Specifying and Reasoning About Computational Systems

Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota

Slides Collated for the CADE 2015 Tutorial
August 2015

Overview of the Slides

These slides are a collated and lightly edited version of those used in graduate courses at ITU and U Minnesota

There are organized into six parts

- Foundations of Logic Programming
- Higher-Order Logic Programming
- λProlog and Lambda Tree Syntax
- Examples of Specifications in λProlog
- Unification of λ-terms
- Reasoning about Specifications

Foundations of Logic Programming

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

$$[\alpha_1 \to \alpha_2] \implies \mathit{arr}([\alpha_1], [\alpha_2])$$
$$[(e_1\ e_2)] \implies \mathit{app}([e_1], [e_2])$$

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

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

$$\Downarrow$$

```
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

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

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

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

  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

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

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

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 $G \in \mathcal{G}$,

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

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
  $$(B_1 \wedge ... \wedge B_n) \supset A$$
  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 $\mathcal{G}_1$

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

$$\mathcal{P} \vdash G_1, ..., G_n \text{ is classically provable}$$
$$\Updownarrow$$
for some i, $1 \leq i \leq n$, $\mathcal{P} \vdash G_i$ 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

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

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

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

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

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

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

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

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

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)

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

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

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

Rather, it should yield

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

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

Higher-Order Logic Programming

The Lambda Calculus and Higher-Order Logic

Some important perspectives for the present setting

- We will take a logical rather than a computational view of lambda calculus
  - Equality between terms will be the primary notion
  - Reduction will mainly be a vehicle for determining equality
- Lambda calculus will help in building a higher-order logic
  - lambda terms will figure as the arguments of predicates
  - lambda terms will also be used to construct complex formula-valued expressions that are quantified over
- Lambda terms will mainly be used to *represent* objects
  - The calculus strength must be calibrated to analyzing syntax structure
  - Stronger (non-syntactic) properties of objects will be expressed via the relational framework

Lambda Calculus Basics

We will started untyped, as we did with first-order terms

The collection of lambda terms is parameterized by a set $\mathcal{C}$ of constants and a set $\mathcal{V}$ of variables:

- any element of $\mathcal{C}$ or $\mathcal{V}$ is a term
- the abstraction of $x \in \mathcal{V}$ over a term t, written $(\lambda x\ t)$, is a term
- the application of term t_1 to t_2, written $(t_1\ t_2)$, is a term

The usual conventions apply for minimizing parentheses

The constants in our setting will typically be *uninterpreted*

The free variables of a term t will be denoted $\mathcal{FV}(t)$

An important notion: the *logically correct* or *capture avoiding* substitution of t_2 for x in t_1, denoted by $t_1[t_2/x]$

Relations on Lambda Terms

- α-*conversion*: Replacement of a subterm of the form $\lambda x\ t$ by $\lambda y\ t[y/x]$ provided $y \notin \mathcal{FV}(t)$
- β-*conversion*: The reflexive and transitive closure of α-conversion and
  - β-*contraction*: Replacing a subterm $(\lambda x\ t_1)\ t_2$ by $t_1[t_2/x]$
  - β-*expansion*: Converse of β-contraction

  Related terminology: β-redex, β-reduction
- η-conversion: The reflexive and transitive closure of
  - η-*contraction*: Replacing a subterm $\lambda x(t\ x)$ by t provided that $x \notin \mathcal{FV}(t)$
  - η-*expansion*: Converse of η-contraction

  Related terminology: η-redex, η-reduction

The reflexive and transitive closure of α-, β- and η-conversion defines equality, called λ-conversion and written $=_\lambda$

Expressivity of the Untyped Lambda Calculus

The Essential Idea

Pick a representation for objects and then ask what functions on them are encodable, using β-reduction for evaluation

Turns out a lot: In fact, all computable functions can be encoded in this way

One particular observation that will be of interest presently

Theorem

Any equation of the form

$$x\ y_1\ \ldots\ y_n = t$$

where $x, y_1, \ldots, y_n$ are variables and t is a λ-term can be solved in the lambda calculus

In other words, there is a λ-term X such that

$$X\ y_1\ \ldots\ y_n =_{\lambda\beta\eta} t[X/x]$$

Building a Higher-Order Logic

The lambda calculus provides a primitive understanding of functionality

Can we build a logic of predicates, connectives and quantifiers over this?

The General Idea

- Designate particular constants from $\mathcal{C}$ to represent connectives and quantifiers
- Build in an interpretation of these symbols through sequent calculus or natural deduction rules

The thought is quite seductive: if it works we would get a *foundation* for mathematics

Untyped Lambda Calculus Logic is Inconsistent

Unfortunately, the idea *does not* work

In particular, any logical system that includes the following rules

$$\frac{X \qquad X =_{\lambda\beta\eta} Y}{Y}\ EQ$$

$$\frac{\begin{array}{c}[X]\\ \vdots\\ Y\end{array}}{X \supset Y}\ \supset I \qquad\qquad \frac{X \qquad X \supset Y}{Y}\ \supset E$$

is inconsistent

These rules are so basic that the project seems doomed

Curry's Paradox

Curry showed that it is possible to construct a derivation for any formula M in such a logic

By the properties of the untyped lambda calculus, we know that there is a term X such that $X =_{\lambda\beta\eta} X \supset M$

We then construct the following proof π of $X \supset M$

$$\dfrac{\dfrac{X \qquad \dfrac{X =_{\lambda\beta\eta} X \supset M}{X \supset M}\, EQ}{M}\, \supset E}{X \supset M}\, \supset \mathsf{I}$$

We now complete the proof of M as follows

$$\dfrac{\pi \quad X \supset M =_{\lambda\beta\eta} X}{\dfrac{X}{M}\, EQ \qquad \pi}\, \supset E$$

The Simple Theory of Types

Church's resolution to this problem was to introduce types

The main formal purpose of types was to recognize a functional hierarchy

More specifically

- we have a type for propositions and possibly several atomic types
- we form function types from other known types

These types restrict the kinds of lambda terms we can form and thereby avoid the paradoxes

The resulting logic is not foundational but still is good for formalization

Following this recipe results in a class of logics that we will generically call the *Simple Theory of Types*

Building a Typed Higher-Order Logic

The logic is constructed by giving the type o a special status

Terms of this type are identified as *formulas* from which we form sequents

We also distinguish specific constants over the type o used to represent connectives and quantifiers

$\vee, \wedge, \supset$	of type `o -> o -> o`
$\top, \bot$	of type o
$\Pi_\alpha, \Sigma_\alpha$	of type `(α -> o) -> o`

Here Π_α and Σ_α represent a *family* of constants, one for each type α

These constants are called *generalized quantifiers*

We will often omit the type subscript when we want to talk of these constants collectively

Representing the Familiar Quantifiers

Church's proposal: Handle binding effect through abstraction, leaving only the predication aspect to be treated specially

For example, consider representing $\forall x\ P(x)$

- First convert $P(x)$ into a "set" by abstracting over the x
  Specifically $\lambda x\ P(x)$ represents the set of things of which P is true
- Then make the quantifier a predication over this set
  $(\Pi\ (\lambda x\ P(x)))$, representing $\forall x\ P(x)$, effectively says "P is true of everything"

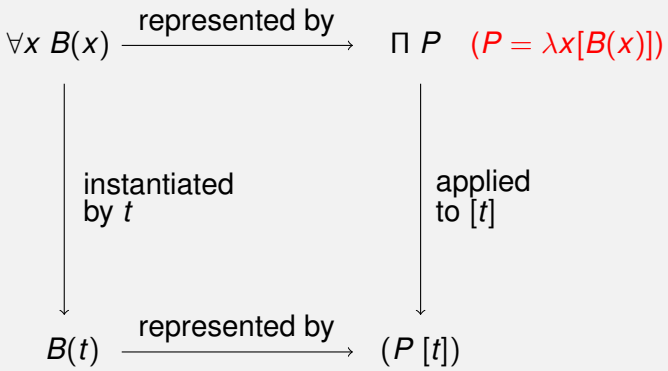
Similar idea works for $\exists x\ P(x)$ and also for other (non-logical) binding operators we will want to encode

In these cases, Π and Σ represent the *universal set* and the *existential set* recognizers

Realizing Substitution via β-Conversion

Advantage of such a representation: Substitution is completely and correctly realized through application and β-conversion

For example, we have the following correspondence



Scope respecting substitution is implicit in the λ-conversion rules

Sequent Calculus for Higher-Order Logic

Our sequents will now contain formulas from the new language

Beyond this, the only change to the LK/LJ rules is to build in λ-conversion

Can be done in one of two ways

- Assume that matching of formulas to schemas incorporates λ-conversion
- Add one more rule of the form

$$\frac{\Gamma' \vdash \Delta'}{\Gamma \vdash \Delta}\ \lambda$$

  where Γ and Γ' and Δ and Δ' differ only by λ-conversion

We will assume the former approach

Note that Church's logic has additional "mathematical" axioms that we do not want or use in our setting

Predicate Variables and Substitution

Even though the logic looks very similar to first-order logic, its behaviour can be wildly different

The main source of difference is predicate variables and what substituting for them can do to the formula structure

For example consider the formula

$$\forall p\ ((p\ A) \supset (c\ A))$$

Instantiating the quantifier with $\lambda z((q\ z) \wedge \forall x\ ((z\ x) \supset (q\ x)))$ yields

$$((q\ A) \wedge \forall x\ ((A\ x) \supset (q\ x))) \supset (c\ A)$$

In other words, a substitution instance can have a vastly different logical structure from the original formula

This difference has a significant impact on the meta-theory and on matters such as proof search

Logic Programming in Higher-Order Logic

We would like to extend *fohh* in such a way that

- useful higher-order features become available
- the search related interpretation is preserved
- determining predicate substitutions can be done in sensible computational steps

To realize these objectives we have to restrict predicate quantification and the kinds of instantiations to be considered

The key questions become

- what atoms can appear at the heads of clauses, and
- what logical symbols can appear in predicate arguments

The Need to Restrict Predicate Quantification

Program clauses in the reduced form of *fohh* have the structure

$$\forall x_1 \ \ldots \forall x_n \ G \supset A$$

where A is an atomic formula

In the higher-order setting, A could be rigid or flexible

Letting it have a flexible head is problematic

- A clause of the form

  $$\forall p \ G \supset (p \ t_1 \ \ldots \ t_m)$$

  adds meaning to *all* relations; this is at least anti-modular

- Such clauses can easily lead to inconsistent programs

  For example, consider the following

  $\forall p \ (q \supset p) \quad \wedge \quad q$

  From this, *any* formula can be derived

We shall therefore require clause heads to be rigid atoms

The Need for Restricting Predicate Arguments

Logical symbols can appear in terms in higher-order logic

Such symbols can end up instantiating predicate variables and thereby affecting top-level formula structure

This can potentially impact the completeness of goal-directed search

For example, consider the "goal formula"

$$\exists Q \, \forall p \, \forall q \, ((r \ (p \supset q)) \supset (r \ (Q \ p \ q))) \wedge \\ (Q \ (t \vee s) \ (s \vee t))$$

This formula is provable but its proof requires solving the goal

$$(t \vee s) \supset (s \vee t)$$

for which no uniform proof exists

Disallowing $\bot$ and $\supset$ in arguments provides a static approach to ruling out such problems

Higher-Order Hereditary Harrop Formulas

A *positive* atom is a (higher-order) atomic formula of the form

$$(P \ t_1 \ \ldots \ t_n)$$

in whose arguments the symbols $\supset$ and $\bot$ do not appear

The atom is *flexible* if P is a variable and *rigid* otherwise

Let $\mathcal{G}_3$ and $\mathcal{D}_3$ be the G and D formulas given by

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

where A stands for a positive atom and A_r for a rigid atomic formula

The *language of higher-order hereditary Harrop formulas* or *hohh* is the triple $\langle \mathcal{D}_3, \mathcal{G}_3, \vdash_I \rangle$

Showing *hohh* is an ALPL

Let a *positive term* be any λ-term in which the symbols $\supset$ and $\bot$ do not appear

Key Lemma

Let $\mathcal{P}$ be an *hohh* program and let G be an *hohh* goal such that $\mathcal{P} \vdash G$ has an intuitionistic proof

Then $\mathcal{P} \vdash G$ must have such a proof in which in any occurrence of the rules

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

the term t is restricted to being positive

Thus it is enough to consider proofs in which all sequents have the "normal form" that makes the first-order proof work

The proof of the lemma relies on a proof transformation

Computation via Proof Search for *hohh*

We can describe a reduced proof system for *hohh* that for the most part preserves the nature of computation for *fohh*:

- The top-level connective of a non-atomic goal determines the next step
- A generalized backchaining works for rigid atomic goals

Of course, in constructing instances, we will now have to consider *typed* λ*-terms* over a given signature

One new thing in the higher-order setting is that we have also to consider *flexible atomic goals*

The meta-theory provides a simple answer here:

- delay the solving of such goals
- eventually, when only flexible goals remain, use a "universal set substitution"

In specification logic applications, though, we will be interested in situations where such flexible goals *will not* arise

λProlog and Lambda Tree Syntax

Overview of the Language λProlog

The λProlog language is a realization of *hohh* with some additional features

- polymorphism in the style of ML is provided for
- AUGMENT and GENERIC (the "scoping constructs") are exploited to provide a form of modularity that is justified via logic
- to support programming, some "impure" control feature are inherited into the language from Prolog

With our focus on specification and reasoning, we will eschew use of the last feature

We will not discuss polymorphic typing in detail because it will be largely irrelevant to λ-tree syntax applications

We will discuss modularity features but only at a programming/specification level

The Types Language

The actual types language includes *type variables* denoted by tokens starting with uppercase letters

Several sorts like `int`, `real`, `string` and `o` are built-in as also the type constructor `list`

The programmer can extend this collection via declarations of the form

```
kind    tycon   type -> ... -> type.
```

The number of occurrences of `type` determines the arity.

For example, the declarations

```
kind    i       type.
kind    pairty  type -> type -> type.
```

make it possible to write the type expression

```
i -> (list int) -> (pairty i (list int))
```

The Terms Language

These are (ML-style) typed λ-terms with $\lambda x\ F$ written as `(x \ F)` and ($F\ T$) written as `(F T)`

Types of variables ar generally inferred but can also be specified at binding site: e.g. `((x:int) \ F)`

The types of constants are identified via declarations such as

```
type  -   int -> int -> int.
type  ::  (A -> (list A) -> (list A)).
```

Constants can also be declared to be operators. e.g.

