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

Logic Programming and Rule-Based Specifications

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

- First-order terms generalize traditional abstract syntax:
  \[ \alpha_1 \rightarrow \alpha_2 \implies arr([\alpha_1],[\alpha_2]) \]
  \[ ([e_1,e_2]) \implies app([e_1],[e_2]) \]

- The logical structure of Horn clauses mirrors closely the structure of (syntax-directed) rules:
  \[ \Gamma \vdash m : a \rightarrow b \quad \Gamma \vdash n : a \]
  \[ \Gamma \vdash (m n) : b \]
  \[ \Downarrow \]
  \[ \text{of}(\Gamma,E1,E2,Ty2) \] :-
  \[ \text{of}(\Gamma,E1,\text{arr}(Ty1,Ty2)), \text{of}(\Gamma,E2,Ty1). \]
An Inadequacy of Traditional Abstract Syntax

No support is provided for binding notions

E.g., consider the mini-ML abstraction \((\text{fn } x : ty \Rightarrow e)\)

The best possible representation in the first-order setting:

\[\text{abst}(x,[ty],[e])\]

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

\[
\begin{align*}
of(Gamma,\text{var}(X),Ty) & :- \text{member}((X:Ty),Gamma). \\
of(Gamma,\text{app}(E1,E2),Ty2) & :- \\
& \quad of(Gamma,E1,\text{arr}(Ty1,Ty2)),of(Gamma,E2,Ty1). \\
of(Gamma,\text{abst}(X,Ty1,E),\text{arr}(Ty1,Ty2)) & :- \\
& \quad of([[(X:Ty1)|Gamma]],E,Ty2).
\end{align*}
\]

This program seems to follow naturally from the typing rules

Unfortunately, it is incorrect

For example, the following query will succeed

\[
? \dashv of([],\text{abst}(x,int,\text{abst}(x,\text{arr}(int,int)),
\quad \text{app}(\text{var}(x),\text{var}(x))),
\quad Ty).
\]

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 $\Gamma \vdash B$
    - $\Gamma$ is called the left-hand side or antecedent
    - $\Delta$ is called the right-hand side or succedent
    - The formulas in $\Gamma$ should be thought of as assumptions and those in $\Delta$ 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: $\bot$, $\top$, $\land$, $\lor$, $\forall$ and $\exists$

The Identity Rules

There are two rules of this kind

$B \vdash B$ \quad \text{init}

$\Gamma_1, B \vdash \Delta_1, B \quad \Gamma_2, B \vdash \Delta_2 \quad \text{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 Structural Rules

There are two kinds of such rules: contraction and weakening

$\Gamma, B, B \vdash \Delta \quad \text{cL}$
$\Gamma, B \vdash \Delta \quad \text{cR}$

$\Gamma \vdash \Delta, B \quad \text{wL}$
$\Gamma \vdash \Delta, B \quad \text{wR}$

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 Operational Rules

These rules introduce logical symbols into formulas

$\Gamma \vdash \Delta, \Gamma \quad \text{R}$
$\bot \vdash \Gamma \quad \text{L}$

$\Gamma, B_1 \vdash \Delta \quad \Gamma, B_2 \vdash \Delta \quad \Gamma \vdash \Delta, B_1 \lor B_2 \quad \text{R}_i$
$\Gamma, B_1 \vdash \Delta \quad \Gamma, B_1 \lor B_2 \vdash \Delta \quad \text{L}_i$
$\Gamma, B_1 \lor B_2 \vdash \Delta \quad \text{L}$

$\Gamma, B_1 \vdash \Delta, B_2 \quad \Gamma, B_2 \vdash \Delta, B_1 \quad \Gamma \vdash \Delta, B_1 \lor B_2 \quad \text{R}$
$\Gamma, B_1 \lor B_2 \vdash \Delta, B_1 \lor B_2 \quad \text{L}$

$\Gamma, B[t/x] \vdash \Delta \quad \Gamma, B[c/x] \vdash \Delta \quad \Gamma \vdash \Delta, B[t/x] \quad \text{R}$
$\Gamma, \forall x B \vdash \Delta \quad \Gamma, \exists x B \vdash \Delta \quad \text{L}

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
  \[
  \begin{align*}
  \Gamma, B &\vdash \Delta \quad \text{cut} \\
  \frac{\Gamma_1 \vdash B, \Gamma_2 \vdash \Delta}{\Gamma_1, \Gamma_2 \vdash \Delta} \\
  \Gamma, B_1 &\vdash \Gamma, B_2 \vdash \Delta \quad \supset L \\
  \frac{\Gamma_1 \vdash B_1, \Gamma_2 \vdash B_2 \vdash \Delta}{\Gamma_1, \Gamma_2 \vdash \Delta} \\
  \end{align*}
  \]
  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 $\bot L\bot$.

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:

\[
\begin{align*}
\frac{\Gamma \vdash \Delta, B_1, B_2}{\Gamma \vdash \Delta, B_1 \lor B_2} &\quad \lor^* \\
\frac{\Gamma, B_1, B_2 \vdash \Delta}{\Gamma, B_1 \land B_2 \vdash \Delta} &\quad \land^* \\
\frac{\Gamma \vdash \Delta, B_1}{\Gamma, B_1 \supset B_2 \vdash \Delta} &\quad \supset^* \\
\frac{\Gamma, \forall x \ B, B[t/x] \vdash \Delta}{\Gamma, \forall x \ B \vdash \Delta} &\quad \forall^* \\
\frac{\Gamma \vdash \Delta, \exists x \ B, B[t/x]}{\Gamma \vdash \Delta, \exists x \ B} &\quad \exists^* \\
\end{align*}
\]

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

\[
\begin{align*}
\frac{\Gamma, B_1 \supset B_2 \vdash \Delta, B_1}{\Gamma, B_1 \supset B_2 \vdash \Delta} \\
\frac{\Gamma, B_2 \vdash \Delta}{\Gamma, B_1 \supset B_2 \vdash \Delta} &\quad \supset L
\end{align*}
\]

Other variants, yielding decidability in the propositional case,

Using the Sequent Calculus

- Using the calculi we can show the derivability of formulas in different systems.
- For example, consider
  \[
  (X \lor (Y \land Z)) \supset (X \lor Y) \land (X \lor Z) \\
  (P \supset Q) \supset ((\neg P) \lor Q) \\
  (\exists x \ P x) \supset (\forall (P a) \lor (P b)) \supset Q) \\
  \exists x \forall y ((P x) \lor (P y))
  \]
  Here $\neg P$ stands for $P \lor \bot$.
- 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
  \[
  \begin{align*}
  \frac{\Gamma, B \vdash B}{\Gamma, B \vdash B} &\quad \text{init}
  \end{align*}
  \]
  Also, we can understand now how to limit $B$ to be atomic.

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 $P \vdash G$.
  - Where $P$ 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

Specifying and Reasoning

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!

Giving the Search Viewpoint Substance

Let $\mathcal{P} \models G$ 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 $\quad \mathcal{P} \models \top$

OR $\quad \mathcal{P} \models (G_1 \lor G_2)$ iff $\mathcal{P} \models G_1$ or $\mathcal{P} \models G_2$

AND $\quad \mathcal{P} \models (G_1 \land G_2)$ iff $\mathcal{P} \models G_1$ and $\mathcal{P} \models G_2$

AUGMENT $\quad \mathcal{P} \models F \supset G$ iff $\mathcal{P} \cup \{F\} \models G$

INSTANCE $\quad \mathcal{P} \models \exists x \ G$ iff $\mathcal{P} \models G[t/x]$ for some $t$

GENERIC $\quad \mathcal{P} \models \forall x \ G$ iff $\mathcal{P} \models G[c/x]$ for a new constant $c$

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 $\mathcal{P} \models G$ 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 \( \langle D, G, \vdash \rangle \) of two sets of formulas and a proof relation is an abstract logic programming language (ALPL) if and only if for any finite multiset \( P \) of elements from \( D \) and any \( G \in G \),
\[
P \vdash G \quad \text{iff} \quad P \vdash G \text{ has a uniform proof}
\]

Terminology
Given an ALPL \( \langle D, G, \vdash \rangle \)

- Members of \( D \) will be called program clauses
- Members of \( G \) will be called goals or goal formulas
- Finite multisets of program clauses will be called programs

