{\bot, \Gamma \vdash \Delta}\ \bot L$$

$$\frac{\Gamma, B_1 \vdash \Delta \quad \Gamma, B_2 \vdash \Delta}{\Gamma, B_1 \vee B_2 \vdash \Delta}\ \vee L \qquad \frac{\Gamma \vdash \Delta, B_i}{\Gamma \vdash \Delta, B_1 \vee B_2}\ \vee R_i$$

$$\frac{\Gamma, B_i \vdash \Delta}{\Gamma, B_1 \wedge B_2 \vdash \Delta}\ \wedge L_i \qquad \frac{\Gamma \vdash \Delta, B_1 \quad \Gamma \vdash \Delta, B_2}{\Gamma \vdash \Delta, B_1 \wedge B_2}\ \wedge R$$

$$\frac{\Gamma_1 \vdash \Delta_1, B_1 \quad \Gamma_2, B_2 \vdash \Delta_2}{\Gamma_1, \Gamma_2, B_1 \supset B_2 \vdash \Delta_1, \Delta_2}\ \supset L \qquad \frac{\Gamma, B_1 \vdash \Delta, B_2}{\Gamma \vdash \Delta, B_1 \supset B_2}\ \supset R$$

$$\frac{\Gamma, B[t/x] \vdash \Delta}{\Gamma, \forall x\ B \vdash \Delta}\ \forall L \qquad \frac{\Gamma \vdash \Delta, B[c/x]}{\Gamma \vdash \Delta, \forall x\ B}\ \forall R$$

$$\frac{\Gamma, B[c/x] \vdash \Delta}{\Gamma, \exists x\ B \vdash \Delta}\ \exists L \qquad \frac{\Gamma \vdash \Delta, B[t/x]}{\Gamma \vdash \Delta, \exists x\ B}\ \exists R$$

In $\forall R$ and $\exists L$, c is a "new" constant and in $\forall L$ and $\exists R$, t is a closed term

Classical, Intuitionistic and Minimal Provability

- Classical provability ($\vdash_C$) is provability of arbitrary sequents in the calculus as presented that is called LK
- Intuitionistic provability ($\vdash_I$) is provability when no sequent is allowed to have more than one formula on the right

  This restriction implies changes to the inference rules, e.g., the *cut* and $\supset L$ rules become

  $$\frac{\Gamma_1 \vdash B \quad \Gamma_2, B \vdash \Delta}{\Gamma_1, \Gamma_2 \vdash \Delta}\ \mathit{cut} \qquad \frac{\Gamma_1 \vdash B_1 \quad \Gamma, B_2 \vdash \Delta}{\Gamma_1, \Gamma_2, B_1 \supset B_2 \vdash \Delta}\ \supset L$$

  Also, contraction on the right becomes inapplicable

  The resulting calculus is called LJ
- Minimal provability ($\vdash_M$) corresponds to intuitionistic provability in a calculus in which $\bot L$ is replaced by

  $$\frac{}{\bot, \Gamma \vdash \bot}\ \bot L$$

Using the Sequent Calculus

- Using the calculi we can show the derivability of formulas in different systems

  For example, consider

  $$(X \vee (Y \wedge Z)) \supset (X \vee Y) \wedge (X \vee Z)$$
  $$(P \supset Q) \supset ((\neg P) \vee Q)$$
  $$((\exists x\ P\ x) \supset Q) \supset (((P\ a) \vee (P\ b)) \supset Q)$$
  $$\exists x\ \forall y\ ((P\ x) \supset (P\ y))$$

  Here $\neg P$ stands for $P \supset \bot$
- To argue that the second and fourth formulas are not derivable in LJ, we need the cut elimination theorem
- We can get rid of thinning if we change the *init* rule to

  $$\frac{}{\Gamma, B \vdash B}\ \mathit{init}$$

  Also, we can understand now how to limit B to be atomic

Natural Deduction Calculus (Digression)

The sequent calculus can be thought to have its origins in a calculus that codifies a natural style of reasoning

- This calculus has introduction rules that essentially interpret the logical connectives, e.g.

$$\frac{A \quad B}{A \wedge B}\ \wedge\text{–}I$$

- Some rules, e.g. $\supset$–I, lead to reasoning from assumptions

  Thus, we have *elimination* rules for connectives based exactly on how the formula had to be derived, e.g.

$$\frac{A \wedge B}{A}\ \wedge\text{–}E_1 \qquad \frac{A \wedge B}{B}\ \wedge\text{–}E_2$$