```
infixl  -  255.
infixr  :: 255.
```

I.e. "`-` and `::` are infix and, resp., left and right associative operators of precedence level 255"

Abstraction has lowest precedence and application highest

Built-In Constants in λProlog

- Usual constants associated with `int`, `real`, `string`, etc
- The following polymorphic constants of `list` type

```
type nil  (list A)
type ::   A -> (list A) -> (list A)
```

- The following *logical constants*

```
type   true  o.
type   &, ;, =>, :-, ,   o -> o -> o.
type   sigma, pi         (A -> o) -> o.
infixl  ,   110.
infixr  &   120.
infixl  ;   100.
infix   :-  0.
infixr  =>  130.
```

`=>` and `,` are intended for use at the top-level in goals

`:-` and `&` are indented for use at the top-level in clauses

Conventions Used in Presenting Clauses and Goals

Programs

Listing of clauses, each terminated with a period

Unbound tokens beginning with upper-case letters are implicitly universally quantified over the clause

For example

```
append nil L L.
append (X::L1) L2 (X::L3) :- append L1 L2 L3.
```

Queries

Unbound tokens beginning with upper-case letters are implicitly existentially quantified over the query, e.g.

```
?- append (1::nil) (2::nil) L.
```

Similar to Prolog syntax, except programs and goals are typed and curried notation is used

A Conservative Extension to *hohh* Program Clauses

Atomic program clauses can have variable heads provided these are bound by goal-level universal quantifiers

For example, the following definition is acceptable

```
reverse L1 L2 :-
   (pi rev_ aux \
      (((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)).
```

By the time the embedded clauses are added to the program, their heads become rigid atoms

This extension allows auxiliary predicate names to be hidden

Modules and Signatures

- In the Teyjus system, code is organized into *modules*:

```
module lists.
type rev  (list A) -> (list A) -> o.
type rev_aux  (list A) -> (list A) -> (list A) -> o.
rev L1 L2  :-  rev_aux L1 nil L2.
rev_aux nil L1 L2.
rev_aux (X::L1) L2 L3 :- rev_aux L1 (X::L2) L3.
end
```

- The external views of a module is provided by a *signature*:

```
sig  lists.
type rev  (list A) -> (list A) -> o.
end
```

- Each module resides in a separate file and the signature resides in a related file
- Hidden constants correspond to existential quantification over clauses derived from universal quantification in goals

Modules and Signatures (Continued)

- Modules can also be *composed* using the accumulation construct

```
module fo_prover.
  accumulate lists.

   ...
end
```

Predicate definitions and data constructors "exported" from `lists` can be used in `fo_prover`

Accumulation can be understood via the following steps

- A module without accumulation corresponds to a *D*-formula
- Accumulation inserts the formula corresponding to the accumulated module in place
- Hiding arising from signatures is realized automatically through the logical interpretation of (existential) quantifiers

Animating Specifications Using the Teyjus System

The Teyjus system consists of three conceptual parts:

- An abstract machine supporting, low-level, λProlog relevant operations
- A compiler for translating to abstract machine programs
- A linker for realizing modularity notions with separate compilation

Running a program involves compiling a module to bytecode form, linking it with other modules and running the result

The full system has other utilities such as a disassembler, a dependency analyzer and a pretty printer

For details, check out `http://teyjus.cs.umn.edu` and the google code page

Higher-Order Programming in *hohh*

Predicate variables permit a familiar style of higher-order programming

```
type mappred (A -> B -> o) ->
        (list A) -> (list B) -> o.
mappred P nil nil.
mappred P (X::L1) (Y::L2) :-
          P X Y, mappred L1 L2.
```

Suppose we are given additionally the following clauses

```
parent bob mary.          parent sue peter.
parent mary john.         parent peter james.
```

The following are then sensible goals

```
?- mappred parent (bob::sue::nil) L.
?- mappred (x\ y\ sigma z\
             ((parent x z),(parent z y)))
           (bob::sue::nil) L.
```

Predicate Substitutions in *hohh*

We could also ask if it is possible to synthesize a predicate substitution

For example, consider the goal

```
?- mappred P
            (bob::sue::nil) (mary::peter::nil).
```

However, the computation of goal solving is too complex to try and run in reverse

Logically, there are also too many solutions to such queries: relevant abstractions over any succeeding goals will work

The meta-theory suggests the "largest" set as a solution

Here that solution would be

```
P = x\ y\ true
```


Mapping Function Evaluation

There is also a functional counterpart to `mappred`:

```
type mapfun (A -> B) -> (list A) -> (list B) -> o.
mapfun F nil nil.
mapfun F (X::L1) ((F X)::L2) :- mapfun F L1 L2.
```

Now computation over list elements involves only β-conversion

For example consider the following query

```
mapfun (x\ (x + 5)) [3,7] L
```

As before, we can ask if a reverse computation is also possible

```
mapfun F [3,7] [3 + 5, 7 + 5]
```

In this case, the weakness of evaluation permits a meaningful answer

The answer that is provided also shows an interesting capability to analyze syntax, e.g. abstracting structural properties

Structural Analysis and Binding Scope

The explicit scoping of quantifiers in *hohh* allows richer structure analysis problems to be posed

Suppose that = is defined as follows

```
type  =  A -> A -> o.
infix =  50.
X = X.
```

Then compare the following

```
?- pi g\ pi a\ sigma F\ (F a) = (g a a).
?- pi g\ sigma F\ pi a\ (F a) = (g a a).
```

A similar issue arises simply from the presence of λ-abstraction

```
?- (x\ F + (G x)) = (x\ (2 + x)).
?- (x\ F + (G x)) = (x\ (x + x)).
?- (x\ (F x) + (G x)) = (x\ (x + x)).
```


Limitations of Equation Solving

The simply typed λ-calculus without any defined (non-logical) constants provides a weak setting for equation solving

For example, the following query has no solutions

```
?- X + 3 = 8.
```

even though the substitution of `5` for *X* would work if `+` were interpreted

As another example, consider

```
?- (F a) = c, (F b) = d
```

A solution to this would require a built-in conditional

The structure analysis capability obtained through our term language is quite weak

However, this an asset rather than a problem: this is what makes structure analysis all about syntax and not "semantics"

Higher-Order Abstract Syntax (HOAS)

This term refers to the use of λ-abstraction in the meta-language to represent binding in object languages

There are varied arguments for HOAS, e.g.

- binding is another notion to be rightly abstracted away in treating syntactic objects
- treating binding in this way makes meta-language devices available for treating issues like scope and substitution

However, the interpretation of HOAS depends a lot on the strength of the λ-calculus and how it is used in representation

Some approaches also have problems with representational adequacy arising for issues like the presence of "exotic terms"

λProlog, which actually pioneered the HOAS idea, supports a particular variant that is free of such problems

λ-Tree Syntax

In this approach, representations use only a *very basic* treatment of binding beyond the encoding of first-order structure

λProlog supports this idea through the following

- only a very weak λ-calculus is used
  λ-terms are for representation, *not* for evaluation
- λ-terms are treated *intensionally*
  I.e., the analysis of syntactic objects can be realized directly through the analysis of their representations
- logical devices are provided for realizing recursion over binding structure
  The use of the relational setting (logic programming) is critical to this

We will illustrate these ideas next through some simple examples

Representing First-Order Formulas

The following λProlog signature defines a representation

```
kind  term, form   type.
type  ff, tt       form.
type  &&, !!, ==>  form -> form -> form.
infixl &&          5.
infixl !!          4.
infixr ==>         3.
type  all, some    (term -> form) -> form.
type   p   term -> form.
type   f   term -> term.
type   a   term.
```

The representation of the formula $\forall x\ \exists y\ ((p(f(x)) \supset p(f(y))))$:

```
(all x\ some y\ ((p (f x)) ==> (p (f y))))
```

Note that equivalence under renaming is immediate from the representation

Recursion Over Formula Structure

Recursion over first-order structure is straightforward

```
type is_term  term -> o.
type atomic, quant_free  form -> o.
is_term a.
is_term (f T) :- is_term T.
atomic (p T) :- is_term T.
quant_free F :- atomic F.
(quant_free ff) & (quant_free tt).
(quant_free (F1 && F2)) &
(quant_free (F1 !! F2)) &
(quant_free (F1 ==> F2)) :-
      quant_free F1, quant_free F2.
```

However, how about recursion over quantifier structure?

The issue: argument of `all` and `some` have function type

Recursion over Binding Structure

The approach used in λProlog

- Use GENERIC to introduce a new constant to lower the type
- Use AUGMENT to assign properties to it before continuing the recursion

For example, consider recognizing *fohc* goal formulas

```
type  is_goal  form -> o.
is_goal F :- atomic F.
is_goal tt.
is_goal (F1 && F2) :- is_goal F1, is_goal F2.
is_goal (F1 !! F2) :- is_goal F1, is_goal F2.
(is_goal (some P)) &
(is_goal (all P)) :-
       pi x\ (term x => (is_goal (P x))).
```

Notice also the use of β-conversion to realize substitution

Is GENERIC Really Needed for the Recursion?

Can a designated constant `dummy` be used for lowering the type?

For example, how about

```
is_goal (all P) :- is_goal (P dummy).
```

Two reasons why this is not a great idea

- The logical treatment provides advantages in reasoning

  For example, we get the following kind of property "for free" from the meta-theory of the specification logic

  > *A substitution instance of a goal* `(all P)` *is a goal*

- the *ad hoc* device just gets it wrong when the recursion involves synthesis as well

An Example Involving Synthesis of Structure

Consider converting formulas in classical logic to prenex normal form (pnf), i.e. to a form

$$\mathcal{Q}_1 x_1 \ldots \mathcal{Q}_n x_n M$$

where $\mathcal{Q}_i$ is either $\forall$ or $\exists$ and M is quantifier free

The conversion would use rules such as

> *The pnf of* $\forall x\ C$ *is* $\forall x\ B$ *if the pnf of* C *is* B

As a λProlog clause using the `dummy` constant

```
pnf (all C) (all B) :- pnf (C dummy) (B dummy).
```

But now consider the query

```
?- pnf (all x\ (p x)) F.
```


Mobility of Binders

Side conditions often involve the movement of binders from object-level to proof-level

In *hohh*, this mobility is paralleled by one from terms to formulas to derivations

For example, consider a sequent calculus theorem prover

```
type  proves  (list form) -> form -> o.
proves Gamma tt.
proves Gamma F :- member F Gamma.
proves Gamma (F1 && F2) :-
        proves Gamma F1, proves Gamma F2.
proves Gamma (F1 ==> F2) :- proves (F1::Gamma) F2.
 ...
proves Gamma (some P) :- proves Gamma (P T).
proves Gamma (all P) :- pi x\ proves Gamma (P x).
```

The dynamic growth of signatures and restrictions in instantiations realizes derivation level binding

Representing Typed Lambda Terms

The following λProlog signature defines a representation for types and terms

```
kind  ty, tm  type.

type   int   ty.
type   -->   ty -> ty -> ty.
infix  -->   3.

type  app  tm -> tm -> tm.
type  abs  ty -> (tm -> tm) -> tm.
```

Thus, (λx : *int* => λy : *int*→*int* => *y x*) would be represented as

```
(abs int (x\
   (abs (int --> int) (y\ (app y x)))))
```

Note: meta-language types *do not* reflect object-language ones

The preferred approach captures such properties in relations instead

Encoding Evaluation and Typing

The following λProlog program constitutes a specification of call-by-name evaluation and typing

```
type  eval  tm -> tm -> o.
type  of    tm -> ty -> o.

eval (abs T R) (abs T R).
eval (app T1 T2) V :-
      eval T1 (abs R), eval (R T2) V.

of (app T1 T2) Ty2 :-
   of T1 (Ty1 --> Ty2), of T2 Ty1.
of (abs Ty1 R) (Ty1 --> Ty2) :-
   pi x\ (of x Ty1) => (of (R x) Ty2).
```

Notice the use of β-conversion to realize substitution and the treatment of recursion

The latter will give us substitution theorems for free in reasoning

The L_λ Sub-Language of *hohh*

Quantifiers can be essentially universal or existential:

- Goals appear positively and program clauses negatively
- Moving to the left of an implication flips polarity
- Moving to a negative context changes existentials to universals and vice versa

The paradigms discussed almost always yield λProlog programs satisfying the following restriction:

> *Essentially existentially quantified variables appear applied only to distinct λ-bound variables or variables essentially universally bound within their scope*

For example, consider

```
pnf (all C) (all B) :- pi x\ pnf (C x) (B x).
```

The (dynamic) L_λ language is λProlog when it satisfies this property (dynamically)

Interesting Properties of L_λ

- Equations that arise in the L_λ setting have unique solutions up to renaming; e.g. contrast the following:

  ```
  ?- pi g\ pi a\ sigma F\ (F a) = (g a a).
  ?- pi g\ sigma F\ pi a\ (F a) = (g a a).
  ?- pi g\ sigma F\ pi a\ (G a a) = (g a a).
  ```

- β-redexes that arise from substitution for essential existential variables have a benign structure

  Specifically, contracting them does not create new redexes

  This style of redexes are called β_0-redexes

- Substitution is one case that violates this restriction, e.g.