Non-Examples of ALPLs

- If \( D \) contains disjunctions and \( G \) contains existentially quantified formulas; \( \vdash \) can be \( \vdash C \), \( \vdash I \) or \( \vdash M \)

Counterexample: Consider \((p a) \lor (p b) \vdash \exists x (p x)\)

- Let \( G \) contain implications and disjunctions and let \( \vdash \) be \( \vdash C \)

Counterexample: Consider \(- (A \supset B) \lor A \) for \( A, B \) atomic

- Let \( D \) contain formulas of the form \((B_1 \land ... \land B_n) \supset A\) where \( B_i \) is either \( A_i \) or \( A_i \supset \bot \) for \( A, A_i \) atomic

- A \( \supset \bot \) is written as \( \neg A \), \( B_i \) is called a literal

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
\[
G ::= T \mid A \mid G \lor G \mid G \land G \mid \exists x G
D ::= A \mid (G \supset A) \mid (D \land D) \mid \forall x D
\]
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 \( P \) is a finite subset of \( \mathcal{D}_1 \) and \( G \in \mathcal{G}_1 \) then \( P \vdash_C G \) iff \( P \vdash_I G \)

Proof Sketch

Generalize the requirement to the following
\[
P \vdash G_1, ..., G_n \text{ is classically provable}
\]

for some \( i, 1 \leq i \leq n, P \vdash G_i \) has an intuitionistic proof

Shown by induction, considering by cases the last rule in the proof

Details are left as an exercise

Gopalan Nadathur Specification and Reasoning
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

Some Observations About fohh
fohh provides means for scoping over names and code
- the goal (∀x G) gives us a new name to use when trying to solve G
- the goal D ⊃ 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
\[ ⊢ p ∨ (p ⊃ q), \text{ and} \]
\[ ((p a) ∧ (p b)) ⊃ q ⊢ ∃x ((p x) ⊃ q) \]
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

First-Order Hereditary Harrop Formulas
Let \( \mathcal{G}_2 \) and \( \mathcal{D}_2 \) be the G and D formulas given by
\[
G ::= \top | A | G ∨ G | G ∧ G | ∃x G | ∀x G | D ⊃ G
\]
\[
D ::= A | G ⊃ D | D ∧ D | ∀x D
\]
where A is a (first-order) atomic formula

The language of first-order hereditary Harrop formulas or fohh is the triple \( \langle \mathcal{D}_2, \mathcal{G}_2, ⊢ \rangle \)
The D-formulas are also called (first-order) hereditary Harrop formulas
These formulas are related to ones attributed to Harrop:
\[
H ::= \top | A | B ⊃ H | H ∧ H | ∀x H
\]
where B is an arbitrary formula and A is atomic
Specifically, the restriction on ∨ and ∃ is applied hereditarily to all positive contexts in program clauses

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 \( D ∈ \mathcal{D}_2 \), its set of instances is denoted by \( [D] \) and is defined as follows
- If \( D = ∀x D' \), then \( [D] = \bigcup \{ [D'[t/x]] \mid t \text{ is a closed term} \} \)
- If \( D = (D_1 ∧ D_2) \), then \( [D] = [D_1] ∪ [D_2] \)
- If \( D = A \), then \( [D] = \{ A \} \)
- If \( D = G ⊃ D' \), then \( [D] = \{ G • C \mid C ∈ [D'] \} \) where
  - \( G • A = G ⊃ A \) if A is atomic
  - \( G • (G' ⊃ A) = (G ∧ G') ⊃ A \)

If \( P \) is fohh program, then \( [P] = \bigcup \{ [D] \mid D ∈ P \} \)
A Simplified Proof System for fohh

Sequents now have the form \( \Sigma ; P \vdash G \) where \( \Sigma \) is signature

\[
\begin{align*}
\Sigma ; P \vdash \top & \quad \text{T Rule} \\
\Sigma ; P \vdash B_1 & \quad \text{R Rule} \\
\Sigma ; P \vdash B_1 \lor B_2 & \quad \text{V Rule} \\
\Sigma ; P \vdash \forall x B & \quad \text{∀ Rule}
\end{align*}
\]

- \( \Sigma ; P \vdash \top \) if \( \Sigma \) is signature
- \( \Sigma ; P \vdash B_1 \) if \( \Sigma ; P \vdash B_2 \)
- \( \Sigma ; P \vdash B_1 \lor B_2 \) if \( \Sigma ; P \vdash B_1 \land B_2 \)
- \( \Sigma ; P \vdash \forall x B \) if \( \Sigma ; P \vdash \exists x B \)

\( t \) is a \( \Sigma \)-term decision, \( D \in P \)

- \( \Sigma ; P \vdash A \) if \( \Sigma ; P \vdash D \text{ and } A \)
- \( \Sigma ; P \vdash G \) if \( \Sigma ; P \vdash D \text{ and } G \)

Comments on the Simplified Sequent Calculus

- This calculus is intended for constructing proofs of sequents of the form \( \Sigma ; P \vdash G \) where \( \Sigma \) is the ambient signature
- By “\( \Sigma \)-term” we mean a (closed) term all of whose constants and function symbols are drawn from \( \Sigma \)
- 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
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

\[
G ::= \top | A | G \lor G | G \land G | \forall x\ G | \exists x\ G | D \supset G
\]

\[
D ::= A | G \supset A | \forall x\ D
\]

Here the correspondence to Prolog-like clauses becomes clear.

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

**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
    \[
    \forall x\ \forall y\ ((heated(y) \land germ(x) \land in(x, y)) \supset dead(x))
    \]
  - A (particular) jar is heated
    \[
    heated(j)
    \]

- The goal: “Is there a sterile jar?”
  \[
  \exists x\ sterile(x)
  \]

**Substitution and Quantification in `fohh`**

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

This means we have to be careful about quantifier scopes.

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

\[
p(X) :- \forall y\ q(X, y)
\]

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

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

Rather, it should yield

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

Similarly, the goal

\[
\exists x\ \forall y\ x = y
\]

should not be solvable.

**Examples**

AUGMENT currently provides a means for nested definitions like let

\[
\begin{align*}
\text{reverse } L1& \text{ L2 :- } \\
& ((\pi l\ \text{rev_aux } nil \ L \ L), \\
& (\pi X\ \pi L1\ \pi L2\ \pi L3\ \\
& \text{rev_aux } (X::L1) \ L2 \ L3 :- \\
& \text{rev_aux } L1 \ (X::L2) \ L3)) \\
& \Rightarrow \text{rev_aux } L1 \ [] \ L2.
\end{align*}
\]

Here `(\pi x\ \ (P x))` is concrete syntax for `\forall x\ P(x)` and `\Rightarrow` 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.
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

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 $C$ of constants and a set $V$ of variables:
- any element of $C$ or $V$ is a term
- the abstraction of $x \in V$ 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 \( y \notin FV(t) \)
- **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 \( x \notin FV(t) \)
  - **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 \( C \) 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

\[
\frac{X \quad X = \lambda \beta \eta Y}{EQ} \\
\frac{[X]}{I} \\
\frac{Y \quad X \supset Y}{E}
\]

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 \( X \equiv \lambda \beta \eta X \supset M \).

We then construct the following proof \( \pi \) of \( X \supset M \):

\[
\begin{align*}
X & \quad X \equiv \lambda \beta \eta X \supset M \quad \text{EQ} \\
X & \quad X \supset M \quad \text{EQ} \\
& \quad M \quad \text{E}
\end{align*}
\]

We now complete the proof of \( M \) as follows:

\[
\begin{align*}
\pi \quad X \supset M = \lambda \beta \eta X \quad \text{EQ} \\
\pi \quad X \quad \text{EQ} \\
\pi \quad M \quad \text{E}
\end{align*}
\]

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 \textit{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 \textit{formulas} from which we form sequents.

We also distinguish specific constants over the type \( o \) used to represent connectives and quantifiers.

\[
\begin{align*}
\lor, \land, \supset & \quad \text{of type} \quad o \rightarrow o \rightarrow o \\
\top, \bot & \quad \text{of type} \quad o \\
\Pi_{\alpha}, \Sigma_{\alpha} & \quad \text{of type} \quad (\alpha \rightarrow o) \rightarrow o
\end{align*}
\]

Here \( \Pi_{\alpha} \) and \( \Sigma_{\alpha} \) represent a \textit{family} of constants, one for each type \( \alpha \).