- Classical logic is obtained by allowing $B \vee \neg B$ as initial formulas for any B
- The intuitionistic version has a constructive interpretation leading to the formulas-as-types interpretation

Sequent Calculus and Natural Deduction

- The sequent calculus essentially makes the bookkeeping aspect of the natural deduction calculus explicit
- The structural rules (contraction and thinning) are needed to realize the set oriented treatment of assumptions
- The inclusion of the cut rule makes it possible to easily show the completeness of the sequent calculus

  For example, consider the $\supset$–E rule and the $\supset L$ rule
- Cut elimination can be seen to be related to derivations in normal form

  I.e., derivations in which no elimination rule immediately follows an introduction rule
- Proof normalizability is, in fact, the same thing as proof term normalizability

Eliminating Contraction

In (the cut-free version of) the calculus for classical logic, we can absorb the essential uses of contraction into other rules as follows

$$\frac{\Gamma \vdash \Delta, B_1, B_2}{\Gamma \vdash \Delta, B_1 \vee B_2}\ \vee R^* \qquad \frac{\Gamma, B_1, B_2 \vdash \Delta}{\Gamma, B_1 \wedge B_2 \vdash \Delta}\ \wedge L^*$$

$$\frac{\Gamma \vdash \Delta, B_1 \quad \Gamma, B_2 \vdash \Delta}{\Gamma, B_1 \supset B_2 \vdash \Delta}\ \supset L^*$$

$$\frac{\Gamma, \forall x\ B, B[t/x] \vdash \Delta}{\Gamma, \forall x\ B \vdash \Delta}\ \forall L^* \qquad \frac{\Gamma \vdash \Delta, \exists x\ B, B[t/x]}{\Gamma \vdash \Delta, \exists x\ B}\ \exists R^*$$

For intuitionistic logic, the "principal formula" of $\supset L$ must also be contracted:

$$\frac{\Gamma, B_1 \supset B_2 \vdash \Delta, B_1 \quad \Gamma, B_2 \vdash \Delta}{\Gamma, B_1 \supset B_2 \vdash \Delta}\ \supset L$$

Other variants, yielding decidability in the propositional case,

One Characterization of Logic Programming

Logic programming means constructing derivations in some logical system

- Programs are finite collections of (arbitrary) formulas
- Goals (or logic programming queries) are arbitrary formulas
- Computation amounts to trying to construct a (classical, intuitionistic, minimal logic) proof for sequents of the form

$$\mathcal{P} \vdash G$$

  where $\mathcal{P}$ is a program and G is a goal formula

This reading has the virtue of associating a logic-based declarative reading with programs and goals

However, it has a severe drawback: it provides no hooks to give the programmer control over the structure of computations

An Alternative Characterization

Logic programming is about specifying and conducting search

- Programs describe a knowledge base, goals specify a search over the knowledge base
- Connectives and quantifiers are the devices for specifying the search
- How a programmer uses logical symbols to construct a goal formula determines the kind of search conducted

Giving the Search Viewpoint Substance

Let $\mathcal{P} \vdash_O G$ stand for "G succeeds when the program is given by $\mathcal{P}$"

Then we can attribute an operational semantics with the logical symbols as follows

SUCCESS	$\mathcal{P} \vdash_O \top$
OR	$\mathcal{P} \vdash_O (G_1 \vee G_2)$ iff $\mathcal{P} \vdash_O G_1$ or $\mathcal{P} \vdash_O G_2$
AND	$\mathcal{P} \vdash_O (G_1 \wedge G_2)$ iff $\mathcal{P} \vdash_O G_1$ and $\mathcal{P} \vdash_O G_2$
AUGMENT	$\mathcal{P} \vdash_O F \supset G$ iff $\mathcal{P} \cup \{F\} \vdash_O G$
INSTANCE	$\mathcal{P} \vdash_O \exists x\ G$ iff $\mathcal{P} \vdash_O G[t/x]$ for *some* t
GENERIC	$\mathcal{P} \vdash_O \forall x\ G$ iff $\mathcal{P} \vdash_O G[c/x]$ for a *new* constant c

Some Comments on the Search Semantics

- No treatment is prescribed for $\bot$

  The typical operational reading of this symbol has a meta-theoretic character that does not fit *within* the system
- No treatment is included of *atomic* goals
  - this is not relevant to describing the search interpretation of the logical symbols
  - saying something about them now would constrain the structure of program formulas too early
- Nothing is said about the *result* of a computation, but the semantics of existential quantifiers provides a possibility

  Solve the existential closure of a goal with free variables and return the instantiations as a trace
- The new characterization has the drawback that the connection with logic is at least partially lost!