  ```
  proves Gamma (some P) :- proves Gamma (P T).
  ```

  However, here the restriction will typically be met dynamically

Examples of Specifications in λProlog

Process Expressions in the π-Calculus

A language for modelling processes that interact using names

Two important syntactic categories: *names* and *processes*

The process expressions in the finite π-calculus:

$$P \quad ::= \quad 0 \mid P \,|'\, P \mid P + P \mid x(y).P \mid \bar{x}y.P \mid [x = y].P \mid \tau.P \mid (y)P$$

Here x and y represent names

The intended meaning of the various expressions

- 0 is the null process, $|$ and $+$ stand for parallel composition and choice, $\tau.P$ is a process that can evolve silently to P
- $x(y).P$ can accept a name along channel x and transform into P with y replaced by this name
- $\bar{x}y.P$ can evolve into P by outputting y along channel x
- $[x = y].P$ can become P if x and y are equal
- $(y)P$ represents the restriction of the name y to P

Representing Process Expressions in λProlog

Declarations provide the framework for an encoding

```
kind name          type.
kind proc          type.

type null          proc.
type plus, par     proc -> proc -> proc.
type in            name -> (name -> proc) -> proc.
type out, match    name -> name -> proc -> proc.
type taup          proc -> proc.
type nu            (name -> proc) -> proc.
```

Note the representation of input processes and restriction

E.g., the process $(b(z).P \mid (y)\bar{b}y.Q)$ will be represented by

```
(par (in b (z\ P')) (nu (y\ (out b y Q'))))
```

where `b` is declared to be a `name` and `P'` and `Q'` are encodings of P and Q

Transitions in the π-Calculus

The operational semantics is given by inference rules defining judgements of the form $P \xrightarrow{A} Q$

To be read as "process P evolves into Q via action A"

Note: This is what is often referred to as "small-step" semantics

There are four kinds of actions

τ	the *silent* action
$x(y)$	the (bound) input action
$\bar{x}y$	the free output action
$\bar{x}(y)$	the bound output action

The last differs from the third in that it emits a private name (bound by a restriction) on the channel x

The bound actions involve side conditions and formalizing their interaction correctly requires some care

Bound Actions and their Interaction

The rules of interest are the following

$$\frac{}{x(z).P \xrightarrow{x(w)} P\{w/z\}}\ \text{INPUT-ACT}, w \text{ not free in } (z)P$$

$$\frac{P \xrightarrow{\bar{x}y} P'}{(y)P \xrightarrow{\bar{x}(w)} P'\{w/y\}}\ \text{OPEN}, x \neq y, w \text{ not free in } (y)P$$

$$\frac{P \xrightarrow{\bar{x}(w)} P' \quad Q \xrightarrow{x(w)} Q'}{P|Q \xrightarrow{\tau} (w)(P'|Q')}\ \text{CLOSE} \qquad \frac{P \xrightarrow{\bar{x}(w)} P' \quad Q \xrightarrow{x(w)} Q'}{Q|P \xrightarrow{\tau} (w)(Q'|P')}\ \text{CLOSE}$$

Here

- The OPEN rule "opens" a scope represented by a restriction operator
- The CLOSE rule closes the corresponding scope after interaction with an input action

For example, consider the evolution of $b(z).P \mid (y)\bar{b}y.Q$

Encoding Transition Rules in λProlog

The key idea in capturing the interaction of bounded actions

Bounded actions will produce abstracted processes that can be combined by being fed a common name

To realize this idea, we encode transitions via *two* predicates

Specifically, we will use the following λProlog declarations

```
kind action       type.
type tau          action.
type up, dn       name -> name -> action.

type one    proc -> action -> proc -> o.
type onep   proc -> (name -> action)
                 -> (name -> proc) -> o.
```

Some actions will yield clauses for only one predicate, some will yield clauses for both

Encoding Transition Rules in λProlog (Continued)

Using the signature, the actions $x(w)$ and $\bar{x}(w)$ will be represented by `(dn x)` and `(up x)` respectively

The (bound) name w will become an abstraction over the resulting process

The clauses for the INPUT-ACT, OPEN and CLOSE rules

```
onep (in X M) (dn X) M.
onep (nu P)   (up X) P' :-
   pi y\ one  (P y) (up X y) (P' y).
one  (par P Q) tau (nu y\ par (P' y) (Q' y)) &
one  (par Q P) tau (nu y\ par (Q' y) (P' y)) :-
            onep P (up X) P', onep Q (dn X) Q'.
```

The use of abstracted processes ensures all the side conditions are met

The clause for the CLOSE action applies these abstractions to a common name to realize the combination

Encoding Transition Rules in λProlog (Continued)

To complete the picture, we consider the encoding of some other typical rules

- The free output action will yield a clause only for `one`

$$\frac{}{\bar{x}y.P \xrightarrow{\bar{x}y} P}\ \text{OUTPUT-ACT}$$

```
one  (out X Y P)  (up X Y) P.
```

- Bound input and free output actions can also interact

$$\frac{P \xrightarrow{\bar{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid (Q[y/z])} \qquad \frac{P \xrightarrow{\bar{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{Q \mid P \xrightarrow{\tau} (Q[y/z]) \mid P}\ \text{COM}$$

```
one  (par P Q) tau (par S (T Y)) :-
    one  P (up X Y) S, onep Q (dn X) T.
one  (par P Q) tau (par (S Y) T) :-
    onep P (dn X) S, one  Q (up X Y) T.
```

Encoding Transition Rules in λProlog (Continued)

- The "congruence" over a choice must be reflected on both `one` and `onep`

$$\text{SUM}: \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \frac{P \xrightarrow{\alpha} P'}{Q + P \xrightarrow{\alpha} P'}$$

```
one  (plus P Q) A P' :- one  P A P'; one  Q A P'.
onep (plus P Q) A P' :- onep P A P'; onep Q A P'.
```

- A similar kind of congruence applies to parallel composition

$$\text{PAR}: \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \frac{P \xrightarrow{\alpha} P'}{Q \mid P \xrightarrow{\alpha} Q \mid P'}$$

```
one  (par  P Q) A (par P' Q) &
one  (par  Q P) A (par Q P') :- one P A P'.
onep (par  P Q) A (y\ par (P' y) Q) &
onep (par  Q P) A (y\ par Q (P' y)) :- onep P A P'.
```


Using the π-Calculus Specifications

- Specifications can be used to experiment with the behaviour of described systems

  λProlog allows the specifications to be animated, facilitating, for example
  - the inspection of one step transitions from processes
  - the examination of traces

  Here we are talking about the *may* behaviour of systems
- We can also consider the use of the specifications to *analyze* the behaviour of systems

  For example, showing that a process cannot make some transitions, showin similarity between processes, etc

  However, for this we also need methods for talking about the *only things* a process can do, i.e. its *must* behaviour

  To do this correctly, we will need a framework that treats the "only if" aspect of logic specifications

Encoding Functional Programs in λProlog

We have already seen how to use λ-tree syntax to represent types and (untyped) lambda terms

It is easy to extend this setup to obtain a framework for encoding arbitrary functional programs, e.g.

```
type  bool  ty.
type  lst   ty -> ty.

type  i           int -> tm.
type  tt, ff      tm.
type  nil, cons   tm.
type  sum         tm.
type  cond        tm -> tm -> tm -> tm.
type  fix         (tm -> tm) -> tm.
```

Other combinators, functions, data types, can also be encoded

If the types language has explicit polymorphism, this can be encoded using λ-tree syntax

Typing and Evaluation for Functional Programs

The earlier judgements for typing and evaluation can be easily extended to accommodate the new constructs

For example, consider

```
(of  tt  bool) & (of ff bool).
of (i I) int.
of sum (int --> int --> int).
...
of  (cond C T E) A :-
     of C bool, of T A), of E A.
of (fix E) Ty :-
      pi x\ (of x Ty => of (E x) Ty).

...
eval (app (app sum E1) E2) (i I) :-
    eval E1 (i I1), eval E2 (i I2), I is I1 + I2.
eval (cond C T _) V :- eval C tt, eval T V.
eval (cond C _ E) V :- eval C ff, eval E V.
eval (fix E) V :- eval (E (fix E)) V.
```

Small-Step Evaluation via Evaluation Contexts

Lambda terms provide an elegant means for characterizing evaluation contexts in computation via repeated rewriting

```
type val, non_val, redex  tm -> o.
type reduce, eval   tm -> tm -> o.
type context        tm -> (tm -> tm) -> tm -> o.

context R (x\ x) R :- redex R.
context (cond M N P) (x\ cond (E x) N P) R :-
  non_val M, context M E R.
context (app M N) (app (x\ (E x)) N) R :-
  non_val M, context M E R.
context (app V M) (x\ (app V (E x)) R :-
  val V, non_val M, context M E R.

eval V V :- val V.
eval M V :- context M E R, reduce R N, eval (E N) V.
```

Here `non_val`, `val`, and `redex` recognize non-values, values and redexes at the root and `reduce` contracts redexes

Recognizing Tail Recursive Structure

In an expression of the form `(fix (f\ F))` we have to check that usage of `f` in `F` is suitably restricted

Assume that the signature of the language has been reified via the predicate `term`

```
type  tr, fn, trabs, headrec, trbody  tm -> o.

tr (fix M) :- pi F\ ((fn F) => (trabs (M F))).

trabs (abs R) :- pi X\ ((term X) => (trabs (R X))).
trabs R :- trbody R.

trbody (cond M N P) :- term M, trbody N, trbody P.
trbody M :- term M ; headrec M.

headrec (app M N) :- (fn M ; headrec M), term N.
```

Recursion over binding structure allows for a generalization of template matching a la Burstall and Darlington (Huet and Lang)

Binding Sensitive Analysis of Functional Programs

Some other examples in the book where λ-tree syntax is used to advantage:

- Partial evaluation, reductions under abstractions

  AUGMENT and GENERIC provide logical support for descent inside the body of an (object-language) abstraction

- Continuation-passing style transformation of programs

  An approach that uses λ-calculus equivalences to correctly transform programs to a tail recursive form

Specifying a Natural Deduction Calculus

We will think of formalizing this as a typing calculus

A sampling of the proof terms that might be used in this pursuit

```
kind  proof type.

type imp_i  (proof -> proof) -> proof.
type imp_e  form -> proof -> proof -> proof.
type or_i1,
     or_i2  proof -> proof.
type or_e   form  -> form -> proof
                  -> (proof -> proof)
                  -> (proof -> proof) -> proof.
type all_e  term  -> (term -> form) -> proof -> proof.
type all_i  (term -> proof) -> proof.
type some_e (term -> form) -> proof ->
            (term -> proof -> proof) -> proof.
type some_i term  -> proof -> proof.
```

Typically we store as much information in the proof term as is necessary to make type checking well-behaved

Specifying a Natural Deduction Calculus (Continued)

The inference rules of the calculus are then formalized as the definition of a predicate that relates proof terms and formulas

```
type  #  proof -> form -> o.     infix # 2.