These constants are called \textit{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, \( \Pi \) and \( \Sigma \) represent the \textit{universal set} and the \textit{existential set} recognizers.
Realizing Substitution via $\beta$-Conversion

Advantage of such a representation: Substitution is completely and correctly realized through application and $\beta$-conversion.

For example, we have the following correspondence:

\[
\forall x \ B(x) \quad \text{represented by} \quad \Pi P \quad (P = \lambda x[B(x)])
\]

\[
\text{instantiated by } t \quad \text{applied to } [t]
\]

\[
B(t) \quad \text{represented by} \quad (P[t])
\]

Scope respecting substitution is implicit in the $\lambda$-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 $\lambda$-conversion.

Can be done in one of two ways:

- Assume that matching of formulas to schemas incorporates $\lambda$-conversion.
- Add one more rule of the form:

\[
\Gamma' \vdash \Delta' \quad \lambda
\]

where $\Gamma$ and $\Gamma'$ and $\Delta$ and $\Delta'$ differ only by $\lambda$-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 $\lambda z((q z) \land \forall x ((z x) \supset (q x)))$ yields

\[
((q A) \land \forall x ((A x) \supset (q x))) \supset (c A)
\]

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 fooh 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
  \[ \forall p \, (q \supset p) \land q \]
  From this, any formula can be derived.

We shall therefore require clause heads to be rigid atoms.

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 \( G_3 \) and \( D_3 \) be the \( G \) and \( D \) formulas given by
\[
G \ ::= \top | A | G \lor G | G \land G | \exists x \, G | \forall x \, G | D \supset G \\
D \ ::= A_r | G \supset D | D \land D | \forall x \, D
\]
where \( A \) stands for a positive atom and \( A_r \) for a rigid atomic formula.

The language of higher-order hereditary Harrop formulas or \textit{fooh} is the triple \( \langle D_3, G_3, \Gamma \rangle \).

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”
\[
\exists Q \forall p \, \forall q \, ((r (p \supset q)) \supset (r (Q \, p \, q))) \land (Q (t \lor s) (s \lor t))
\]
This formula is provable but its proof requires solving the goal
\[(t \lor s) \supset (s \lor t)\]
for which no uniform proof exists.

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

Showing \textit{fooh} is an ALPL

Let a positive term be any \( \lambda \)-term in which the symbols \( \supset \) and \( \bot \) do not appear.

Key Lemma

Let \( P \) be an \textit{fooh} program and let \( G \) be an \textit{fooh} goal such that \( P \vdash G \) has an intuitionistic proof.

Then \( P \vdash G \) must have such a proof in which in any occurrence of the rules
\[
\frac{\Gamma, \forall x \, B, B[t/x] \vdash \Delta}{\Gamma \vdash B[t/x]} \quad \forall L \\
\frac{\Gamma \vdash B[t/x] \quad \exists R}{\Gamma, \exists x \, B \vdash \Delta}
\]
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 $\mathsf{hohh}$

We can describe a reduced proof system for $\mathsf{hohh}$ that for the most part preserves the nature of computation for $\mathsf{fohh}$:

- 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 $\lambda$-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.

Overview of the Language $\mathsf{\lambda Prolog}$

The $\mathsf{\lambda Prolog}$ language is a realization of $\mathsf{hohh}$ 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 $\mathsf{\lambda}$-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 $\mathsf{int}$, $\mathsf{real}$, $\mathsf{string}$ and $\mathsf{o}$ are built-in as also the type constructor $\mathsf{list}$.

The programmer can extend this collection via declarations of the form

$$\text{kind tycon type} \rightarrow \ldots \rightarrow \text{type}.$$  

The number of occurrences of $\text{type}$ determines the arity.

For example, the declarations

$$\text{kind i type.}$$
$$\text{kind pairty type} \rightarrow \text{type} \rightarrow \text{type}.$$  

make it possible to write the type expression

$$\mathsf{i} \rightarrow (\mathsf{list \ int}) \rightarrow (\mathsf{pairty \ i \ (\mathsf{list \ int})}).$$
The Terms Language

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

Types of variables are generally inferred but can also be specified at binding site: e.g. \(((x:\text{int}) \ \lambda F)\)

The types of constants are identified via declarations such as

\[
\text{type} - \quad \text{int} \to \text{int} \to \text{int}.
\]
\[
\text{type} :: \quad (\text{A} \to (\text{list A}) \to (\text{list A})).
\]

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

\[
\text{infixl} - 255.
\]
\[
\text{infixr} :: 255.
\]

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 \( \lambda \)Prolog

- Usual constants associated with \( \text{int}, \text{real}, \text{string}, \text{etc} \)
- The following polymorphic constants of \( \text{list} \) type
  \[
  \begin{align*}
  \text{type} \ & \text{nil} \quad (\text{list A}) \\
  \text{type} :: \ & \quad (\text{A} \to (\text{list A}) \to (\text{list A})).
  \end{align*}
  \]
- The following \( \text{logical} \) constants
  \[
  \begin{align*}
  \text{type} \ & \text{true} \quad \text{o}.
  \text{type} \ & \&, ;, =>, :-, , \quad \text{o} \to \text{o} \to \text{o}.
  \text{type} \ & \sigma, \pi \quad (\text{A} \to \text{o}) \to \text{o}.
  \text{infixl} , \quad 110.
  \text{infixr} \& \quad 120.
  \text{infixl} ; \quad 100.
  \text{infixl} :- \quad 0.
  \text{infixr} => \quad 130.
  \end{align*}
  \]

\( \Rightarrow \) 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

\[
\begin{align*}
\text{append} \ & \text{nil} \ L. \\
\text{append} \ (X::L1) \ L2 \ (X::L3) \ & \quad \text{append} \ L1 \ L2 \ L3.
\end{align*}
\]

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

\[
?\text{-} \quad \text{append} \ (1::\text{nil}) \ (2::\text{nil}) \ L.
\]

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

A Conservative Extension to \( \text{hohh} \) 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

\[
\text{reverse} \ L1 \ L2 :- \\
(\pi \text{rev}_\text{aux} \ \\
(((\pi L \ \text{rev}_\text{aux} \text{nil} \ L \ L)) \ & \\
(\pi X \ \pi L1 \ \pi L2 \ \pi L3 \ \text{rev}_\text{aux} \ (X::L1) \ L2 \ L3 \ :- \\
\text{rev}_\text{aux} \ L1 \ (X::L2) \ L3)) \\
\Rightarrow \text{rev}_\text{aux} \ L1 \ [\ ] \ L2).
\]

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*:
  ```prolog
  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*:
  ```prolog
  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

---

Animations 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](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
```prolog
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
```prolog
parent bob mary.
parent sue peter.
parent mary john.
parent peter james.
```

The following are then sensible goals
```prolog
?- 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
\(\text{(bob::sue::nil)} \ (\text{mary::peter::nil)}).\)

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:

\(P = x \ y \ \text{true}\)

---

Mapping Function Evaluation

There is also a functional counterpart to *mappred*:

\[
\text{type } \text{mapfun} \ (A \to B) \to \text{list A} \to \text{list B} \to o.
\]

\[
\text{mapfun } F \ nil \ nil.
\]

\[
\text{mapfun } F \ (X::L1) \ ((F X)::L2) :- \text{mapfun } F \ L1 \ L2.
\]

Now computation over list elements involves only \(\beta\)-conversion.

For example consider the following query:

\[
\text{mapfun } (x \ (x + 5)) \ [3,7] \ L
\]

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

\[
\text{mapfun } F \ [3,7] \ [3 + 5, 7 + 5]
\]

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:

\[
\text{type } = \ A \to A \to o.
\]

\[
\text{infix } = 50.
\]

\[
X = X.
\]

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 \(\lambda\)-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 \(\lambda\)-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 $\lambda$-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 $\lambda$-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” $\lambda$-Prolog, which actually pioneered the HOAS idea, supports a particular variant that is free of such problems.