Combining the Two Viewpoints

The first step in this direction: give the operational semantics a logical significance

The search semantics of each logical symbol is already sound with respect to its declarative (logical) reading

We can then think of restricting program formulas, goal formulas and the proof relation so that it is also *complete*

Definition *(Uniform Proofs)*
A uniform proof is an intuitionistic proof in which any sequent that contains a non-atomic formula on the right is the lower sequent of an inference rule that introduces the top-level logical symbol of that formula.

Thus $\mathcal{P} \vdash_O G$ can be thought of as the uniform provability of $\mathcal{P} \vdash G$

Combining the Two Viewpoints (Continued)

Of course, we cannot always guarantee the existence of uniform proofs

Logic programming corresponds to restricting programs, goals and proof relations in such a way that we can

Definition *(Abstract Logic Programming Language)*
A triple $\langle \mathcal{D}, \mathcal{G}, \vdash_R \rangle$ of two sets of formulas and a proof relation is an abstract logic programming language (ALPL) if and only if for any finite multiset $\mathcal{P}$ of elements from $\mathcal{D}$ and any $G \in \mathcal{G}$,

$$\mathcal{P} \vdash_R G \qquad \text{iff} \qquad \mathcal{P} \vdash G \text{ has a uniform proof}$$

Terminology
Given an ALPL $\langle \mathcal{D}, \mathcal{G}, \vdash_R \rangle$

- Members of $\mathcal{D}$ will be called *program clauses*
- Members of $\mathcal{G}$ will be called *goals* or *goal formulas*
- Finite multisets of program clauses will be called programs

Non-Examples of ALPLs

- If $\mathcal{D}$ contains disjunctions and $\mathcal{G}$ contains existentially quantified formulas; $\vdash_R$ can be $\vdash_C$, $\vdash_I$ or $\vdash_M$
  Counterexample: Consider $(p\ a) \vee (p\ b) \vdash \exists x\ (p\ x)$
- Let $\mathcal{G}$ contain implications and disjunctions and let $\vdash_R$ be $\vdash_C$
  Counterexample: Consider $\vdash (A \supset B) \vee A$ for A, B atomic
- Let $\mathcal{D}$ contain formulas of the form
  $$(B_1 \wedge \ldots \wedge B_n) \supset A$$
  where B_i is either A_i or $A_i \supset \bot$ for A, A_i atomic
  $A_i \supset \bot$ is written as $\neg A$, B_i is called a literal
  Let $\mathcal{G}$ contain existentials and let $\vdash_R$ be $\vdash_C$
  Counterexample: $p \supset (q\ a), \neg p \supset (q\ b) \vdash \exists x\ (q\ x)$

First two examples show that program clauses must not contain disjunctive information and $\vdash_C$ is too strong for AUGMENT

The Language of First-Order Horn Clauses

Let $\mathcal{G}_1$ and $\mathcal{D}_1$ be the set of G and D formulas given by

$$G \quad ::= \quad \top \mid A \mid G \vee G \mid G \wedge G \mid \exists x\ G$$
$$D \quad ::= \quad A \mid (G \supset A) \mid (D \wedge D) \mid \forall x\ D$$

where A is a (first-order) atomic formula

The *language of first-order Horn clauses* or *fohc* is either $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_C \rangle$ or $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_I \rangle$

Some Observations

- Within classical logic,
  - a G-formula is equivalent to the disjunction of a set of Prolog-style queries
  - a D-formula is equivalent to a set of positive Horn clauses (the formulas corresponding to Prolog clauses)
- Conversely, the logical formulas on which Prolog is based are contained in the relevant set $\mathcal{D}_1$ or $\mathcal{G}_1$

Recap: Search Interpretation for Logical Symbols

The search interpretation we will use for logical symbols:

SUCCESS	$\mathcal{P} \vdash_O \top$
OR	$\mathcal{P} \vdash_O (G_1 \vee G_2)$ iff $\mathcal{P} \vdash_O G_1$ or $\mathcal{P} \vdash_O G_2$
AND	$\mathcal{P} \vdash_O (G_1 \wedge G_2)$ iff $\mathcal{P} \vdash_O G_1$ and $\mathcal{P} \vdash_O G_2$
AUGMENT	$\mathcal{P} \vdash_O F \supset G$ iff $\mathcal{P} \cup \{F\} \vdash_O G$
INSTANCE	$\mathcal{P} \vdash_O \exists x\ G$ iff $\mathcal{P} \vdash_O G[t/x]$ for *some* t
GENERIC	$\mathcal{P} \vdash_O \forall x\ G$ iff $\mathcal{P} \vdash_O G[c/x]$ for a *new* constant c