(imp_i Q) # (A ==> B) :- pi p\ (p # A) => ((Q p) # B).
(imp_e A P1 P2) # B   :- (P1 # A), (P2 # (A ==> B)).
(or_i1 P) # (A !! B)  :- P # A.
(or_i2 P) # (A !! B)  :- P # B.
(or_e A B P Q1 Q2) # C :-
   (P # (A !! B)),
   (pi p1\ (p1 # A) => ((Q1 p1) # C)),
   (pi p2\ (p2 # B) => ((Q2 p2) # C)).
(all_i Q) # (all A)   :- pi y\ (Q y) # (A y).
(all_e T A P) # (A T) :- (P # (all A)).
(some_i T P) # (some A) :- P # (A T).
(some_e A P1 Q) # B :-
   (P1 # (some A)),
   pi y\ pi p\ (p # (A y)) => ((Q y p) # B).
```


Specifying versus Implementing Proof Systems

- Declarative specifications like that for the natural deduction calculus are well-suited for meta-theoretic reasoning
- Sometimes, such calculi can also be structured to provide a basis for proof search

  E.g, Dyckhoff's calculus and calculi that integrate focusing
- However typically such calculi, together with depth-first exploration, are not good for proof search
- Tactics and tacticals based approaches provide a means to "bake your own" control regime
  - Inference rules are encoded as standalone goal transformers called tactics
  - Tacticals provide a framework for combining such tactics into larger units
  - In the λProlog setting, definitions of tacticals use predicate variables

Unification of Lambda Terms

The Need for Unification

Recall the $\forall L$ and $\exists R$ rules in the reduced proof system

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

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

These rules require us to guess a suitable term before we have any information of how to do this

These rules are therefore not quite suitable for proof search

The standard technique is to overcome this obstacle is to delay such choices using *logic variables* and unification

Logic Variables and Unification

Logic variables, denoted by tokens starting with uppercase letters, are place-holders for actual terms to be decided later

Using them, the $\forall L$ and $\exists R$ rules become

$$\frac{\Sigma; \mathcal{P} \vdash B[X/x] \qquad X \text{ is a } \Sigma\text{-term}}{\Sigma; \mathcal{P} \vdash \exists x\ B}\ \exists R$$

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

Actual substitutions will be dictated by the needs of the *init* rule

$$\frac{}{\Sigma; \mathcal{P} \xrightarrow{A'} A}\ \textit{init}, A \text{ and } A' \text{ unify}$$

Some important points to note

- substitutions generated by *init* must be applied to the entire derivation
- Instantiations must respect signature constraints

Unification Under a Quantifier Prefix

A nice conceptual treatment of signature constraints results from moving quantifiers to the derivation level

These quantifiers then become a prefix to unification problems

Specifically, a unification problem has the structure

$$\mathcal{Q}_1 x_1\ \ldots\ \mathcal{Q}_n x_n\ [t_1 = s_1 \wedge \ldots \wedge t_m = s_m]$$

where $\mathcal{Q}_i$ is $\forall$ or $\exists$

Such a problem is solved by a substitutions for the existentially quantified variables that respects quantifier scopes

For uniformity, top-level constants are represented in this framework by outermost universal quantifiers

Unification Under a Quantifier Prefix (Continued)

The prefixed form represents the unification problem along only *one branch* of a proof tree

Note that prenexing is not a sound principle:

$$\forall y\ \exists x\ [y = y \wedge x = x] \quad \text{and} \quad (\forall y\ [y = y]) \wedge (\exists x\ [x = x])$$

do not have the same solutions

it is therefore unclear how to transform a problem with the structure

$$(\forall x\ \exists z\ \ldots) \wedge (\forall y\ \exists w\ \ldots)$$

into a prefixed form that preserves the solutions

Thus, we will in general be concerned with solving a *conjunction* of unification problems with shared prefixes

Unifiers versus Solutions to Unification Problems

To be logically correct, solutions must be *closed*, i.e., they must only contain suitable universally quantified variables

However, often it is better to stop short of exhibiting actual solutions

For example, consider

$$\forall a\ \forall b\ \exists x\ \exists y\ x = y$$

Here x and y must be identical *and* must be one of a or b

However, making the last choice now is premature

Substitutions that solve all the equations but do not necessarily produce actual terms are called *unifiers*

We will focus on finding unifiers, noting that eventually exhibiting an inhabitant is important for the logic

Simplifying the Quantifier Prefix via Raising

Often we will treat unification problems in which the prefix has a $\forall\exists\forall$ or even a $\exists\forall$ form

Any arbitrary unification problem can be put in this form by a technique called *raising*

Specifically, raising moves existential quantifiers using the following kind of transformation

$$\forall x\ \exists y\ B \quad \rightsquigarrow \quad \exists f\ \forall x\ B[(f\ x)/y]$$

The end point for such a transformation is a $\exists\forall$ prefix

However, it is often convenient to leave the outermost universal quantifiers representing top-level constants implicit

The Correctness of Raising

Raising is justified purely on proof-theoretic principles

Proposition

There is a one-to-one correspondence between the solutions to $\forall x\ \exists y\ B$ and $\exists f\ \forall x\ B[(f\ x)/y]$

Proof

We show how to go to and from solutions to the two problems

- Suppose that $\{y \mapsto t\}$ solves the first problem
  Consider the substitution $\lambda x\ t$ for f
- Suppose $\{f \mapsto \lambda z\ t\}$ solves the second problem
  Consider the substitution of $t[x/z]$ for y

Note also that a back-and-forth transformation preserves the original substitution

Note: Raising preserves the property of being in L_λ

Comparison with Skolemization

Dual to raising that moves existential quantifiers inwards using

$$\exists x\ \forall y\ P \rightsquigarrow \forall f\ \exists x\ P[(f\ x)/y]$$

Intuitively, the dependency of $(f\ x)$ on x prevents it from appearing in a substitution for x

Used in settings where there are no higher-order variables

However, there are problems with Skolemization

- Assume $z : (a{\to}b){\to}a$, $x : a$, $y : b$, $f : a{\to}b$ and contrast

  $$\forall z\ \exists x\ \forall y\ \top \quad \text{with} \quad \forall z\ \forall f\ \exists x\ \top$$

  The second has a solution whereas the first does not
- Assume $z : a, g : b{\to}a, x : a, y : b, f : a{\to}b$ and contrast

  $$\forall z\ \forall g\ \exists x\ \forall y\ \top \quad \text{with} \quad \forall z\ \forall g\ \forall f\ \exists x\ \top$$

  The first has one solution whereas the second has an infinite number, i.e., it is not clear how to relate solutions

A Simplified Form for Equations

Since we solve equations modulo λ-conversion, we can assume equations have the form

$$\lambda x_1\ \ldots \lambda x_n\ (h_1\ t_1\ \ldots\ t_n) = \lambda x_1\ \ldots \lambda x_n\ (h_2\ s_1\ \ldots\ s_k)$$

where the body has atomic type and h_1 and h_2 are existential, universal or lambda-bound variables

Moreover, using the fact that the solutions for

$$\overline{\mathcal{Q}}\,[\lambda x\ t = \lambda x\ s] \quad \text{and} \quad \overline{\mathcal{Q}}\,\forall x\ [t = s]$$

are identical, we can assume that the binders are empty

We will base our analysis only on problems in this form

Also, we will use the terminology of *rigid-rigid*, *rigid-flexible* and *flexible-flexible* pairs

Properties of Higher-Order Unification

The general problem possesses many properties that are at least daunting from a computational perspective

- We cannot guarantee that there is *one* unifier that covers all others, e.g. consider

  $$\forall a_i \; \forall g_{\; i \to i \to i} \exists F_{i \to i} \; [(F \; a) = (g \; a \; a)]$$

  Four incomparable unifiers exist for this problem
- "Complete sets of unifiers" may be infinite, e.g. consider

  $$\forall u_{i \to i} \; \exists F_{i \to i} \; \forall w_i \; [u \; (F \; w) = F \; (u \; w)]$$

  Any substitution of the form $\{F \mapsto \lambda y (u \; \ldots \; (u \; y) \ldots)\}$ is a unifier
- Unifiers cannot always be generated in a non-redundant way

Properties of Higher-Order Unification (Continued)

- Determining whether or not there are unifiers is an undecidable problem

  Finding integer solutions to Diophantine equations can be encoded as a higher-order unification problem

Despite all these "bad" properties, higher-order unification has been found useful in systems like Isabelle, Twelf and λProlog

There are several observations that help resolve the paradox

- A procedure can be described to at least determine unifiability
- The problems that arise in practical settings are well behaved
- In fact many applications can be limited to a specially well behaved class of problems

Simplifying Rigid-Rigid Pairs

A rigid-rigid equation has the form

$$(c_1 \; t_1 \; \ldots \; t_n) = (c_2 \; s_1 \; \ldots \; s_m)$$

where c_1 and c_2 are universally quantified variables

Note that substitutions cannot change the heads of either term

Thus, it is not difficult to see

- the problem can have a unifier only if $c_1 = c_2$ and $m = n$
- its unifiers in this case are identical to those for

  $$t_1 = s_1 \wedge \ldots \wedge t_n = s_n$$
- Applying this transformation replaces an equation by a finite number of new ones in which the terms are smaller

In short, we can "eliminate" rigid-rigid pairs via a terminating process akin to term reduction for first-order terms

Processing Flexible-Rigid Pairs

Assuming symmetry for the equality symbol, these pairs have the form

$$(F \; t_1 \; \ldots \; t_n) = (c \; s_1 \; \ldots \; s_m)$$

where F is existentially quantified and c is universally quantified

Clearly a unifier must substitute for F in such a way that the head of the left term becomes c

There are really only two possible ways that this can happen

- the substitution for F directly introduces c in the head
- the substitution projects onto one of the arguments of F in the term that then does the rest

We think of generating this substitution in two kinds of incremental steps called *imitation* and *projection*

Note: One of these steps can be shown to get us closer if a unifier exists but there can also be a loop if there is no unifier

The Imitation Substitution

The pair that we are trying to "solve" has the form

$$(F\ t_1\ \ldots\ t_n) = (c\ s_1\ \ldots\ s_m)$$

where the type of F is $\alpha_1 \to \ldots \to \alpha_n \to \beta$ for β an atomic type

Note that an imitation substitution for F can exist only if c is quantified before F

To be truly incremental, such a substitution must only fix the head of the F, leaving all other decisions to later

Let the type of c be $\gamma_1 \to \cdots \to \gamma_m \to \beta$

Let H_i be a fresh variable of type $\alpha_1 \to \cdots \to \alpha_n \to \gamma_i$ for $i \leq m$

The imitation substitution for F, if it exists, then has the form

$$\lambda x_1\ \ldots\ \lambda x_n\ (c\ (H_1\ x_1\ \ldots\ x_n)\ \ldots\ (H_m\ x_1\ \ldots\ x_n))$$

with the proviso that we introduce existential quantifiers for H_i adjacent to the one for F

The Projection Substitutions

Again, the pair that we are trying to "solve" has the form

$$(F\ t_1\ \ldots\ t_n) = (c\ s_1\ \ldots\ s_m)$$

where the type of F is $\alpha_1 \to \ldots \to \alpha_n \to \beta$ for β an atomic type

Let α_i be $\gamma_1 \to \cdots \to \gamma_k \to \delta$, for δ an atomic type

A projection onto the ith argument is possible only if $\delta = \beta$

Again, we want a truly incremental substitution that does not compromise any future decisions

Let H_i be a fresh variable of type $\alpha_1 \to \cdots \to \alpha_n \to \gamma_i$ for $i \leq m$

The ith projection substitution for F, if it exists, is

$$\lambda x_1\ \ldots\ \lambda x_n\ (x_i\ (H_1\ x_1\ \ldots\ x_n)\ \ldots\ (H_k\ x_1\ \ldots\ x_n))$$

with the proviso that we introduce existential quantifiers for H_i adjacent to the one for F

Iteration of Simplification and Substitution Phases

The application of imitation or projection substitutions can produce new rigid-rigid pairs, thereby leading to an iteration

The iteration process can be organized into a structure called a *matching tree*, e.g. consider $\forall g_{\ i \to i \to i} \forall a_i\ \exists F_{i \to i}\ (F\ a) = (g\ a\ a)$

Some properties of matching trees

- simple types ensure the tree is finitely branching
  Note, though, that the tree structure is sensitive to typing
- "terminated" branches correspond to failure or (at most) flexible-flexible unification problems
- Some branches can go on for ever; for example, consider the imitation substitution $\lambda x\ u\ (H\ x)$ for F for
  $$\forall u_{i \to i}\ \exists F_{i \to i}\ \forall w_i\ [u\ (F\ w) = F\ (u\ w)]$$
  This yields a problem that has an identical structure

Properties of Flexible-Flexible Unification Problems

- Providing a unifier for such problems is easy
  A pair of the form
  $$(F\ t_1\ \ldots\ t_n) = (G\ s_1\ \ldots\ s_m)$$
  can be solved by picking a canonical variable of base type and projecting F and G onto that
- Providing a solution to such a problem requires showing habitation of the base type but this is decidable
- Enumeration of *all* unifiers can, however, be unstructured
  In fact, this part of the search cannot even be guaranteed to be non-redundant

Interestingly, solving flexible-flexible pairs becomes much more structured in the setting of the L_λ language

In fact, in this setting, there are even *most general* unifiers

Higher-Order Pattern Unification

Unification problems that arise in the L_λ setting are called higher-order pattern unification problems

We will observe that under the L_λ restrictions

- at most one branch in the matching tree can lead to success
- the flexible-flexible problems have a most general unifier
- With an additional "occurs-check" when processing flexible-rigid pairs, termination can be assured

Thus, higher-order pattern unification is decidable and admits the MGU property

The Matching Tree in the HOPU Setting

Flexible-rigid pairs in this context have the form

$$(F\ c_1\ \ldots c_n) = (c\ t_1\ \ldots t_m)$$

where $c_1, \ldots, c_n$ are distinct variables quantified universally within the scope of the quantifier for F

Consider, then, the possibilities for c

- c is quantified within the scope of the quantifier for F
  Thus
  - An imitation substitution cannot exist
  - At most one of the projections will be successful
- c is quantified outside the scope of the quantifier for F
  In this case only the imitation substitution will be applicable

It follows easily that there can be at most one success branch in the tree

Flexible-Flexible Problems in the HOPU Setting

One case of a flexible-flexible equation to consider is

$$(F\ c_1\ \ldots c_n) = (F\ d_1\ \ldots d_m)$$

where $c_1, \ldots, c_n$ and $d_1, \ldots, d_m$ are distinct variables quantified universally within the scope of the quantifier for F

Here the *same* substitution will be applied to the two sides

Thus, a c or d variable can appear in a common instance of the two terms only if $c_i = d_i$

Let $a_1, \ldots, a_k$ be a listing of the cs such that $c_i = d_i$

Then an MGU for this problem is

$$\{F \mapsto \lambda c_1\ \ldots \lambda c_n\ H\ a_i\ \ldots\ a_k\}$$

where H is a new existentially quantified variable

Flexible-Flexible HOPU Problems (Continued)

The other case of flexible-flexible equations to consider is

$$(F\ c_1\ \ldots c_n) = (G\ d_1\ \ldots d_m)$$

where $c_1, \ldots, c_n$ ($d_1, \ldots, d_m$) are distinct variables quantified universally within the scope of the quantifier for F (resp G)

Here a c or d variable can appear in a common instance of the two sides only if it appears *both* in the cs and in the ds

Let $a_1, \ldots, a_k$ be a listing of such variables

An MGU for the problem then is the substitution

$$\begin{array}{ll}\{ & F \mapsto \lambda c_1\ \ldots \lambda c_n\ (H\ a_1\ \ldots\ a_k), \\ & G \mapsto \lambda d_1\ \ldots \lambda d_m\ (H\ a_1\ \ldots\ a_k) \quad \}\end{array}$$

where H is a new existentially quantified variable

Occurs-Check and Termination

Because substitutions produce only β_0-reductions, there is hope that the the matching tree will be finite

However, loops can still exist because of the flexible head can appear in the rigid term in a flexible-rigid pair, e.g. consider

$$\forall f_{i \to i}\ \exists X_i\ X = (f\ X)$$

To overcome this problem, we can include an "occurs-check" in the processing of flexible-rigid pairs

Fail if the flexible head appears in the rigid term

Note that this simple test *does not* work for the general higher-order unification setting

To illustrate the algorithm, consider the following problems

$$\forall f\ \forall g\ \exists U\ \exists V\ \forall w\ \forall x\ \forall y\ [(f\ (U\ x\ y)) = (f\ (g\ (V\ y\ w)))]$$
$$\forall f\ \forall g\ \exists U\ \exists V\ \forall w\ \forall x\ \forall y\ [(U\ x\ y) = (g\ (U\ y\ w))]$$
$$\forall f\ \forall g\ \exists U\ \exists V\ \forall w\ \forall x\ \forall y\ [(U\ x\ y) = (g\ w)]$$

Some Additional Observations about HOPU

- The incremental generation of substitutions leads to structure traversal that is highly redundant

  Instead, we can factor the process into two major steps

  - Simplification, that walks over the top-level rigid structure to at most leave flex-rigid and flex-flex pairs
  - Variable elimination, that takes a variable and a vector of arguments to try to generate a complete substitution

- In this form, the algorithm is similar to the first-order one except for a more involved variable elimination

  In fact, a nearly linear algorithm can be provided

- Preprocessing via raising can be practically costly

  Raising can be delayed and given an "on-the-fly" treatment

All the implementations discussed in this course use the first and third ideas

Reasoning about Specifications

Specifications as Definitions

We want to treat program clauses as *definitions* of relations

Computational interpretation gives only one half of this reading

For example, given