$\lambda$-Tree Syntax

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

$\lambda$Prolog supports this idea through the following:

- only a very weak $\lambda$-calculus is used
- $\lambda$-terms are for representation, not for evaluation
- $\lambda$-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 $\lambda$Prolog signature defines a representation:

<table>
<thead>
<tr>
<th>Kind</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>term</td>
<td>form</td>
</tr>
<tr>
<td>ff, tt</td>
<td>form</td>
</tr>
<tr>
<td>&amp; &amp;</td>
<td>form -&gt; form -&gt; form</td>
</tr>
<tr>
<td>all, some</td>
<td>(term -&gt; form) -&gt; form</td>
</tr>
<tr>
<td>p</td>
<td>term -&gt; form</td>
</tr>
<tr>
<td>f</td>
<td>term -&gt; term</td>
</tr>
<tr>
<td>a</td>
<td>term</td>
</tr>
</tbody>
</table>

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

```
( all x \ some y \ ((p (f x)) \supset (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 $\lambda$Prolog
- 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

```prolog
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 $\beta$-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

```prolog
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
    
    \[ \text{A substitution instance of a goal} \ (\text{all } P) \ \text{is a goal} \]

- 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

\[ Q_1 x_1 \ldots Q_n x_n M \]

where $Q_i$ 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 $\lambda$Prolog clause using the dummy constant

```prolog
pnf (all C) (all B) :~ pnf (C dummy) (B dummy).
```

But now consider the query

```prolog
?- pnf (all X \ (p X)) F.
```

Mobility of Binders

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

In $hohn$, this mobility is paralleled by one from terms to formulas to derivations

For example, consider a sequent calculus theorem prover

```prolog
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 \( \lambda \)Prolog 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 ty tm --> tm.
```

Thus, \( (\lambda x : int \rightarrow \lambda y : int \rightarrow int \rightarrow y x) \) 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 \( \lambda \)Prolog program constitutes a specification of call-by-name evaluation and typing:

```
type eval tm --> tm o.
type of tm --> ty 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 \( \beta \)-conversion to realize substitution and the treatment of recursion.

The latter will give us substitution theorems for free in reasoning.

The L\( \lambda \) Sub-Language of \textit{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 \( \lambda \)Prolog programs satisfying the following restriction:

\textit{Essentially existentially quantified variables appear applied only to distinct \( \lambda \)-bound variables or variables essentially universally bound within their scope.}

For example, consider:

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

The (dynamic) L\( \lambda \) language is \( \lambda \)Prolog when it satisfies this property (dynamically).

Interesting Properties of L\( \lambda \)

- Equations that arise in the L\( \lambda \) setting have unique solutions up to renaming; e.g. contrast the following:
  
  \( ?- pi g\ pi a\ sigma F\ (F a) = (g a a) \).
  \( ?- pi g\ sigma F\ pi a\ (F a) = (g a a) \).
  \( ?- pi g\ sigma F\ pi a\ (G a a) = (g a a) \).

- \( \beta \)-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 \( \beta_0 \)-redexes.

- Substitution is one case that violates this restriction, e.g.
  
  \textit{proves Gamma (some P) :- proves Gamma (P T).}

  However, here the restriction will typically be met dynamically.
Examples of Specifications in $\lambda$Prolog

Gopalan Nadathur
Specification and Reasoning

Process Expressions in the $\pi$-Calculus

A language for modelling processes that interact using names
Two important syntactic categories: names and processes
The process expressions in the finite $\pi$-calculus:

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

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 $\lambda$Prolog

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 $\pi$-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

$\tau$ 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

\[
\begin{align*}
\text{INPUT-ACT,} & \quad w \text{ not free in } (z)P \\
x(z).P & \longrightarrow P\{w/z\} \\
\text{OPEN,} & \quad x \neq y, w \text{ not free in } (y)P \\
(y)P & \xrightarrow{\bar{x}(w)} P'\{w/y\} \\
\text{CLOSE} & \\
P & \xrightarrow{\bar{x}(w)} P' \quad Q & \xrightarrow{\bar{x}(w)} Q' \\
\tau & \quad \longrightarrow \quad (w)(P'|Q') \\
\text{CLOSE} & \\
\end{align*}
\]

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 \( \lambda \text{Prolog} \)

The key idea in capturing the interaction of bounded actions is that 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:

- \text{kind action} type.
- \text{type} \tau \quad \text{action}.
- \text{type} \up, \dn \quad 
  \begin{align*}
  \text{name} & \rightarrow \text{name} & \rightarrow \text{action}.
\end{align*}
- \text{type} \one \quad \text{proc} & \rightarrow \text{proc} & \rightarrow \text{proc} & \rightarrow o.
- \text{type} \onenep \quad 
  \begin{align*}
    \text{proc} & \rightarrow (\text{name} & \rightarrow \text{action}) & \\
    & \rightarrow (\text{name} & \rightarrow \text{proc}) & \rightarrow o.
  \end{align*}

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

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:

\[
\begin{align*}
\onenep & \ (in \ X \ M) \ (dn \ X) \ M. \\
\onenep & \ (nu \ P) \ (up \ X) \ P' :- \\
& \quad \pi y \ \one \ (P \ y) \ (up \ X \ y) \ (P' \ y). \\
\one & \ (par \ P \ Q) \ tau \ (nu \ y \ \par \ (P' \ y) \ (Q' \ y)) & \& \\
\one & \ (par \ Q \ P) \ tau \ (nu \ y \ \par \ (Q' \ y) \ (P' \ y)) :- \\
& \quad \onenep \ P \ (up \ X) \ P', \onenep \ Q \ (dn \ X) \ Q'.
\end{align*}
\]

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.

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

- The free output action will yield a clause only for \( \onenep \ \ \text{OUTPUT-ACT} \)

\[
\begin{align*}
\bar{x}y.P & \longrightarrow P \\
\end{align*}
\]

- The free output action will yield a clause only for \( \onenep \ \ \text{OUTPUT-ACT} \)

\[
\begin{align*}
\bar{x}y.P & \longrightarrow P \\
\end{align*}
\]

- Bound input and free output actions can also interact

\[
\begin{align*}
P & \xrightarrow{\bar{x}y} P' \quad Q & \xrightarrow{x(z)} Q' \\
\tau & \quad \longrightarrow \quad (Q'[y/z]) \\
\text{COM} & \\
P & \xrightarrow{\bar{x}y} P' \quad Q & \xrightarrow{x(z)} Q' \\
\tau & \quad \longrightarrow \quad (Q[y/z]) | P \\
\end{align*}
\]

- Bound input and free output actions can also interact

\[
\begin{align*}
P & \xrightarrow{\bar{x}y} P' \quad Q & \xrightarrow{x(z)} Q' \\
\tau & \quad \longrightarrow \quad (Q'[y/z]) \\
\text{COM} & \\
P & \xrightarrow{\bar{x}y} P' \quad Q & \xrightarrow{x(z)} Q' \\
\tau & \quad \longrightarrow \quad (Q[y/z]) | P \\
\end{align*}
\]

- Bound input and free output actions can also interact

\[
\begin{align*}
P & \xrightarrow{\bar{x}y} P' \quad Q & \xrightarrow{x(z)} Q' \\
\tau & \quad \longrightarrow \quad (Q'[y/z]) \\
\text{COM} & \\
P & \xrightarrow{\bar{x}y} P' \quad Q & \xrightarrow{x(z)} Q' \\
\tau & \quad \longrightarrow \quad (Q[y/z]) | P \\
\end{align*}
\]
Encoding Transition Rules in \( \lambda \)Prolog (Continued)

- The “congruence” over a choice must be reflected on both one and onep

  \[
  \text{SUM} : \begin{array}{c}
P \xrightarrow{\alpha} P' \\
P + Q \xrightarrow{\alpha} P'
\end{array}
\]

  \[
  \begin{array}{c}
  P + Q \xrightarrow{\alpha} P' \\
  Q + P \xrightarrow{\alpha} P'
\end{array}
\]

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

- A similar kind of congruence applies to parallel composition

  \[
  \text{PAR} : \begin{array}{c}
P \xrightarrow{\alpha} P' \\
Q \xrightarrow{\alpha} Q'
\end{array}
\]

  \[
  \begin{array}{c}
P|Q \xrightarrow{\alpha} P'|Q \\
Q|P \xrightarrow{\alpha} Q|P'
\end{array}
\]

  one (par P Q) A (par P' Q) &
  onep (par P Q) A (par P' Q') :- 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 \( \pi \)-Calculus Specifications

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

\( \lambda \)Prolog 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, showing 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 \( \lambda \)Prolog