Uniform Proofs: Proofs in LJ that are faithful to the search interpretation

ALPL: A triple of the form $\langle \mathcal{D}, \mathcal{G}, \vdash_R \rangle$ such that uniform proofs are complete for $\mathcal{P} \vdash G$ for any finite $\mathcal{P} \subseteq \mathcal{D}$ and $G \in \mathcal{G}$

The Language of First-Order Horn Clauses

Let $\mathcal{G}_1$ and $\mathcal{D}_1$ be the set of G and D formulas given by

$$\begin{array}{lll} G & ::= & \top \mid A \mid G \vee G \mid G \wedge G \mid \exists x\ G \\ D & ::= & A \mid (G \supset A) \mid (D \wedge D) \mid \forall x\ D \end{array}$$

where A is a (first-order) atomic formula

The *language of first-order Horn clauses* or *fohc* is either $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_C \rangle$ or $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_I \rangle$

The Language of First-Order Horn Clauses (Contd)

Whether we choose $\vdash_C$ or $\vdash_I$ in defining *fohc* makes no difference in identifying the language

This is because of the following observation

Lemma

If $\mathcal{P}$ is a finite subset of $\mathcal{D}_1$ and $G \in \mathcal{G}_1$ then $\mathcal{P} \vdash_C G$ iff $\mathcal{P} \vdash_I G$

Proof Sketch

Generalize the requirement to the following

$$\mathcal{P} \vdash G_1, ..., G_n \text{ is classically provable}$$

$$\Updownarrow$$

$$\text{for some } i,\ 1 \leq i \leq n,\ \mathcal{P} \vdash G_i \text{ has an intuitionistic proof}$$

Shown by induction, considering by cases the last rule in the proof
Details are left as an exercise □

The Language of First-Order Horn Clauses (Contd)

Theorem

fohc is an abstract logic programming language

Proof will be a special case of a later result using the equivalence of classical and intuitionistic provability

An Interesting Observation

Although Prolog is often explained via classical logic, its interpreters actually search for intuitionistic proofs

This is evident from the way disjunctive and existential goals are treated and the way program clauses are used

In particular, there is always only *one* formula that the interpreter tries to prove from the program

A Further Point about fohc

The language provides very little capability to model dynamics: programs and signature never change

First-Order Hereditary Harrop Formulas

Let $\mathcal{G}_2$ and $\mathcal{D}_2$ be the G and D formulas given by

$$\begin{array}{lll} G & ::= & \top \mid A \mid G \vee G \mid G \wedge G \mid \exists x\ G \mid \forall x\ G \mid D \supset G \\ D & ::= & A \mid G \supset D \mid D \wedge D \mid \forall x\ D \end{array}$$

where A is a (first-order) atomic formula

The *language of first-order hereditary Harrop formulas* or *fohh* is the triple $\langle \mathcal{D}_2, \mathcal{G}_2, \vdash_I \rangle$

The D-formulas are also called *(first-order)* hereditary Harrop formulas

These formulas are related to ones attributed to Harrop:

$$H \quad ::= \quad \top \mid A \mid B \supset H \mid H \wedge H \mid \forall x\ H$$

where B is an arbitrary formula and A is atomic

Specifically, the restriction on $\vee$ and $\exists$ is applied hereditarily to all positive contexts in program clauses

Some Observations About fohh

- *fohh* provides means for scoping over names and code
  - the goal $(\forall x\ G)$ gives us a new name to use when trying to solve G
  - the goal $D \supset G$ allows the code in D to be additionally used in solving G

  We will see later the use of these devices in treating side conditions and recursion over binding structure
- The restriction of the proof relation to intuitionistic provability for *fohh* is *necessary*, e.g. the sequents

  $\vdash p \vee (p \supset q)$, and
  $((p\ a) \wedge (p\ b)) \supset q \vdash \exists x\ ((p\ x) \supset q)$

  both have classical proofs but no uniform proofs
- Employing intuitionistic provability to provide the declarative semantics of logic programming is both *acceptable* and potentially *useful*

Towards Showing that fohh is an ALPL

Our proof will rely on the ability to apply the left inference rules in a *focused* form in *fohh* related derivations