```
append nil L L.
append (X::L1) L2 (X::L3) :- append L1 L2 L3.
```

we can derive goals like

```
?- append (1::nil) (2::nil) (1::2::nil).
```

but not

```
?- append (1::nil) nil nil => false.
```

Deducing negative properties from definitions can be important to meta-theoretic reasoning

Of course, validating negative properties depends on a "closed-world" assumption

Extensional Interpretation of Universals

The specification logic treats universals intensionally, i.e. proof must be generic and independent of domain

If we interpret specifications in an "if and only if" fashion, extensional treatment may also be useful

For example, given

```
nat z.
nat (s N) :- nat N.

plus z L L.
plus (s N) M (s P) :- plus N M P.
```

we want to be able to prove not only

```
pi L\ (plus z L L)
```

but also the following

```
pi L\ (nat L) => (plus L z L)
```


Proving Subject Reduction (Case Study)

Call-by-name evaluation and typing were rendered into *hohh* specifications as follows

```
eval (abs T R) (abs T R).
eval (app T1 T2) V :-
      eval T1 (abs R), eval (R T2) V.

of (app T1 T2) Ty2 :-
   of T1 (Ty1 --> Ty2), of T2 Ty1.
of (abs Ty1 R) (Ty1 --> Ty2) :-
   pi x\ (of x Ty1) => (of (R x) Ty2).
```

In this setting, we want to be able to prove

$$\forall \texttt{e}\, \forall \texttt{v}\, \forall \texttt{ty}\, ((\texttt{eval e v}) \wedge (\texttt{of e ty}) \supset (\texttt{of v ty}))$$

This reasoning task depends on both the closed world reading and the extensional treatment of universals

Informal Proof of Subject Reduction

At the outermost level, we use an induction on the derivation of `eval e v` in the specification logic

Then we consider the cases for the clause used at the end of this derivation

First Case:

The clause used at the end is

```
eval (abs T R) (abs T R).
```

Here *e* and *v* are identical and it follows immediately that the same typing derivations hold for both

Note: The case analysis is leading us to consider instantiations for the eigenvariables *e* and *v*

Informal Proof of Subject Reduction (Second Case)

Here, `e` must have the form `(app e1 e2)` and there must be shorter derivations for

(1) `(eval e1 (abs t r))` and (2) `(eval (r e2) v)`

Since `(of (app e1 e2) ty)` is also derivable

(3) `(of e1 (ty1 --> ty))` is derivable for some `ty1`

(4) `(of e2 ty1)` is derivable for the same `ty1`

By induction on (1) and using (3), it follows that `(of (abs ty1 r) (ty1 --> ty))` is derivable

But then `(pi x\ (of x ty1) => (of (r x) ty))` must be derivable in the specification logic

Using (4) and the instantiation principle for that logic, `(of (r e2) ty)` must be derivable

Using the induction hypothesis now with (2) it follows that `(of v ty)` is derivable

Desiderata for Reasoning Logic

To be able to formalize the style of reasoning illustrated by the subject reduction argument, we need a logic that

- Has the capability to embed definitions in a way that permits case analysis based on a closed world assumption
- Understands the properties of the specification logic and allows these to be used in reasoning
- Allows for the instantiation of eigenvariables to support extensional interpretations of universal quantifiers
- Supports inductive (and perhaps also co-inductive) arguments based on definitions

To realize these requirements, we will introduce a logic of (fixed-point) definitions

Note: We will implicitly assume a simply typed setting with no quantification over predicate types

Universal versus Generic Quantification

The reasoning logic needs also a form of universal quantification whose every instance is proved uniformly

- Such a "generic" reading is what was intended of the universal quantifier in the specification logic, e.g. consider