We have already seen how to use \( \lambda \)-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 \( \lambda \)-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

\[
(\text{of } \text{tt bool)} & (\text{of } \text{ff bool}).
\]

\[
\text{of } (\text{i I) int.}
\]

\[
\text{of } \text{sum } (\text{int } --> \text{int} --> \text{int}).
\]

\[
\ldots
\]

\[
\text{of } (\text{cond } C T E) A :-
\]

\[
\quad \text{of } C \text{ bool, of } T A, \text{ of } E A.
\]

\[
\text{of } (\text{fix } E) \text{ Ty} :-
\]

\[
\quad \text{pi } x \\text{l} \text{of } x \text{Ty }\Rightarrow \text{of } (E \text{ x) Ty}.
\]

\[
\ldots
\]

\[
\text{eval } (\text{app } (\text{app } \text{sum } E1) E2) \text{ (i I)} :-
\]

\[
\quad \text{eval } E1 \text{ (i I1), eval } E2 \text{ (i I2), I is I1 + I2}.
\]

\[
\text{eval } (\text{cond } C T _) V :- \text{eval } C \text{ tt, eval } T V.
\]

\[
\text{eval } (\text{cond } C _ E) V :- \text{eval } C \text{ ff, eval } E V.
\]

\[
\text{eval } (\text{fix } E) V :- \text{eval } (E \text{ (fix } E)) V.
\]
Small-Step Evaluation via Evaluation Contexts

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

\[
\text{type} \ \text{val, non_val, redex} \ \text{tm} \rightarrow \text{o.}
\]
\[
\text{type} \ \text{reduce, eval} \ \text{tm} \rightarrow \text{tm} \rightarrow \text{o.}
\]
\[
\text{type} \ \text{context} \ \text{tm} \rightarrow (\text{tm} \rightarrow \text{tm}) \rightarrow \text{tm} \rightarrow \text{o.}
\]
\[
\text{context} \ (x \ x) \ R :- \text{redex} \ R.
\]
\[
\text{context} \ (\text{cond} \ M \ N \ P) \ (x \ (\text{cond} \ (E \ x) \ N \ P)) \ R :-
\]
\[
\text{context} \ (\text{app} \ M \ N) \ (x \ (\text{app} \ (E \ x)) \ N) \ R :-
\]
\[
\text{context} \ (\text{app} \ V \ M) \ (x \ (\text{app} \ V \ (E \ x)) \ R :-
\]
\[
\text{eval} \ V \ V :- \text{val} \ V
\]
\[
\text{eval} \ M \ V :- \text{context} \ M \ E \ R, \text{reduce} \ R \ N, \text{eval} \ (E \ N) \ V.
\]

Here \text{non_val, val, and redex} recognize non-values, values and redexes at the root and \text{reduce} contracts redexes.

Recognizing Tail Recursive Structure

In an expression of the form \((\text{fix} \ (f \ F))\) 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 \text{term}.

\[
\text{type} \ tr, fn, trabs, \text{headrec, trbody} \ \text{tm} \rightarrow \text{o.}
\]
\[
tr \ (\text{fix} \ M) :- \pi \ F \ ((fn \ F) \Rightarrow (\text{trabs} \ (M \ F))).
\]
\[
\text{trabs} \ (abs \ R) :- \pi \ X \ ((\text{term} \ X) \Rightarrow (\text{trabs} \ (R \ X))).
\]
\[
\text{trbody} \ (\text{cond} \ M \ N \ P) :- \text{term} \ M, \text{trbody} \ N, \text{trbody} \ P.
\]
\[
\text{headrec} \ (\text{app} \ M \ N) :- (fn \ M ; \text{headrec} \ M), \text{term} \ N.
\]

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 \(\lambda\)-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 \(\lambda\)-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:

\[
\text{kind} \ \text{proof} \ \text{type}.
\]
\[
\text{type} \ \text{imp}_i \ (\text{proof} \rightarrow \text{proof}) \rightarrow \text{proof}.
\]
\[
\text{type} \ \text{imp}_e \ \text{form} \rightarrow \text{proof} \rightarrow \text{proof} \rightarrow \text{proof}.
\]
\[
\text{type} \ \text{or}_i1, \ \text{or}_i2 \ \text{proof} \rightarrow \text{proof}.
\]
\[
\text{type} \ \text{or}_e \ \text{form} \rightarrow \text{form} \rightarrow \text{proof} \rightarrow \text{proof} \rightarrow \text{proof}.
\]
\[
\text{type} \ \text{all}_e \ \text{term} \rightarrow (\text{term} \rightarrow \text{form}) \rightarrow \text{proof} \rightarrow \text{proof}.
\]
\[
\text{type} \ \text{all}_i \ (\text{term} \rightarrow \text{proof}) \rightarrow \text{proof}.
\]
\[
\text{type} \ \text{some}_e \ (\text{term} \rightarrow \text{form}) \rightarrow \text{proof} \rightarrow \text{proof}.
\]
\[
\text{type} \ \text{some}_i \ \text{term} \rightarrow \text{proof} \rightarrow \text{proof}.
\]

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.
  - Tactics provide a framework for combining such tactics into larger units.
  - In the λProlog setting, definitions of tacticals use predicate variables.

The Need for Unification

Recall the ∀L and ∃R rules in the reduced proof system:

\[
\begin{align*}
\Sigma; \mathcal{P} & \vdash B[t/x] & t \text{ is a } \Sigma\text{-term} & \exists R \\
\Sigma; \mathcal{P} & \vdash \exists x \ B \\
\Sigma; \mathcal{P} & \vdash D[t/x] & t \text{ is a } \Sigma\text{-term} & \forall L \\
\Sigma; \mathcal{P} & \vdash \forall x \ D & A
\end{align*}
\]

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

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

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

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

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

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

$$
Q_1x_1 \ldots Q_nx_n [t_1 = s_1 \land \ldots \land t_m = s_m]
$$

where $Q_i$ 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:

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

do not have the same solutions.

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

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

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.
Often we will treat unification problems in which the prefix has a $\forall\exists$ 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:

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

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 $L_\lambda$.

### Comparison with Skolemization

Dual to raising that moves existential quantifiers inwards using

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

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 $z : (a \rightarrow b) \rightarrow a$, $x : a$, $y : b$, $f : a \rightarrow b$ and contrast
  $$\forall z \exists x \forall y T \quad \text{with} \quad \forall z \forall f \exists x T$$

  The second has a solution whereas the first does not.

- Assume $z : a, g : b \rightarrow a$, $x : a$, $y : b$, $f : a \rightarrow b$ and contrast
  $$\forall z \forall g \exists x \forall y T \quad \text{with} \quad \forall z \forall g \forall f \exists x T$$

  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 $\lambda$-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

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

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

\[ \forall a, \forall g, \exists F \forall i \rightarrow i [F(a) = (g\ a\ a)] \]

Four incomparable unifiers exist for this problem

- “Complete sets of unifiers” may be infinite, e.g. consider

\[ \forall u, \exists F \forall w [u(Fw) = F(uw)] \]

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

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 \( \lambda Prolog \)

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

\[ t_1 = s_1 \land \ldots \land t_n = s_n \]

- 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 \( \alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \beta \) for \( \beta \) 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 \( \gamma_1 \rightarrow \ldots \rightarrow \gamma_m \rightarrow \beta \).

Let \( H_i \) be a fresh variable of type \( \alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \gamma_i \) 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 \( \alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \beta \) for \( \beta \) an atomic type.

Let \( \alpha_i \) be \( \gamma_1 \rightarrow \ldots \rightarrow \gamma_k \rightarrow \delta \), for \( \delta \) an atomic type.

A projection onto the \( i^{th} \) 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 \( \alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \gamma_i \) for \( i \leq m \).

The \( i^{th} \) 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 \( \forall g \exists a \forall F (F a) = (g a a) \).

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 forever; for example, consider the imitation substitution \( \lambda x \ u (H x) \) for \( F \) for
  \( \forall u \exists F (F a) \forall w [u (F w) = F (u w)] \).

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 \( L_\lambda \) language.

In fact, in this setting, there are even most general unifiers.
Higher-Order Pattern Unification

Unification problems that arise in the L_\lambda setting are called higher-order pattern unification problems.

We will observe that under the L_\lambda 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.

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 \(c\)s 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 \(c_1, \ldots, c_n\) and \(d_1, \ldots, d_m\) 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 \(c\)s and in the \(d\)s.

Let \(a_1, \ldots, a_k\) be a listing of such variables.

An MGU for the problem then is the substitution

\[ \{ F \mapsto \lambda c_1 \ldots \lambda c_n (H \ a_1 \ldots a_k), \ G \mapsto \lambda d_1 \ldots \lambda d_m (H \ a_1 \ldots a_k) \} \]

where \(H\) is a new existentially quantified variable.