The following definition is useful in articulating this idea

Definition *(Instances of program clauses)*

For a formula $D \in \mathcal{D}_2$, its set of *instances* is denoted by $[\![D]\!]$ and is defined as follows

- If D is $\forall x\ D'$, then $[\![D]\!] = \bigcup\{[\![D'[t/x]]\!] \mid t \text{ is a closed term}\}$
- If D is $(D_1 \wedge D_2)$, then $[\![D]\!] = [\![D_1]\!] \cup [\![D_2]\!]$
- If D is A, then $[\![D]\!] = \{A\}$
- If D is $G \supset D'$, then $[\![D]\!] = \{G \bullet C \mid C \in [\![D']\!]\}$ where
  - $G \bullet A = G \supset A$ if A is atomic
  - $G \bullet (G' \supset A) = (G \wedge G') \supset A$

If $\mathcal{P}$ is *fohh* program, then $[\![\mathcal{P}]\!] = \bigcup\{[\![D]\!] \mid D \in \mathcal{P}\}$

Towards Showing that fohh is an ALPL (Contd)

Lemma If $\mathcal{P} \vdash G$ has an intuitionistic proof of size l, then

- G is $\top$, or
- G is $G_1 \vee G_2$ and either $\mathcal{P} \vdash G_1$ or $\mathcal{P} \vdash G_2$ has an intuitionistic proof of size less than l, or
- G is $G_1 \wedge G_2$ and $\mathcal{P} \vdash G_1$ and $\mathcal{P} \vdash G_2$ both have intuitionistic proofs of size less than l, or
- G is $D \supset G'$ and $\mathcal{P}, D \vdash G$ has an intuitionistic proof of size less than l, or
- G is $\exists x\ G$ and, for some closed term t, $\mathcal{P} G[t/x]$ has an intuitionistic proof of size less than l, or
- G is $\forall x\ G$ and, for a new constant c, $\mathcal{P} \vdash G[c/x]$ has an intuitionistic proof of size less than l or
- $G = A$ is an atom such that $A \in [\![\mathcal{P}]\!]$ or there is a formula $((G_1 \wedge \cdots \wedge G_n) \supset A) \in [\![\mathcal{P}]\!]$ such that, for $1 \leq i \leq n$, $\mathcal{P} \vdash G_i$ has an intuitionistic proof of size less than l

Showing that fohh is an ALPL

Sketch of the Proof of the Lemma:

Shown by an induction on the sizes of proofs

A crucial observation used in the proof:

> *Every sequent in a proof of $\mathcal{P} \vdash G$ has the same form, i.e., the lefthand side of each sequent is a program and the righthand side is a goal formula*

Details of the proof are left as an exercise □

Theorem: *fohh* is an ALPL.

Proof Follows immediately from the lemma

Actually, the lemma provides additional information: atomic goals can be solved by a generalized "backchaining" step

A Simplified Proof System for fohh

Sequents now have the form $\Sigma; \mathcal{P} \vdash G$ where Σ is signature

$$\frac{}{\Sigma; \mathcal{P} \vdash \top}\ \top R \qquad \frac{\Sigma; \mathcal{P} \vdash B_1 \qquad \Sigma; \mathcal{P} \vdash B_2}{\Sigma; \mathcal{P} \vdash B_1 \wedge B_2}\ \wedge R$$

$$\frac{\Sigma; \mathcal{P} \vdash B_1}{\Sigma; \mathcal{P} \vdash B_1 \vee B_2}\ \vee R_1 \qquad \frac{\Sigma; \mathcal{P} \vdash B_2}{\Sigma; \mathcal{P} \vdash B_1 \vee B_2}\ \vee R_2 \qquad \frac{\Sigma; \mathcal{P}, B_1 \vdash B_2}{\Sigma; \mathcal{P} \vdash B_1 \supset B_2}\ \supset R$$

$$\frac{\Sigma; \mathcal{P} \vdash B[t/x] \qquad t \text{ is a } \Sigma\text{-term}}{\Sigma; \mathcal{P} \vdash \exists x\ B}\ \exists R \qquad \frac{c, \Sigma; \mathcal{P} \vdash B[c/x]}{\Sigma; \mathcal{P} \vdash \forall x\ B}\ \forall R, c \text{ new}$$

$$\frac{\Sigma; \mathcal{P} \xrightarrow{D} A}{\Sigma; \mathcal{P} \vdash A}\ \textit{decide}, D \in \mathcal{P} \qquad \frac{}{\Sigma; \mathcal{P} \xrightarrow{A} A}\ \textit{init}$$