```
of (abs Ty1 R) (Ty1 --> Ty2) :-
   pi x\ (of x Ty1) => (of (R x) Ty2).
```

- Actually, we will also need a *distinctness of binding* property for our generic quantifiers

  For example, in most logics, the following is derivable

$$\forall x\, \forall y\, (P\ x\ y) \supset \forall z\, (P\ z\ z)$$

  This can be problematic if universal quantification is used in treating names, e.g. in an encoding of the π-calculus

Comment: the latter issue becomes relevant only when we also have to reason about formulas (premises) in *negative* contexts

The ∇ (Nabla) Quantifier

This is a new quantifier for representing generic quantification

The informal meaning of $\nabla x\ B$:

> *B has a proof that does not depend on the form of x but the proof can use its freshness*

In a minimal formalization of this idea, judgements (i.e. formulas) are decorated with a local context of parameters

The rules for the ∇ quantifier then are the following

$$\frac{\Sigma; \Gamma, ((a, \sigma) \rhd B[a/x]) \vdash \Delta}{\Sigma; \Gamma, (\sigma \rhd \nabla x\ B) \vdash \Delta}\ \nabla\mathcal{L} \qquad \frac{\Sigma; \Gamma \vdash (a, \sigma \rhd B[a/x])}{\Sigma; \Gamma \vdash (\sigma \rhd \nabla x\ B)}\ \nabla\mathcal{R}$$

where a is a parameter that is new to the sequent

Also, local contexts are treated as binders: two (extended) judgements match if they do so under renaming

Note: We will consider only an intuitionistic version of the logic

Quantifier Rules with Local Contexts

The usual quantifier rules must now be adjusted to account for the scopes of ∇ quantifiers as well

For essential existential quantification, parameters from local contexts must be permitted in forming valid terms

$$\frac{\Sigma; \Gamma \vdash (\sigma \rhd B[t/x]) \qquad t \text{ is a } (\Sigma \cup \sigma)\text{-term}}{\Sigma; \Gamma \vdash (\sigma \rhd \exists x\ B)}\ \exists R$$

$$\frac{\Sigma; \Gamma, (\sigma \rhd B[t/x]) \vdash \Delta \qquad t \text{ is a } (\Sigma \cup \sigma)\text{-term}}{\Sigma; \Gamma, (\sigma \rhd \forall x\ B) \vdash \Delta}\ \forall L$$

Considering the local context in the instantiation/generalization term takes into account the ∇ quantifiers with larger scope

Quantifier Rules with Local Contexts (Continued)

For essential universal quantifiers, we must also take into account that eigenvariables may be instantiated later

To properly constrain such instantiations, we use raising

Formally, the rules become

$$\frac{h, \Sigma; \Gamma \vdash (\sigma \rhd B[(h\ \sigma)/x])}{\Sigma; \Gamma \vdash (\sigma \rhd \forall x\ B)}\ \forall R$$

$$\frac{h, \Sigma; \Gamma, (\sigma \rhd B[(h\ \sigma)/x]) \vdash \Delta}{\Sigma; \Gamma, (\sigma \rhd \exists x\ B) \vdash \Delta}\ \exists L$$

Here, h is a new eigenvariable and $(h\ \sigma)$ denotes $(h\ x_1, \ldots, x_n)$ if σ is $x_1, \ldots, x_n$

Raising introduces the permitted dependencies where eigenvariable instantiations will be limited to closed terms

Local Contexts and Propositional Rules

Essentially, the local context must distribute over the connective

$$\frac{}{\Sigma; \Gamma \vdash (\sigma \rhd \top)}\ \top R \qquad \frac{}{\Sigma; (\sigma \rhd \bot), \Gamma \vdash \Delta}\ \bot L$$

$$\frac{\Sigma; \Gamma, (\sigma \rhd B_1) \vdash \Delta \quad \Sigma; \Gamma, (\sigma \rhd B_2) \vdash, \Delta}{\Sigma; \Gamma, (\sigma \rhd B_1 \vee B_2) \vdash \Delta}\ \vee L$$

$$\frac{\Sigma; \Gamma \vdash (\sigma \rhd B_1)}{\Sigma; \Gamma \vdash (\sigma \rhd B_1 \vee B_2)}\ \vee R_1 \qquad \frac{\Sigma; \Gamma \vdash (\sigma \rhd B_2)}{\Sigma; \Gamma \vdash (\sigma \rhd B_1 \vee B_2)}\ \vee R_2$$

$$\frac{\Sigma; \Gamma, (\sigma \rhd B_1) \vdash \Delta}{\Sigma; \Gamma, (\sigma \rhd B_1 \wedge B_2) \vdash \Delta}\ \wedge L_1 \qquad \frac{\Sigma; \Gamma, (\sigma \rhd B_2) \vdash \Delta}{\Sigma; \Gamma, (\sigma \rhd B_1 \wedge B_2) \vdash \Delta}\ \wedge L_2$$

$$\frac{\Sigma; \Gamma \vdash (\sigma \rhd B_1) \quad \Sigma; \Gamma \vdash (\sigma \rhd B_2)}{\Sigma; \Gamma \vdash (\sigma \rhd B_1 \wedge B_2)}\ \wedge R$$

$$\frac{\Sigma; \Gamma \vdash (\sigma \rhd B_1) \quad \Sigma; \Gamma, (\sigma \rhd B_2) \vdash \Delta}{\Sigma; \Gamma, (\sigma \rhd B_1 \supset B_2) \vdash \Delta}\ \supset L$$

$$\frac{\Sigma; \Gamma, (\sigma \rhd B_1) \vdash (\sigma \rhd B_2)}{\Sigma; \Gamma \vdash (\sigma \rhd B_1 \supset B_2)}\ \supset R$$

The Identity Rules

These are the only rules in which we have to "match up" different formulas

Here, we implicitly use renaming by requiring the local context "binders" to be identical

$$\frac{}{\Sigma; \Gamma, (\sigma \rhd B) \vdash (\sigma \rhd B)}\ \mathit{init}$$

$$\frac{\Sigma; \Gamma_1 \vdash (\sigma \rhd B) \qquad \Sigma; \Gamma_2, (\sigma \rhd B) \vdash \Delta}{\Sigma; \Gamma_1, \Gamma_2 \vdash \Delta}\ \mathit{cut}$$

Notice though that

- the binders have to be of equal length, and
- occurrences in formula and location in the binder must be correlated

Adding Definitions to the Logic

Predicate constants will be interpreted via clauses of the form

$$\forall \vec{x}.H \stackrel{\triangle}{=} B$$

where H is a (rigid) atomic formula called the *head* of the clause, and B is a (first-order) formula called the *body*

These clauses seem similar to program clauses but there are some important differences

- Collections of clauses (definitions) will *parameterize* the logic and *will not* be part of a sequent
- The body is not syntactically limited in the way program clauses are
- The logic will give a set $\mathcal{D}$ of such clauses, called a definition, a stronger *fixed-point* interpretation
- To ensure consistency of such a logic, definitions will be required to satisfy some "stratification conditions"

A Stratification Condition for Definitions

Let lvl be a function that assigns natural numbers or "levels" to predicate constants

We extend this assignment into one for arbitrary formulas

- $lvl(\top) = lvl(\bot) = 0$
- $lvl(p\ t_1\ \ldots\ t_n) = lvl(p)$
- $lvl(B \wedge C) = lvl(B \vee C) = \max(lvl(B), lvl(C))$
- $lvl(B \supset C) = \max(lvl(B) + 1, lvl(C))$
- $lvl(\forall x\ B) = lvl(\exists x\ B) = lvl(\nabla x\ B) = lvl(B)$

A definition $\mathcal{D}$ is then *good* if it satisfies the following condition

> *There is some assignment lvl to predicates such that*
> $\forall \vec{x}.H \stackrel{\triangle}{=} B \in \mathcal{D}$ *implies* $lvl(H) \leq lvl(B)$

Essentially, this amounts to a monotonicity condition where the entire definition of a predicate is fixed in one go

Introduction Rules for Atomic Formulas

We must first make precise some notions needed to talk about matching the head of a definition with atomic judgements

Definition

The raising of $\forall x_1 \ldots \forall x_n\,.\,H \stackrel{\triangle}{=} B$ over σ is given as follows

$$\forall h_1 \ldots \forall h_n.\sigma \rhd (H\theta) \stackrel{\triangle}{=} \sigma \rhd (B\theta)$$

where $\theta = \{(h_1\ \sigma)/x_1, \ldots, (h_n\ \sigma)/x_n\}$ for new variables $h_1, \ldots, h_n$ of appropriate types

Definition

The application of a substitution θ to $x_1, \ldots, x_n \rhd B$, denoted by $(x_1, \ldots, x_n \rhd B)\theta$, is given as follows

$$(x_1, \ldots, x_n \rhd B)\theta = y_1, \ldots, y_n \rhd B' \quad \text{if} \quad (\lambda x_1 \ldots \lambda x_n\ B)\theta = \lambda y_1 \ldots \lambda y_n\ B'$$

Definition

We write $\Sigma\theta$ for the signature that results from Σ by removing variables in the domain of θ and adding those in its range

Introduction Rules for Atomic Formulas (Continued)

Given a set of clauses $\mathcal{D}$, let $\sigma \rhd \mathcal{D}$ denote the raising of each member of $\mathcal{D}$ over σ

Then the introduction rules for atomic formulas are the following

$$\frac{\Sigma; \Gamma \vdash (\sigma \rhd B)\theta}{\Sigma; \Gamma \vdash \sigma \rhd A}\ def\mathcal{R} \qquad \sigma \rhd A = H\theta, \forall \vec{x}.H \stackrel{\triangle}{=} B \in \sigma \rhd \mathcal{D}$$

$$\frac{\{\Sigma\theta; \Gamma\theta, (\sigma \rhd B)\theta \vdash C\theta \mid (\sigma \rhd A)\theta = H\theta, \forall \vec{x} H \stackrel{\triangle}{=} B \in \sigma \rhd \mathcal{D}\}}{\Sigma; \Gamma, \sigma \rhd A \vdash C}\ def\mathcal{L}$$

Comments

- $def\mathcal{R}$ has the flavour of backchaining, $def\mathcal{L}$ of case analysis
- $def\mathcal{L}$ considers substitutions for eigenvariables
- $def\mathcal{L}$ must consider all clauses and all substitutions
- For cut-free proofs, it suffices to consider only complete sets of unifiers

Using Definitions (Example)

Suppose that the set of definition clauses is

$$p\ a \stackrel{\triangle}{=} \top, \quad p\ b \stackrel{\triangle}{=} \top, \quad q\ a \stackrel{\triangle}{=} \top, \quad q\ b \stackrel{\triangle}{=} \top, q\ c \stackrel{\triangle}{=} \top$$

Then $\quad ; \vdash \cdot \rhd \forall x\ p\ x \supset q\ x \quad$ is derivable

In fact, the following is a derivation for it:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{; \cdot \rhd \top \vdash \cdot \rhd \top}}{; \cdot \rhd \top \vdash \cdot \rhd q\ a}\ def\mathcal{R} \quad \dfrac{\dfrac{}{; \cdot \rhd \top \vdash \cdot \rhd \top}}{; \cdot \rhd \top \vdash \cdot \rhd q\ b}\ def\mathcal{R}}{Y; \cdot \rhd p\ Y \vdash \cdot \rhd q\ Y}\ def\mathcal{L}}{; \vdash \cdot \rhd \forall x\ p\ x \supset q\ x}\ \forall R; \supset R$$

Notice that the $def\mathcal{L}$ rule instantiates an eigenvariable

Mixing Finite Success with Finite Failure

A possible style of reasoning in the logic of definitions

- Work as usual with a formula on the right of the sequent
- When an atom becomes available on the left
  - simplify it immediately to the $\top$, computing the corresponding substitution for the eigenvariables
  - prove the formula on the right for each of these substitutions

For this style to work, some requirements have to be satisfied

- Only invertible rules should be applicable on the left
- Every "solution path" for an atom on the left must be finite

Restricting formulas to be proved can yield the first property and limiting specifications yields the second

Mixing Finite Success with Finite Failure (Continued)

The property we want of formulas in the left of a sequent is obtained by a kind of stratification

Specifically, we identify two levels of formulas

Level 0: $G ::= \top \mid \bot \mid A \mid G \wedge G \mid G \vee G \mid \exists x\ G \mid \nabla x\ G$

Level 1: $D ::= \top \mid \bot \mid A \mid D \wedge D \mid D \vee D \mid G \supset D$
$\exists x\ D \mid \nabla x\ D \mid \forall x\ D$

Also there are restrictions on the Level 0 and Level 1 atoms:

- Level 0 atoms must be defined by level 0 formulas
- Level 1 atoms can be defined by level 0 or level 1 formulas

Notice that Level 0 formulas encompass goals in Horn clause logic and that level 1 formulas include level 0 ones

The end goal is always to prove a sequent of the form

$$; \vdash \cdot \rhd D$$

where D is a level 1 formula

Mixing Finite Success with Finite Failure (Continued)

An observation concerning sequents seen by the prover:

> *Only (decorated) G formulas appear on the left and all the rules applicable to them are invertible*

Also, there is a correspondence between uniform provability and the application of left rules to (decorated) G formulas

Thus, proof search for $; \vdash \sigma \rhd G \supset D$ can be structured as follows

Step 1 Run a logic programming interpreter with $\sigma \rhd G$, treating eigenvariables as logic variables

Step 2 Collect *all* answer substitutions in Step 1 and attempt to prove D under each

If there are *no* answers in Step 1, the prover succeeds immediately

The Bedwyr System

An implementation in OCaml of the described proof search strategy

The main ingredients of the implementation

- uses a logic programming interpreter on both the right and left sides of the sequent
- interprets eigenvariables as instantiatable on the left and produces all answers in a lazy stream-based manner
- treats local contexts via abstractions and uses higher-order pattern matching to match atoms and clause heads

Bedwyr has been used in several prototyping examples, e.g.

- checking (open) bisimilarity in the finite π-calculus
- identifying winning strategies in games
- reasoning about security protocols: the SPEC system `http://users.cecs.anu.edu.au/~tiu/spec/index.html`.

Limitations of Bedwyr

- Eigenvariables and logic variables used for full automation lead to incompleteness if present together

  For example, consider proving the formula

  $$\forall x\ \exists y\ (px \wedge py \wedge x = y \supset \bot)$$

  where p is defined as $\{pa \stackrel{\triangle}{=} \top, pb \stackrel{\triangle}{=} \top, pc \stackrel{\triangle}{=} \top\}$

  Completing this proof requires solving *disunification* problems: For each x, find an y such that $x \neq y$
- Reasoning process relies intrinsically on the finiteness of success and failure for specifications

  The system has been extended to treat some forms of (co-)inductive reasoning by using tabling and cyclic proofs

We will discuss a richer proof system that treats definitions (co-)inductively to extend the second kind of capability

We will also turn our focus to interactive theorem proving

Treating Equality as a Defined Symbol

Our interpretation of definitions is related to a "closed-world" treatment of equality

This treatment can be made explicit via introduction rules for the equality symbol

$$\frac{}{\Sigma; \Gamma \vdash \sigma \rhd (t = t)}\ eqR$$

$$\frac{\{\Sigma\theta; \Gamma\theta \vdash \Delta\theta \mid (\lambda x_1 \ldots \lambda x_n\ u)\theta =_\lambda (\lambda x_1 \ldots \lambda x_n\ v)\theta\}}{\Sigma; \Gamma, x_1, \ldots, x_n \rhd u = v \vdash \Delta}\ eqL$$

The real content is in the *eqL* rule that enforces a syntactic understanding of equality

Once again, for practicality, we may limit ourselves to a complete set of unifiers when considering cut-free proofs

Definitions and Equality

Definitions can now be reduced to at most one clause for each predicate that has the form

$$\forall \vec{x}.p\ \vec{x} \stackrel{\triangle}{=} B\ p\ \vec{x}$$

where p and $\vec{x}$ do not appear in B

Further, the rules for definitions can be simplified to just unfolding:

$$\frac{\Sigma; \Gamma, \sigma \rhd B\ p\ \vec{t} \vdash \Delta}{\Sigma; \Gamma, \sigma \rhd p\ \vec{t} \vdash \Delta}\ def\mathcal{L} \qquad \frac{\Sigma; \Gamma \vdash \sigma \rhd B\ p\ \vec{t}}{\Sigma; \Gamma \vdash \sigma \rhd p\ \vec{t}}\ def\mathcal{R}$$

In the presence of the equality rules, the earlier rules for definitions become derived ones

Definitions and Equality (Example)

For example, consider the clauses

$$nat\ 0 \stackrel{\triangle}{=} \top \qquad \forall N.nat\ (s\ N) \stackrel{\triangle}{=} nat\ N$$

These can be transformed into

$$\forall N.nat\ N \stackrel{\triangle}{=} (N = 0) \vee \exists N'\ (N = s\ N' \wedge nat\ N')$$

The following derivation then realizes the effect of the earlier *def*$\mathcal{L}$ rule:

$$\dfrac{\dfrac{\{\Sigma\theta; \Gamma\theta \vdash \Delta\theta \mid t\theta = 0\}}{\Sigma; \Gamma, \sigma \rhd t = 0 \vdash \Delta}\ eqL \quad \dfrac{\dfrac{\{(\Sigma, N')\theta; \Gamma\theta, \sigma \rhd (nat\ N')\theta \vdash \Delta\theta \mid t\theta = (s\ N')\theta\}}{\Sigma, N'; \Gamma, \sigma \rhd (t = s\ N') \wedge nat\ N' \vdash \Delta}\ eqL}{\Sigma; \Gamma, \sigma \rhd \exists N'\ (t = s\ N') \wedge nat\ N' \vdash \Delta}\ \exists L}{\Sigma; \Gamma, \sigma \rhd (t = 0) \vee \exists N'\ (t = s\ N' \wedge nat\ N') \vdash \Delta}\ \vee L$$