Flexible-Rigid HOPU Problems

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.

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.
Occurs-Check and Termination

Because substitutions produce only $\beta_0$-reductions, there is hope that 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. Consider:

$$\forall f \rightarrow \exists X, X = (f X)$$

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:

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

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.

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

\[
\text{nat } z. \\
\text{nat } (s \text{ N}) \leftarrow \text{nat } \text{N}.
\]

\[
\text{plus } z \text{ L } L. \\
\text{plus } (s \text{ N}) \text{ M } (s \text{ P}) \leftarrow \text{plus } \text{N M P}.
\]

we want to be able to prove not only

\[
\pi \text{ L} \leftarrow (\text{plus } z \text{ L } L)
\]

but also the following

\[
\pi \text{ L} \leftarrow (\text{nat } \text{L}) \rightarrow (\text{plus } \text{L } z \text{ L})
\]

Call-by-name evaluation and typing were rendered into \textit{hohh} specifications as follows

\[
\text{eval } (\text{abs } T \text{ R}) (\text{abs } T \text{ R}). \\
\text{eval } (\text{app } T1 \text{ T2}) V \leftarrow \\
\quad \text{eval } T1 (\text{abs } R), \text{eval } (R \text{ T2}) V.
\]

\[
\text{of } (\text{app } T1 \text{ T2}) \text{ Ty2} \leftarrow \\
\quad \text{of } T1 (\text{Ty1 } \rightarrow \text{ Ty2}), \text{of } T2 \text{ Ty1}.
\]

\[
\text{of } (\text{abs } \text{Ty1 } \text{R}) (\text{Ty1 } \rightarrow \text{ Ty2}) \leftarrow \\
\quad \pi x \leftarrow (\text{of } \text{x Ty1}) \rightarrow (\text{of } (R \text{ x}) \text{ Ty2}).
\]

In this setting, we want to be able to prove

\[
\forall e \forall v \forall \text{ty} ((\text{eval } e \text{ v}) \land (\text{of } e \text{ ty}) \supset (\text{of } v \text{ ty}))
\]

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 \text{eval } e \text{ 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

\[
\text{eval } (\text{abs } T \text{ R}) (\text{abs } T \text{ R}).
\]

Here \text{e} and \text{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 \text{e} and \text{v}.
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

\[
\text{of } (\text{abs } T y_1 \ R) \ (T y_1 \rightarrow T y_2) :-
\]

\[
\text{pi } x \ (\text{of } x \ T y_1) \Rightarrow (\text{of } (R \ x) \ T y_2).
\]

- 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 ∇ \( x \) \( B \):

*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) \vdash B[a/x]) \vdash \Delta}{\Sigma; \Gamma, (\sigma \vdash \Delta) \vdash \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \nega...
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

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

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

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

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

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.

\[
\frac{\Sigma; \Gamma, (\sigma \triangleright B) \vdash (\sigma \triangleright B)}{\Sigma; \Gamma_1, (\sigma \triangleright B) \vdash \Delta, \Sigma; \Gamma_2, (\sigma \triangleright B) \vdash \Delta} \quad \text{cut}
\]

That

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

Local Contexts and Propositional Rules

Essentially, the local context must distribute over the connective

\[
\begin{align*}
\vdash R & \quad \vdash L \\
\Sigma; \Gamma \vdash (\sigma \triangleright \top) & \vdash R \\
\Sigma; \Gamma, (\sigma \triangleright \bot) & \vdash \Delta \quad \vdash L \\
\Sigma; \Gamma, (\sigma \triangleright B_1) & \vdash \Delta \\
\Sigma; \Gamma, (\sigma \triangleright B_2) & \vdash \Delta \\
\Sigma; \Gamma, (\sigma \triangleright B_1 \lor B_2) & \vdash \Delta \quad \lor R \\
\Sigma; \Gamma, (\sigma \triangleright B_1 \land B_2) & \vdash \Delta \quad \land L \\
\Sigma; \Gamma, (\sigma \triangleright B_1 \land B_2) & \vdash \Delta \quad \land R
\end{align*}
\]

Adding Definitions to the Logic

Predicate constants will be interpreted via clauses of the form

\[\forall x. H \triangleq B\]

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 \( D \) 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 $lv$ be a function that assigns natural numbers or “levels” to predicate constants

We extend this assignment into one for arbitrary formulas

- $lv(T) = lv(\bot) = 0$
- $lv(p, t_1 \ldots t_n) = lv(p)$
- $lv(B \land C) = max(lv(B), lv(C))$
- $lv(B \lor C) = max(lv(B), lv(C))$
- $lv(\forall x B) = lv(\exists x B) = lv(\neg x B) = lv(B)$

A definition $D$ is then good if it satisfies the following condition

**There is some assignment $lv$ to predicates such that**

$$\forall \bar{x}. H \triangleq B \in D \implies lv(H) \leq lv(B)$$

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

Introduction Rules for Atomic Formulas (Continued)

Given a set of clauses $D$, let $\sigma \triangleright D$ denote the raising of each member of $D$ over $\sigma$

Then the introduction rules for atomic formulas are the following

$$\Sigma; \Gamma \vdash (\sigma \triangleright B)\theta$$

**defR**

$$\Sigma; \Gamma \vdash \sigma \triangleright A \implies H_{\theta}, \forall \bar{x}. H \triangleq B \in \sigma \triangleright D$$

$$\{\Sigma; \theta, (\sigma \triangleright B)\theta \vdash C\theta \mid (\sigma \triangleright A)\theta = H_{\theta}, \forall \bar{x}. H \triangleq B \in \sigma \triangleright D\}$$

**defC**

Comments

- **defR** has the flavour of backchaining, **defC** of case analysis
- **defC** considers substitutions for eigenvariables
- **defC** must consider all clauses and all substitutions
- For cut-free proofs, it suffices to consider only complete sets of unifiers

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 $\forall x_1 \ldots \forall x_n . H \triangleq B$ over $\sigma$ is given as follows

$$\forall h_1 \ldots \forall h_n . \sigma \triangleright (H_{\theta}) \triangleq \sigma \triangleright (B_{\theta})$$

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 $\theta$ to $x_1, \ldots, x_n \triangleright B$, denoted by $(x_1, \ldots, x_n \triangleright B)\theta$, is given as follows

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

**Definition**

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

Using Definitions (Example)

Suppose that the set of definition clauses is

$$p a \triangleq T, \quad p b \triangleq T, \quad q a \triangleq T, \quad q b \triangleq T, \quad q c \triangleq T$$

Then $\vdash \triangleright \forall x p x \triangleright q x$ is derivable

In fact, the following is a derivation for it:

$$\vdash \triangleright T \vdash \triangleright T$$

**defR**

$$\vdash \triangleright q \ a \\text{defR} \quad \vdash \triangleright q \ b \\text{defR}$$

$$\vdash \forall x p x \triangleright q x \\forall R, \triangleright R$$

Notice that the **defC** 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

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:
\[
G ::= \top | \bot | A | G \land G | G \lor G | \exists x \ G | \forall x \ G
\]

Level 1:
\[
D ::= \top | \bot | A | D \land D | D \lor D | G \supset D |
\exists x \ D | \forall x \ D
\]

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

\[
\vdash \sigma \ D
\]

where \( D \) is a level 1 formula

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 \( \pi \)-calculus
- identifying winning strategies in games
- reasoning about security protocols: the SPEC system

Limitations of Bedwyr

- Eigenvariables and logic variables used for full automation lead to incompleteness if present together.
  For example, consider proving the formula
  \[ \forall x \exists y \,(px \land py \land x = y \supset \bot) \]
  where \( p \) is defined as \( \{ pa \overset{\Delta}{=} T, pb \overset{\Delta}{=} T, pc \overset{\Delta}{=} T \} \).
  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.

Definitions and Equality

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

\[ \forall \vec{x}.p\vec{x} \triangleq B\ p\ \vec{x} \]

where \( p \) and \( \vec{x} \) do not appear in \( B \).