$$\frac{\Sigma; \mathcal{P} \xrightarrow{D} A \quad \Sigma; \mathcal{P} \vdash G}{\Sigma; \mathcal{P} \xrightarrow{G \supset D} A}\ \supset L \qquad \frac{\Sigma; \mathcal{P} \xrightarrow{D_i} A}{\Sigma; \mathcal{P} \xrightarrow{D_1 \wedge D_2} A}\ \wedge L \qquad \frac{\Sigma; \mathcal{P} \xrightarrow{D_2} A}{\Sigma; \mathcal{P} \xrightarrow{D_1 \wedge D_2} A}\ \wedge L$$

$$\frac{\Sigma; \mathcal{P} \xrightarrow{D[t/x]} A \qquad t \text{ is a } \Sigma\text{-term}}{\Sigma; \mathcal{P} \xrightarrow{\forall x\ D} A}\ \forall L$$

Comments on the Simplified Sequent Calculus

- This calculus is intended for constructing proofs of sequents of the form $\Sigma; \mathcal{P} \vdash G$ where Σ is the ambient signature
- By "Σ-term" we mean a (closed) term all of whose constants and function symbols are drawn from Σ
- Clearly, for *fohh*, the signature and program components of sequents can grow during proof search
- For *fohc*, to which this calculus also applies, the program and signature remain fixed throughout the computation
- The left rules have taken on a focused character: a clause is picked and then "processed" completely
- The completeness of this calculus follows easily from the lemma

  The left rules essentially generate a clause instance dynamically and use it in search

A Simplified Structure for Program Clauses

The operational semantics for atomic goals allows a "compilation" of program clauses

In particular, the goal formulas and program clauses for *fohh* can be reduced to the form

$$\begin{array}{lll} G & ::= & \top \mid A \mid G \vee G \mid G \wedge G \mid \exists x\ G \mid \forall x\ G \mid D \supset G \\ D & ::= & A \mid G \supset A \mid \forall x\ D \end{array}$$

Here the correspondence to Prolog-like clauses becomes clear

In particular, each clause adds meaning to the definition of a particular predicate

Examples

AUGMENT currently provides a means for nested definitions like let

```
reverse L1 L2 :-
   ((pi L\ rev_aux nil L L),
    (pi X\ pi L1\ pi L2\ pi L3\
       rev_aux (X::L1) L2 L3 :-
           rev_aux L1 (X::L2) L3))
   => rev_aux L1 [] L2.
```

Here `(pi x\ (P x)` is concrete syntax for $\forall x\ P(x)$ and `=>` for $\supset$

Also, we have used a Prolog-like representation for conjunctions and top-level quantifiers

For interesting uses of GENERIC, we must wait till higher-order features have been introduced

Examples

fohh encompasses all the programs from (pure) Prolog

"Real" examples using the new search primitives must wait till we have introduced higher-order capabilities

However the structure of the proof system can be explored using the following AI example

- The program has the following components:
  - A jar is sterile if every germ in it is dead
    $$\forall y\ ((\forall x\ (germ(x) \supset (in(x, y) \supset dead(x)))) \supset sterile(y))$$
  - A germ in a heated jar is dead
    $$\forall x\ \forall y\ ((heated(y) \wedge germ(x) \wedge in(x, y)) \supset dead(x))$$
  - A (particular) jar is heated
    $$heated(j)$$
- The goal: "Is there a sterile jar?"
  $$\exists x\ sterile(x)$$

Substitution and Quantification in *fohh*

In *fohh*, quantifiers have scope that is narrower than the entire goal or program clause

This means we have to be careful about quantifier scopes

For example, given the clause (written as in Prolog)

$$p(X)\ :-\ \forall y\ q(X, y)$$

instantiating X with $f(y)$ should *not* yield

$$p(X)\ :-\ \forall y\ q(f(y), y)$$

Rather, it should yield

$$p(X)\ :-\ \forall z\ q(f(y), z)$$

Similarly, the goal

$$\exists x\ \forall y\ x = y$$

should not be solvable

Towards an Interpreter for *fohh*

The simplified proof system provides a structure for an interpreter

However, two questions must be treated more carefully in a realistic interpreter

- How to deal with essential existential quantifiers?
- How to continue the search when the goal is atomic?

For the former, we can think of using "logic variables"

For the latter, the simplified structure of program clauses yields a natural form of backchaining a la Prolog

However, we have to be careful about quantifier scopes

We can solve the last issue using a numeric tag with constants and variables and an "occurs-check" in unification