$$\Sigma; \Gamma, \sigma \rhd nat\ t \vdash \Delta\ def\mathcal{L}$$

A similar observation holds with respect to the earlier *def*$\mathcal{R}$ rule

Inductive and Co-Inductive Definitions

The unfolding rules amount to treating $\forall \vec{x}.p\ \vec{x} \stackrel{\triangle}{=} B\ p\ \vec{x}$ as a fixed-point of the operator B

The following rule interprets it as the least of the pre-fixed points, leading to an inductive treatment

$$\frac{\vec{x}; B\ S\ \vec{x} \vdash S\ \vec{x} \quad \Sigma; \Gamma, \sigma \rhd S\ \vec{t} \vdash \Delta}{\Sigma; \Gamma, \sigma \rhd p\ \vec{t} \vdash \Delta}\ \mu L$$

Here S is a closed term of the same type as p that serves as an *invariant*

Similarly, the following rule interprets the it as the greatest of the post-fixed points, treating it co-inductively

$$\frac{\Sigma; \Gamma \vdash \sigma \rhd S\ \vec{t} \quad \vec{x}; S\ \vec{x} \vdash B\ S\ \vec{x}}{\Sigma; \Gamma \vdash \sigma \rhd p\ \vec{t}}\ \nu R$$

Here S is a closed term of same type as p called the *co-invariant*

Inductive and Co-Inductive Definitions (Continued)

We have to be careful when mixing inductive and co-inductive definitions

For example, a proof of $; \vdash \bot$ can easily be constructed by interpreting

$$p \stackrel{\triangle}{=} p$$

both inductively and co-inductively

Cut-elimination and, consequently, consistency holds under the following additional conditions

- each clause is treated exclusively inductively, co-inductively or simply as a fixed-point (annotated $\stackrel{\mu}{=}$, $\stackrel{\nu}{=}$ or $\stackrel{\triangle}{=}$)
- the earlier stratification condition holds: $lvl(B\ p\ \vec{x}) \leq lvl(p)$
- "mutual recursion" is prohibited: $lvl(B\ (\lambda \vec{x}\ \top)\ \vec{x}) < lvl(p)$

We will assume these conditions implicitly henceforth

An Example of Inductive Reasoning

Consider the following definition

$$\forall I, J, K.plus\ I\ J\ K \stackrel{\mu}{=} (I = 0 \wedge J = K) \vee \exists M \exists N\ I = (s\ M) \wedge K = (s\ N) \wedge plus\ M\ J\ N$$

Now consider proving that *plus* is functional, i.e.

$$; \vdash \rhd \forall I \forall J \forall K\ plus\ I\ J\ K \supset \forall L\ plus\ I\ J\ L \supset K = L$$

Eliding the local context that is always empty in case, this proof reduces to one for

$$I, J, K; plus\ I\ J\ K \vdash \forall L\ plus\ I\ J\ L \supset K = L$$

Now use induction on *plus* with the invariant

$$D \equiv \lambda I \lambda J \lambda K\ \forall L\ plus\ I\ J\ L \supset K = L$$

I.e. with the righthand side over which the eigenvariables have been abstracted

An Example of Inductive Reasoning (Continued)

To look at the use of induction, rewrite the definition of *plus* as

$$\forall I, J, K.plus\ I\ J\ K \stackrel{\mu}{=} B\ plus\ I\ J\ K$$

where

$$B = \lambda P \lambda I \lambda J \lambda K\ ((I = 0 \wedge J = K) \vee \exists M \exists N\ I = (s\ M) \wedge K = (s\ N) \wedge plus\ M\ J\ N$$

Then the proof at the end has the following shape

$$\frac{L, M, N; B\ D\ L\ M\ N \vdash D\ L\ M\ N \quad \overline{I, J, K; D\ I\ J\ K \vdash D\ I\ J\ K}\ init}{I, J, K; plus\ I\ J\ K \vdash D\ I\ J\ K}\ \mu L$$

Based on the structure of B, we get for the left premise

$$\frac{J; \vdash D\ 0\ J\ J \quad L', M, N'; D\ L'\ M\ N' \vdash D\ (s\ L')\ M\ (s\ N')}{L, M, N; B\ D\ L\ M\ N \vdash D\ L\ M\ N}\ \vee L, eqL, \exists L$$

This proof can be completed now by using rules based on the structure of D and case analysis on *plus*

Another Example of Inductive Reasoning

Consider the following definitions

$$\forall N.nat\ N \stackrel{\mu}{=} (N = 0) \vee \exists N'\ N = (s\ N') \wedge nat\ N'$$

$$\forall I, J, K.plus\ I\ J\ K \stackrel{\mu}{=} (I = 0 \wedge J = K) \vee \exists M \exists N\ I = (s\ M) \wedge K = (s\ N) \wedge plus\ M\ J\ N$$

We can then show that *plus* is total on natural numbers

$$; \vdash \cdot \triangleright \forall I\ nat\ I \supset \forall J\ nat\ J \supset \exists K\ plus\ I\ J\ K$$

This can be reduced to proving

$$I; \cdot \triangleright nat\ I \vdash \cdot \triangleright \forall J\ nat\ J \supset \exists K\ plus\ I\ J\ K$$

Now use induction on *nat* with the invariant

$$\lambda I\ \forall J\ nat\ J \supset \exists K\ plus\ I\ J\ K$$

Notice again that this is the righthand side of the sequent with the eigenvariables abstracted

An Easier-to-Use Derived Rule for Induction

Suppose the objective is to prove a formula of the form

$$\forall \vec{x}\ H_1 \supset \cdots \supset (p\ \vec{t}) \supset \cdots \supset H_n \supset A$$

using induction on p which is defined as $\forall \vec{y}.p\ \vec{y} \stackrel{\mu}{=} B\ p\ \vec{y}$

Then, this formula can be changed to

$$\forall \vec{x}\ H_1 \supset \cdots \supset (p^{@}\ \vec{t}) \supset \cdots \supset H_n \supset A$$

and the formula

$$\forall \vec{x}\ H_1 \supset \cdots \supset (p^{*}\ \vec{t}) \supset \cdots \supset H_n \supset A$$

can be added to the assumptions before continuing the proof

The interpretation of $p^{@}$ and p^{*} here are the following

- p^{*} will match with p only if it has the $*$ annotation
- To unfold $p^{@}\ \vec{t}$, replace it with $B\ p^{*}\ \vec{t}$

A proof based on this schema can be transformed into one with the usual induction rule

An Example of the Use of the Derived Rule

Assume the earlier definition of *nat* and consider proving

$$\forall I \forall J\ nat\ I \supset nat\ J \supset \exists K\ plus\ I\ J\ K$$

by induction on the first occurrence of *nat*

Then we would add as a hypothesis

$$IH = \forall I \forall J\ nat^{*}\ I \supset nat\ J \supset \exists K\ plus\ I\ J\ K$$

After some introduction rules, we would be left with proving

$$I, J; IH, nat^{@}\ I, nat\ J \vdash \exists K\ plus\ I\ J\ K$$

Unfolding $(nat^{@}\ I)$ now leaves us to prove the two sequents

$$J; IH, nat^{*}\ 0, nat\ J \vdash \exists K\ plus\ 0\ J\ K \quad \text{and}$$

$$N, J; IH, nat^{*}\ N, nat\ J \vdash \exists K\ plus\ (s\ N)\ J\ K$$

These sequents can be proved easily using the induction hypothesis

Some Issues with the Nabla Quantifier

The ∇ quantifier gives us the ability to reason about binding by replacing bound variables with names

In a sense, it allows us to reasoning about "open terms"

When a term is not open, we might expect the reasoning capability to be equivalent to that on closed terms

Thus, we might desire the following formulas to be provable

$$\forall x\ (\nabla n\ p\ x) \supset p\ x$$

$$\forall x\ p\ x \supset (\nabla n\ p\ x)$$

Similarly, what seems important is which "variables" are free, not the order we talk of them

In other words, we might expect the following to hold

$$\nabla x \nabla y\ p\ x\ y \equiv \nabla y \nabla x\ p\ x\ y$$

Neither of these principles hold within the current logic

Open Terms and Induction

Induction over open terms is also not currently possible

Specifically, in the rule that treats inductive definitions

$$\frac{\vec{x}; B\ S\ \vec{x} \vdash S\ \vec{x} \quad \Sigma; \Gamma, \sigma \rhd S\ \vec{t} \vdash \Delta}{\Sigma; \Gamma, \sigma \rhd p\ \vec{t} \vdash \Delta}\ \mu L$$

the local context for establishing the invariant is empty

Thus, the invariant cannot contain any "names" in it

As a specific example, consider the following structural property

$$\nabla n \forall x\ nat\ (x\ n) \supset \exists y\ x = \lambda z\ y$$

i.e., "no name can appear in a natural number"

We could prove this if the following invariant

$$S = \lambda t\ (\forall x\ t = (x\ n) \supset \exists y\ x = \lambda z\ y)$$

where n is a name is acceptable

However, this is not acceptable in the current logic

A Richer Treatment of the Nabla Quantifier

To overcome the first set of issues we extend the logic to include the following equivalences

$$\nabla x \nabla y\ B\ x\ y \equiv \nabla y \nabla x B\ x\ y$$

$$\nabla x\ B \equiv B \quad \text{if } x \text{ does not appear in } B$$

The extension is realized in a rather simple way

- Add to the vocabulary a denumerable set of new symbols called *nominal constants*
- Drop local contexts from formulas but use a nominal constant not appearing in the formula in the rules for ∇
- Use nominal constants arbitrarily for existentials, raise over the ones in the formula when introducing eigenvariables
- When matching formulas (e.g. in the *init* or *cut* rules), we consider permutation mappings for nominal constants

The issue related to induction also is solved by this change: nominal constants can appear in the invariant

A Richer Treatment of the Nabla Quantifier

To overcome these issues we extend the logic to include the following equivalences

$$\nabla x \nabla y\ B\ x\ y \equiv \nabla y \nabla x B\ x\ y$$

$$\nabla x\ B \equiv B \quad \text{if } x \text{ does not appear in } B$$

We refer to these equivalences as the interchange and strengthening principles

The addition of these principles also allows us to simplify the proof rules by making local contexts implicit using *nominal constants*

The Logic LG^{ω}

This logic realizes the interchange and strengthening principles related to ∇ quantification

Formally, we change the existing reasoning logic as follows

- We assume a denumerable number of nominal constants at each type where ∇ quantification is permitted
  $\mathcal{C}$ stands for the set of all nominal constants, $\mathcal{K}$ for normal constants
- We drop local contexts from judgements, i.e., these become simply formulas again
- Formulas can now contain nominal constants arbitrarily, but the scope of such constants will be local
  However, definitions *must not* contain nominal constants

Equality between formulas is given by identity modulo λ-conversion and a permutation mapping on nominal constants

Written as $B' \approx B$ and read "B' is equivariant to B"

The Inference Rules for LG^{ω}

There is not much to say about the rules other than the ones involving identity and the quantifiers

For the former, we should use equivariance to match formulas

$$\frac{}{\Sigma; \Gamma, B' \vdash B}\ \mathit{init} \qquad B \approx B'$$

$$\frac{\Sigma; \Gamma_1 \vdash B \quad \Sigma; \Gamma_2, B' \vdash \Delta}{\Sigma; \Gamma_1, \Gamma_2 \vdash \Delta}\ \mathit{cut} \qquad B \approx B'$$

For the quantifier rules, an important question is what to raise over in the $\forall R$ and $\exists L$ rules

We should raise over enough nominal constants but still leave some for any ∇ quantifiers of smaller scope

It turns out to be adequate to raise only over the nominal constants appearing in the quantified formula

The Quantifier Inference Rules of LG^{ω}

Let $\mathit{supp}(B)$ denote a listing of the nominal constants appearing in B

$$\frac{\Sigma; \Gamma, B[a/x] \vdash \Delta}{\Sigma; \Gamma, \nabla x\ B \vdash \Delta}\ \nabla\mathcal{L} \qquad a \notin \mathit{supp}(B)$$

$$\frac{\Sigma; \Gamma \vdash B[a/x]}{\Sigma; \Gamma \vdash \nabla x\ B}\ \nabla\mathcal{R} \qquad a \notin \mathit{supp}(B)$$

$$\frac{\Sigma, h; \Gamma, B[(h\ \sigma)/x] \vdash \Delta}{\Sigma; \Gamma, \exists x\ B \vdash \Delta}\ \exists L \qquad \sigma = \mathit{supp}(B), h \text{ new}$$

$$\frac{\Sigma; \Gamma \vdash B[t/x]\Delta \quad t \text{ is a } (\Sigma \cup \mathcal{C})\text{-term}}{\Sigma; \Gamma \vdash \exists x\ B}\ \exists R$$

$$\frac{\Sigma; \Gamma, B[t/x] \vdash \Delta \quad t \text{ is a } (\Sigma \cup \mathcal{C})\text{-term}}{\Sigma; \Gamma, \forall x\ B \vdash \Delta}\ \forall L$$

$$\frac{\Sigma, h; \Gamma \vdash B[(h\ \sigma)/x]}{\Sigma; \Gamma \vdash \forall x\ B}\ \exists L \qquad \sigma = \mathit{supp}(B), h \text{ new}$$

Nominal Constants and Substitution

Nominal constants are accompanied by an implicit formula level binding that substitutions must be careful to respect

Sometimes, such as in the quantifier rules, we need nominal constants in the substitution terms to be interpreted locally

This is identical to the earlier notion of substitution and we will continue to use that notation ($B\theta$, $\Gamma\theta$, etc) for it

However, at other times, substitutions determined with one formula have to be applied to another

This happens, for example, with the left introduction rule for equality

In this case, the names of nominal constants must be permuted first before applying the substitution to avoid incorrect capture

This is the "logical" notion of substitution and will be written as $B[\![\theta]\!]$ ($\Gamma[\![\theta]\!]$, etc)

A Missing Feature in LG^{ω}

Nominal constants are used to represent free variables in "open" terms

Sometimes we may want to examine the structure of these terms with respect to such free variables

For example, questions such as the following arise naturally in reasoning based on syntax

- Does a given term correspond to a free variable?
- Is a given term devoid of a particular variable, i.e. is the variable fresh to the term?

The current logic does not support such analyses directly

Some of these checks can be done indirectly, e.g. a term is a nominal constant if it cannot be something else

However, this can be a cumbersome encoding that also makes reasoning difficult

A Concrete Example

Consider the following rendition of typing where tokens with capitals are implicitly universally quantified at the top-level

$$of\ \Gamma\ X\ A \stackrel{\mu}{=} member\ (X : A)\ \Gamma$$
$$of\ \Gamma\ (app\ M\ N)\ B \stackrel{\mu}{=} \exists A\ (of\ \Gamma\ M\ (arr\ A\ B) \wedge (of\ \Gamma\ N\ A))$$
$$of\ \Gamma\ (abs\ A\ M)\ (arr\ A\ B) \stackrel{\mu}{=} \nabla x\ (of\ ((x : A) :: \Gamma)\ (M\ x)\ B)$$

Then determinateness of typing can be expressed as

$$\forall \Gamma \forall M \forall A \forall B\ ctx\ \Gamma \supset (of\ \Gamma\ M\ A) \supset (of\ \Gamma\ M\ B) \supset A = B$$

if $ctx\ \Gamma$ holds iff $\Gamma = (x_1 : A_1) :: \ldots :: (x_n, A_n) :: nil$ where

- each x_i is a nominal constant and
- x_i and x_j are distinct if $i \neq j$

Note that these properties of *ctx* must be articulated for the reasoning to go through

Nominal Abstraction

This is a relation between terms that provides a means for inspecting nominal constant occurrences

This relation, written $s \trianglerighteq t$, holds between terms s and t if

- s is of type $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau$ and t is of type τ and
- s is λ-convertible to the result of abstracting over n distinct nominal constants in t

E.g, for nominal constants c_1 and c_2, the following relations hold

$$\lambda x\ x \trianglerighteq c_1 \qquad \lambda x p\ x\ c_2 \trianglerighteq p\ c_1\ c_2 \qquad \lambda x \lambda y\ p\ x\ y \trianglerighteq p\ c_1\ c_2$$

but the following do not hold

$$\lambda x\ x \trianglerighteq p\ c_1\ c_2 \qquad \lambda x\ p\ x\ c_2 \trianglerighteq p\ c_2\ c_1 \qquad \lambda x \lambda y\ p\ x\ y \trianglerighteq p\ c_1\ c_1$$

Also, a substitution θ is said to be a *solution* to the nominal abstraction $s \trianglerighteq t$ if $(s \trianglerighteq t)[\![\theta]\!]$ holds

Nominal Abstraction and Term Structure