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

\[ \frac{\Sigma; \Gamma, \sigma \triangleright B\ p\ \vec{t} \triangleright \Delta}{\Sigma; \Gamma, \sigma \triangleright \ p\ \vec{t} \triangleright \Delta} \quad \text{defL} \]

\[ \frac{\Sigma; \Gamma \triangleright \sigma \triangleright B\ p\ \vec{t} \triangleright \Delta}{\Sigma; \Gamma \triangleright \sigma \triangleright p\ \vec{t} \triangleright \Delta} \quad \text{defR} \]

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

Definitions and Equality (Example)

For example, consider the clauses

\[ \text{nat} \ 0 \overset{\Delta}{=} T \quad \forall N.\text{nat} \ (s\ N) \overset{\Delta}{=} \text{nat} \ N \]

These can be transformed into

\[ \forall N.\text{nat} \ N \overset{\Delta}{=} (N = 0) \lor \exists N'.\ (N = s\ N' \land \text{nat}\ N') \]

The following derivation then realizes the effect of the earlier defL rule:

\[ \frac{\{ \Sigma; \sigma \triangleright (N' \theta); \Gamma \theta \triangleright (\lambda x.\ x \theta) \triangleright \Delta \mid t\theta = (s\ N' \theta) \}}{\Sigma; \Gamma \triangleright \exists N'.\ (t = s\ N' \land \text{nat}\ N') \triangleright \Delta} \quad \text{eqL} \]

A similar observation holds with respect to the earlier defR rule.
**Inductive and Co-Inductive Definitions**

The unfolding rules amount to treating $\forall \overline{x}. p \overline{x} \triangleq B p \overline{x}$ 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:

$$ \mu L $$

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

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

$$ \nu R $$

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

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 $p \overline{x}$ 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 $\mu$, $\nu$ or $\triangleq$).
- The earlier stratification condition holds: $\text{lvl}(B p \overline{x}) \leq \text{lvl}(p)$.
- "Mutual recursion" is prohibited: $\text{lvl}(B (\lambda \overline{x} \top) \overline{x}) < \text{lvl}(p)$.

We will assume these conditions implicitly henceforth.

**An Example of Inductive Reasoning**

Consider the following definition:

$$ \forall I, J, K. plus I J K \triangleq (I = 0 \land J = K) \lor \exists M \exists N I = (s M) \land K = (s N) \land plus M J N $$

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

$$ \vdash plus I J K \Rightarrow \forall L plus I J L \Rightarrow K = L $$

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

$$ I, J, K; plus I J K \vdash plus I J L \Rightarrow K = L $$

Now use induction on $plus$ with the invariant $D \triangleq \lambda I \lambda J \lambda K. \forall L plus I J L \Rightarrow K = L$

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

$$ \forall I, J, K. plus I J K \triangleq B plus I J K $$

where

$$ B = \lambda P \lambda I \lambda J \lambda K. ((I = 0 \land J = K) \lor \exists M \exists N I = (s M) \land K = (s N) \land plus M J N) $$

Then the proof at the end has the following shape:

$$ L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L, M, N; B D L M N \vdash D L M N \overline{i} I, J, K; D I J K \vdash D I J K \text{ init } L,
Another Example of Inductive Reasoning

Consider the following definitions
\[ \forall N. \text{nat } N \overset{\text{def}}{=} (N = 0) \lor \exists N'. N = (s N') \land \text{nat } N' \]
\[ \forall I, J, K. \text{plus } I J K \overset{\text{def}}{=} (I = 0 \land J = K) \lor \exists M \exists N. I = (s M) \land K = (s N) \land \text{plus } M J N \]

We can then show that \text{plus} is total on natural numbers

We then can show that \text{plus} is total on natural numbers

An Easier-to-Use Derived Rule for Induction

Suppose the objective is to prove a formula of the form
\[ \forall x. H_1 \supset \cdots \supset (p \overset{\text{def}}{=} \bar{u}) \supset \cdots \supset H_n \supset A \]
using induction on \( p \) which is defined as \( \forall y. p \overset{\text{def}}{=} B p \bar{y} \)

Then, this formula can be changed to
\[ \forall x. H_1 \supset \cdots \supset (p^0 \overset{\text{def}}{=} \bar{u}) \supset \cdots \supset H_n \supset A \]

and the formula
\[ \forall x. H_1 \supset \cdots \supset (p^* \overset{\text{def}}{=} \bar{u}) \supset \cdots \supset H_n \supset A \]

can be added to the assumptions before continuing the proof

The interpretation of \( p^0 \) and \( p^* \) here are the following

- \( p^0 \) will match with \( p \) only if it has the \( * \) annotation
- To unfold \( p^0 \bar{u} \), replace it with \( B p^* \bar{u} \)

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 \text{nat} and consider proving
\[ \forall N \forall J. \text{nat } I \supset \text{nat } J \supset \exists K. \text{plus } I J K \]

by induction on the first occurrence of \text{nat}

Then we would add as a hypothesis
\[ \neg \neg \forall N \forall J. \text{nat } I \supset \text{nat } J \supset \exists K. \text{plus } I J K \]

After some introduction rules, we would be left with proving
\[ I; \neg \neg \forall N \forall J. \text{nat } I \supset \text{nat } J \supset \exists K. \text{plus } I J K \]

Unfolding \( \text{nat}^0 \bar{u} \) now leaves us to prove the two sequents
\[ J; \neg \neg \forall N \forall J. \text{nat } I \supset \text{nat } J \supset \exists K. \text{plus } 0 J K \]

\[ N, J; \neg \neg \forall N \forall J. \text{nat } I \supset \text{nat } J \supset \exists K. \text{plus } (s N) J K \]

These sequents can be proved easily using the induction hypothesis

Some Issues with the Nabla Quantifier

The \( \forall \) 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
\[ \forall x. (\forall n. x) \supset p x \]
\[ \forall x. p x \supset (\forall n. x) \]

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
\[ \forall x. \forall y. p x y = \forall y. \forall x. p x y \]

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

\[
\frac{x; B \vdash S x \quad S; \Gamma, \sigma \triangleright S \vdash \Delta}{\Sigma; \Gamma, \sigma \triangleright p \vdash \Delta} \quad \mu L
\]

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

\[\nabla n \forall x \text{nat} (x n) \supset \exists y x = \lambda z y\]

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 these issues we extend the logic to include the following equivalences

\[\nabla x \nabla y B x y \equiv \nabla y \nabla x B x y\]

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

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 \(\nabla\).
- Use nominal constants arbitrarily for existentials, raise over the ones in the formula when introducing eigenvariables.
- When matching formulas (e.g. in the \(\text{init}\) or \(\text{cut}\) 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 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 \(\nabla\).
- Use nominal constants arbitrarily for existentials, raise over the ones in the formula when introducing eigenvariables.
- When matching formulas (e.g. in the \(\text{init}\) or \(\text{cut}\) 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.

The Logic \(LG^{\omega}\)

This logic realizes the interchange and strengthening principles related to \(\nabla\) quantification.

Formally, we change the existing reasoning logic as follows:
- We assume a denumerable number of nominal constants at each type where \(\nabla\) quantification is permitted.
- \(C\) stands for the set of all nominal constants, \(K\) 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 \(\lambda\)-conversion and a permutation mapping on nominal constants. Written as \(B' \approx B\) and read “\(B'\) is equivariant to \(B\)”. 
The Inference Rules for $LG^\omega$

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

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

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

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 $\forall$ quantifiers of smaller scope. It turns out to be adequate to raise only over the nominal constants appearing in the quantified formula.

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[\sigma] (\Gamma[\sigma], etc)

The Quantifier Inference Rules of $LG^\omega$

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

$$\frac{\Sigma; \Gamma, B[a/x] \vdash \Delta \ \forall L \ a \notin \text{supp}(B)}{\Sigma; \Gamma \vdash \forall x B \vdash \Delta}$$

$$\frac{\Sigma; \Gamma \vdash \forall x B \vdash \Delta \ \exists R \ a \notin \text{supp}(B)}{\Sigma; \Gamma, (h \sigma)/x \vdash \Delta \ \exists L} \quad \sigma = \text{supp}(B), h \text{ new}$$

A Missing Feature in $LG^\omega$

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

\[
\begin{align*}
&\text{of } \Gamma X A \equiv \text{member } (X : A) \Gamma \\
&\text{of } \Gamma (\text{app } M N) B \equiv \exists A (\text{of } \Gamma M (\text{arr } A B) \land (\text{of } \Gamma N A)) \\
&\text{of } \Gamma (\text{abs } A M) (\text{arr } A B) \equiv \forall x (\text{of } ((x : A) :: \Gamma) (M x) B)
\end{align*}
\]

Then determinateness of typing can be expressed as:

\[
\forall \Gamma \forall M \forall A \forall B \text{ ctx } \Gamma \supset (\text{of } \Gamma M A) \supset (\text{of } \Gamma M B) \supset A = B
\]

if \( \text{ctx } \Gamma \) holds iff \( \Gamma = (x_1 : A_1) :: \ldots :: (x_n, A_n) :: \text{nil} \) 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 \( \text{ctx} \) must be articulated for the reasoning to go through.