Existence of solutions allow us to characterize terms based on nominal constant occurrences

For example, consider the following

$$\lambda x\ fresh\ x\ T \trianglerighteq S$$

This is satisfied only by (ground) terms of the form $fresh\ n\ S'$ where n is a nominal constant that does not appear in S'

Similarly, consider the following

$$\lambda x\ name\ x \trianglerighteq name\ S$$

To satisfy this, S must be a nominal constant

Note that $\trianglerighteq$ extends the equality relation and its "structure discerning" capabilities to terms with nominal constants

Proof Rules for Nominal Abstraction

We will use ambiguously the notation $s \triangleright t$ to denote in the logic the semantic notion just described

The rules for introducing this relation are the following

$$\frac{}{\Sigma; \Gamma \vdash s \trianglerighteq t}\ \trianglerighteq\mathcal{R} \qquad s \trianglerighteq t \text{ holds}$$

$$\frac{\{\Sigma\theta; \Gamma[\![\theta]\!] \vdash C[\![\theta]\!] \mid \theta \text{ is a solution to } (s \trianglerighteq t)\}}{\Sigma; \Gamma, s \trianglerighteq t \vdash \Delta}\ \trianglerighteq\mathcal{L}$$

Substitutions in the $\trianglerighteq\mathcal{L}$ rule can be restricted in a cut-free setting to a *complete set of nominal abstraction solutions*

This is a set S such that

- each $\theta \in S$ is a solution to $s \trianglerighteq t$, and
- every solution to $s \trianglerighteq t$ is "covered" over Σ by one in S

Moreover, such sets can be computed using higher-order (pattern) unification

Defining Typing Contexts Using Nominal Abstraction

The property we wanted of typing contexts can now be articulated via the following definition

$$ctx\ K \triangleq (K = nil) \vee \exists T \exists L\ ((\lambda x\ (x : T) :: L) \trianglerighteq K \wedge ctx\ L)$$

Given this definition, *ctx L* is provable exactly when *L* is a type assignment to distinct nominal constants

Conversely, if *ctx L* appears as a premise, the definition allows it to be concluded that *L* satisfies the desired restrictions

Exercise:

Using this definition of *ctx* try to prove determinateness of typing

$$\forall \Gamma \forall M \forall A \forall B\ ctx\ \Gamma \supset (of\ \Gamma\ M\ A) \supset (of\ \Gamma\ M\ B) \supset A = B$$

A Pattern-Based Form for Definitions

An alternative to allowing nominal abstractions is to let a predicate be defined via multiple clauses of the form

$$\forall \vec{x}.(\nabla \vec{z}.(p\ \vec{t}) \triangleq B\ p\ \vec{x})$$

i.e., clauses that use patterns and nabla quantifiers in the head

The Meaning of Such a Clause

An instance of $p\ \vec{t}$ is true if the corresponding instance of B is true, provided

- the variables in $\vec{z}$ are instantiated with unique nominal constants $\vec{a}$
- the variables in $\vec{x}$ are instantiated with terms not containing constants in $\vec{a}$

Encoding properties of typing contexts with such definitions:

$$ctx\ nil \triangleq \top$$

$$\forall L, A.\ (\nabla x.ctx\ ((x : A) :: L)) \triangleq ctx\ L$$

Clauses with Patterns versus Nominal Abstractions

The effect of a predicate definition based on *n* "patterned clauses" of the form

$$\forall \vec{x}_i.(\nabla \vec{z}_i.(p\ \vec{t}_i) \triangleq B_i\ p\ \vec{x}_i)$$

can be realized by the following definition in the "one clause" format

$$\forall \vec{y}.(p\ \vec{y}) \triangleq \bigvee_{i \in 1,\ldots,n} \exists \vec{x}_i\ (\lambda \vec{z}_i\ (p'\ \vec{t}_i) \trianglerighteq p'\ \vec{y}) \wedge B_i\ p\ \vec{x}_i$$

where p' is a new predicate name with type identical to that of p and the $\vec{y}_i$ are variables chosen to not appear in $\vec{t}_i$

Left and right introduction rules capturing the intended meaning of patterned clauses can be provided based in this translation

Inductive definitions can be provided a similar treatment through patterned clauses with nabla in the head

Introduction Rules for Pattern Based Definitions

Let $\mathcal{D}$ represent the collection of clauses for p in pattern form

- Right introduction rules for such a definition

$$\frac{\Sigma; \Gamma \vdash (B\ p\ \vec{x})[\theta]}{\Sigma; \Gamma \vdash p\ \vec{s}}\ def\mathcal{R}^p$$

  where $\forall \vec{x}.(\nabla \vec{z}.p\ \vec{t}) \triangleq B\ p\ \vec{x} \in \mathcal{D}$ and θ is such that $range(\theta) \cap \Sigma = \emptyset$ and $(\lambda \vec{z}.p\ \vec{t})[\theta] \trianglerighteq p\ \vec{s}$ holds

- Left introduction rule for such a definition

$$\frac{\left\{ \Sigma\theta; \Gamma[\![\theta]\!], (B\ p\ \vec{x})[\![\theta]\!] \vdash C[\![\theta]\!] \;\middle|\; \begin{array}{l} \forall \vec{x}.(\nabla \vec{z}.p\ \vec{t}) \triangleq B\ p\ \vec{x} \in \mathcal{D} \\ \text{and } \theta \text{ is a solution} \\ \text{to } ((\lambda \vec{z}.p\ \vec{t}) \trianglerighteq p\ \vec{s}) \end{array} \right\}}{\Sigma; \Gamma, p\ \vec{s} \vdash C}\ def\mathcal{L}^p$$

These rules can also be used for inductive and co-inductive definitions, i.e. if $\stackrel{\mu}{=}$ or $\stackrel{\nu}{=}$ is used in the definition instead of $\triangleq$

Left Rule for Inductive Definitions

Suppost that p is defined by n clauses of the form

$$\forall \vec{x}_i.(\nabla \vec{z}_i.(p\ \vec{t}_i) \stackrel{\mu}{=} B_i\ p\ (\vec{x}_i)$$

Then the following rule can be used to introduce p on the left of a sequent

$$\frac{\left\{\vec{x}_i; B_i\ S\ \vec{x}_i \vdash \nabla \vec{z}_i.S\ \vec{t}_i\right\}_{i \in 1..n} \quad \Sigma; \Gamma, S\ \vec{s} \vdash C}{\Sigma; \Gamma, p\ \vec{s} \vdash C}\ \mu L^p$$

The Two-Level Logic Approach to Reasoning

Specifications can be encoded and reasoned about directly in $\mathcal{G}$, the fully featured reasoning logic

However, we propose instead to do the following

- Write specifications in a restriction of *hohh* called hH^2
- Embed the specification logic via a definition based on its operational semantics into the reasoning logic
- Reason about the object languages/systems via the encoding of the specification logic

We refer to this as the two level logic approach to reasoning about specifications

Tradeoffs Involved in the Two-Level Logic Approach

There are some drawbacks related to the use of this approach

- The full flexibility of $\mathcal{G}$ is not available for writing specifications
- There is an indirection involved in reasoning: we have to do this through the encoding of the specification logic

However, there are some advantages too

- Specifications can be used both for animation and for reasoning
- Meta-theoretic properties of the specification logic can be exploited in the reasoning process

We will try to see how the reasoning overhead can be reduced sufficiently to make this approach viable

The Abella system and experiments with it provide evidence for the richness of the framework in practice

The hH^2 Logic

The main restrictions to *hohh* that results in hH^2 are

- Predicate quantification is disallowed
- Implication goals can have only atoms on left

Thus, goals and program clauses have the following structure

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

The sequents that arise from applying the *hohh* operational semantics to this language have the form

$$\Sigma; \Delta, \mathcal{L} \vdash G$$

with the following connotations

- Δ consists of the program clauses from the original specification
- $\mathcal{L}$ is a set of atoms introduced via implication goals

We will use this property in the embedding

The Encoding of Formulas and Terms

Both $\mathcal{G}$ and hH^2 are based on the simply typed lambda calculus so the encoding of the syntax can be shallow

- Sorts in hH^2 give rise to corresponding ones in $\mathcal{G}$ with one special case

  The type o in hH^2 is represented by a type *fmla* in $\mathcal{G}$
- Constants in hH^2 carve out ones in $\mathcal{G}$ with one proviso

  Logical constants become nonlogical ones yielding terms of type *fmla*
- The sort *atm* is used for atomic formula from hH^2

  $\langle A \rangle$ makes a *fmla* from an *atm* A and $\supset$ is encoded via a constant of type $atm \to fmla \to fmla$
- Eigenvariables in hH^2 are encoded by nominal constants in $\mathcal{G}$

Adequacy of the encoding, denoted $\{\!\{\cdot\}\!\}$, follows from showing a suitable bijection and compositionality of substitution

Encoding hH^2 Programs

For each clause $\forall \vec{x}\ G_1 \supset \cdots \supset G_n \supset A$ in the fixed hH^2 specification Δ, we will use a clause of the form

$$prog\ A\ (G_1 \wedge \ldots \wedge G_n) \stackrel{\triangle}{=} \top$$

Here, *prog* has type $atm \to fmla \to o$

For example, corresponding to the formalization of type evaluation, we would have the clauses

$$prog\ (of\ (abs\ A\ M)\ (arr\ A\ B))$$
$$(\forall x\ (of\ x\ A) \supset \langle of\ (R\ x)\ B \rangle) \stackrel{\triangle}{=} \top$$
$$prog\ (of\ (app\ M\ N))$$
$$(\langle of\ M\ (arr\ A\ B) \rangle \wedge \langle of\ N\ A \rangle) \stackrel{\triangle}{=} \top$$

Notice that these clauses can be generated automatically from a specification as indeed is done by Abella

Encoding the Operational Semantics

Three special definitions are used in the encoding of hH^2 derivations

- For an special sort *nt* and $z : nt$, $s : nt \to nt$ and $nat : nt \to o$

  $$nat\ z \stackrel{\mu}{=} \top \qquad nat\ (s\ N) \stackrel{\mu}{=} nat\ N$$

  For encoding induction on the lengths of hH^2 derivations
- For *lt* of type $nt \to nt \to o$

  $$lt\ z\ (s\ N) \stackrel{\mu}{=} \top \qquad lt\ (s\ M)\ (s\ N) \stackrel{\mu}{=} lt\ M\ N$$

  Helps in realizing strong induction over hH^2 derivations
- For sort *atmlist*, $nil : atmlist$, $::$ an infix operator of type $atm \to atmlist \to atmlist$ and $member : atm \to atmlist \to o$

  $$member\ A\ (A :: L) \stackrel{\mu}{=} \top \qquad member\ A\ (B :: L) \stackrel{\mu}{=} member\ A\ L$$

  Used for encoding "lookup" in a context created in the course of a derivation

Encoding the Operational Semantics (Continued)

Let $seq : nt \to atmlist \to fmla \to o$ be defined by the clauses

$$seq\ (s\ N)\ L\ \top \stackrel{\mu}{=} \top$$
$$seq\ (s\ N)\ L\ (G_1 \vee G_2) \stackrel{\mu}{=} seq\ N\ L\ G_1$$
$$seq\ (s\ N)\ L\ (G_1 \vee G_2) \stackrel{\mu}{=} seq\ N\ L\ G_2$$
$$seq\ (s\ N)\ L\ (G_1 \wedge G_2) \stackrel{\mu}{=} seq\ N\ L\ G_1 \wedge seq\ N\ L\ G_2$$
$$seq\ (s\ N)\ L\ (A \supset G) \stackrel{\mu}{=} seq\ N\ (A :: L)\ G$$
$$seq\ (s\ N)\ L\ (\forall x\ B\ x) \stackrel{\mu}{=} \nabla x\ seq\ N\ L\ (B\ x)$$
$$seq\ (s\ N)\ L\ \langle A \rangle \stackrel{\mu}{=} member\ A\ L$$
$$seq\ (s\ N)\ L\ \langle A \rangle \stackrel{\mu}{=} \exists b\ prog\ A\ b \wedge seq\ N\ L\ b$$

Then hH^2 derivability of $\Sigma; \Delta, \mathcal{L} \vdash G$ amounts (for suitable *prog* clauses) to the $\mathcal{G}$-provability of $\exists n\ ((nat\ n) \wedge (seq\ n\ \{\!\{\mathcal{L}\}\!\}\ \{\!\{G\}\!\}))$

We will abbreviate the latter by $\mathcal{L} \Vdash G$, assuming a fixed Δ

Proving Properties of Specifications

Properties of specifications can be expressed in $\mathcal{G}$ modulo the encoding of derivability in hH^2

For example, given *prog* clauses encoding evaluation and typing, subject reduction can be expressed via the statement

$$\forall e \forall v \forall ty\ (\Vdash \langle eval\ e\ v \rangle) \supset (\Vdash \langle of\ e\ ty \rangle) \supset (\Vdash \langle eval\ v\ ty \rangle)$$

Proofs of such properties will typically involve induction on the lengths of hH^2 derivations

For example, the subject reduction formula is equivalent to

$$\forall e \forall v \forall ty \forall n\ (nat\ n \wedge seq\ n\ nil\ \langle eval\ e\ v \rangle \supset (\Vdash \langle of\ e\ ty \rangle) \supset (\Vdash \langle eval\ v\ ty \rangle))$$

If we try to prove this by induction on *nat n*, this will amount to induction on the length of the hH^2 derivation of *eval e v*

Using Strong Induction on Derivation Lengths

To prove the subject reduction formula, we will actually need to unfold the *seq* definition twice

- we first look up the clause for $(eval\ (app\ e_1\ e_2)\ v)$
- we then use the definition of *seq* for a conjunction

To deal with this, we prove instead the formula

$$\forall e \forall v \forall ty \forall n \forall n\ (nat\ n \wedge (lt\ m\ n) \wedge seq\ m\ nil\ \langle eval\ e\ v \rangle \supset (\Vdash \langle of\ e\ ty \rangle) \supset (\Vdash \langle eval\ v\ ty \rangle))$$

Induction on $(nat\ n)$ and the resulting case analysis lead to assuming the formula for any *m* for which $(lt\ m\ n)$ holds

To conclude the proof, we only need to observe that the original formula must hold if this generalization does

This is actually a general schema for justifying strong induction in this setting

Leveraging the Specification Logic Meta-Theory

Meta-theoretic properties of the specification logic such as the following can be stated and proved via its encoding in $\mathcal{G}$

Monotonicity

$\forall \Gamma \forall \Delta \forall G\ (\forall X\ member\ X\ \Gamma \supset member\ X\ \Delta) \supset \Gamma \Vdash G \supset \Delta \Vdash G$

Cut Admissibility

$\forall \Gamma \forall A \forall G\ (A :: \Gamma) \Vdash G \supset \Gamma \Vdash A \supset \Gamma \Vdash G$

Instantiation

$\forall \Gamma \forall G\ (\nabla x\ (\Gamma\ x) \Vdash (G\ x)) \supset \forall t\ (\Gamma\ t) \Vdash (G\ t)$

If the framework allows for lemma use, then these results can simplify proofs

For example, cut admissibility and instantiation immediately yield a substitutivity principle for typing judgements

$(\nabla x\ (of\ x\ a) :: L \Vdash \langle of\ (T\ x)\ B \rangle) \supset (L \Vdash \langle of\ u\ a \rangle) \supset L \Vdash \langle of\ (T\ u)\ B \rangle$

The Abella System

This system realizes the two-level logic approach using hH^2 and $\mathcal{G}$

- It automatically creates *prog* clauses from hH^2 specs
- It treats judgements of the form $L \Vdash G$ in a special way, building in the processing of *seq* and *prog* clauses
- It uses the annotation based treatment of induction on $L \Vdash G$ judgements, thereby building in strong induction
- It embeds meta-theoretic properties of the specification logic into tactics by exploiting a lemma facility

These features considerably reduce the syntactic overhead of the two-level logic approach

The Abella website documents many successful applications

Exercise Suggestion: Try using the system to formalize proofs of meta-theorems of intuitionistic logic from the first problem set

Adequacy of the Two-Level Logic Approach

To allow results to be extracted about the object systems, we have to show the following

- The encoding in hH^2 of the object system is adequate

  Has to be done for each encoding, but there is evidence that λ-tree syntax simplifies this task
- The encoding of hH^2 into $\mathcal{G}$ must be adequate

  In particular, theorems proved in $\mathcal{G}$ via the encoding should be transferrable to hH^2
- The results in the previous two items should "compose" to yield object language theorems from one proved in $\mathcal{G}$

  This again is dependent on particular object but seems to be easily achievable

The second step is complicated by habitation questions

One result: This adequacy holds when eigenvariables and nominal constants are permitted only at already inhabited types