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

For example, consider the following:

\[\lambda x \text{ fresh } x T \supset S\]

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

Similarly, consider the following:

\[\lambda x \text{ name } x \supset \text{name } S\]

To satisfy this, \( S \) must be a nominal constant.

Note that \( \supset \) extends the equality relation and its “structure discerning” capabilities to terms with nominal constants.

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

This relation, written \( s \supset t \), 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 \( \tau \)
- \( s \) is \( \lambda \)-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:

\[\lambda x x \supset c_1 \quad \lambda x p x c_2 \supset p c_1 c_2 \quad \lambda x \lambda y p x y \supset p c_1 c_2\]

but the following do not hold:

\[\lambda x x \supset p c_1 c_2 \quad \lambda x p x c_2 \supset p c_2 c_1 \quad \lambda x \lambda y p x y \supset p c_1 c_1\]

Also, a substitution \( \theta \) is said to be a solution to the nominal abstraction \( s \supset t \) if \((s \supset t)[\theta] \) holds.

We will use ambiguously the notation \( s \supset t \) to denote in the logic the semantic notion just described.

The rules for introducing this relation are the following:

\[\Sigma; \Gamma \vdash s \supset t \vdash R \quad s \supset t \] holds

\[\{\Sigma \theta; \Gamma[\theta] \vdash C[\theta] \mid \theta \text{ is a solution to } (s \supset t)\} \vdash \Delta\]

Substitutions in the \( \vdash \Delta \) 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 \( s \supset t \), and
- every solution to \( s \supset t \) is “covered” over \( \Sigma \) 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

\[ \text{ctx } K \triangleq (K = \text{nil}) \lor \exists T \exists L ((\lambda x (x : T) :: L) \supset K \land \text{ctx } L) \]

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

\[ \forall \forall M \forall A \forall B \text{ctx } \Gamma \supset (\text{of } \Gamma \ M \ A) \supset (\text{of } \Gamma \ M \ B) \supset A = B \]

Clauses with Patterns versus Nominal Abstractions

The effect of a predicate definition based on n “patterned clauses” of the form

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

can be realized by the following definition in the “one clause” format

\[ \forall \vec{y}. (p \vec{y}) \triangleq \bigvee_{i \in 1, \ldots, n} \exists \vec{x}_i (\lambda \vec{z}. (p' \vec{t}) \supset p' \vec{y}) \land B_i \ p \vec{x}_i \]

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

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.

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:

\[ \text{ctx } \text{nil} \triangleq \top \]

\[ \forall L, A. (\nabla x. \text{ctx } ((x : A) :: L)) \triangleq \text{ctx } L \]

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

\[
\frac{\Sigma; \Gamma \vdash (B \ p \vec{x})[\theta]}{\Sigma; \Gamma \vdash p \vec{s}} \text{ def}_{RP} \]

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

- Left introduction rule for such a definition

\[
\frac{\{ \Sigma; \Gamma[\theta], (B \ p \vec{x})[\theta] \vdash C[\theta], \exists \vec{x}. (\nabla \vec{z}. p \vec{t}) \triangleq B \ p \vec{x} \in \mathcal{D} \text{ and } \theta \text{ is a solution to } ((\lambda \vec{z}. p \vec{t}) \supset p \vec{s}) \}}{\Sigma; \Gamma, p \vec{s} \vdash C} \text{ def}_{LP} \]

These rules can also be used for inductive and co-inductive definitions, i.e. if \( \triangleq \) or \( \subseteq \) is used in the definition instead of \( \triangleq \).
Left Rule for Inductive Definitions

Suppose that $p$ is defined by $n$ clauses of the form

$$\forall \bar{x}_i. (\neg \bar{Z}_i. (p \bar{t})) \models B_i \ p (\bar{x}_i)$$

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

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

The Two-Level Logic Approach to Reasoning

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

However, we propose instead to do the following

- Write specifications in a restriction of $hooh$ 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 $G$ 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 $hooh$ 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

$$G ::= T \mid A_r \mid G \lor G \mid G \land G \mid \exists x \ G \mid \forall x \ G \mid A_r \supset G$$
$$D ::= A_r \mid G \supset D \mid \forall x \ D$$

The sequents that arise from applying the $hooh$ operational semantics to this language have the form

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

with the following connotations

- $\Delta$ 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 $\mathcal{H}^2$ are based on the simply typed lambda calculus so the encoding of the syntax can be shallow

- Sorts in $\mathcal{H}^2$ give rise to corresponding ones in $\mathcal{G}$ with one special case
  - The type $o$ in $\mathcal{H}^2$ is represented by a type $\textit{fmla}$ in $\mathcal{G}$
- Constants in $\mathcal{H}^2$ carve out ones in $\mathcal{G}$ with one proviso
  - Logical constants become nonlogical ones yielding terms of type $\textit{fmla}$
- The sort $\textit{atm}$ is used for atomic formula from $\mathcal{H}^2$
  - $\langle A \rangle$ makes a $\textit{fmla}$ from an $\textit{atm}$ $A$ and $\supset$ is encoded via a constant of type $\textit{atm} \rightarrow \textit{fmla} \rightarrow \textit{fmla}$
- Eigenvariables in $\mathcal{H}^2$ 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 the Operational Semantics

Three special definitions are used in the encoding of $\mathcal{H}^2$ derivations

- For an special sort $\textit{nt}$ and $z : \textit{nt}$, $s : \textit{nt} \rightarrow \textit{nt}$ and $\textit{nat} : \textit{nt} \rightarrow o$
  - $\textit{nat} z \triangleq \top$
  - $\textit{nat} (s \textit{N}) \triangleq \textit{nat} \textit{N}$

For encoding induction on the lengths of $\mathcal{H}^2$ derivations

- For $\textit{lt}$ of type $\textit{nt} \rightarrow \textit{nt} \rightarrow o$
  - $\textit{lt} z (s \textit{N}) \triangleq \top$
  - $\textit{lt} (s \textit{M}) (s \textit{N}) \triangleq \textit{lt} \textit{M} \textit{N}$

Helps in realizing strong induction over $\mathcal{H}^2$ derivations

- For sort $\textit{atmlist}$, $\textit{nil} : \textit{atmlist}$, $\cdot$ an infix operator of type $\textit{atm} \rightarrow \textit{atmlist} \rightarrow \textit{atmlist}$ and $\textit{member} : \textit{atm} \rightarrow \textit{atmlist} \rightarrow o$
  - $\textit{member} A (A :: L) \triangleq \top$
  - $\textit{member} A (B :: L) \triangleq \textit{member} A L$

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

Encoding $\mathcal{H}^2$ Programs

For each clause $\forall x \ ; G_1 \supset \cdots \supset G_n \supset A$ in the fixed $\mathcal{H}^2$ specification $\Delta$, we will use a clause of the form

$$\text{prog} \ A \ (G_1 \wedge \ldots \wedge G_n) \triangleq \top$$

Here, $\text{prog}$ has type $\textit{atm} \rightarrow \textit{fmla} \rightarrow o$

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

$$\text{prog} \ (\text{of} \ (\text{abs} \ A \ M) \ (\text{arr} \ A \ B)) \triangleq \top$$

$$\text{prog} \ (\text{of} \ (\text{app} \ M \ N)) \triangleq \top$$

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

Encoding the Operational Semantics (Continued)

Let $\textit{seq} : \textit{nt} \rightarrow \textit{atmlist} \rightarrow \textit{fmla} \rightarrow o$ be defined by the clauses

$$\textit{seq} \ (s \textit{N}) \textit{L} \top \triangleq \top$$

$$\textit{seq} \ (s \textit{N}) \textit{L} (G_1 \vee G_2) \triangleq \textit{seq} \textit{N} \textit{L} G_1$$

$$\textit{seq} \ (s \textit{N}) \textit{L} (G_1 \vee G_2) \triangleq \textit{seq} \textit{N} \textit{L} G_2$$

$$\textit{seq} \ (s \textit{N}) \textit{L} (G_1 \wedge G_2) \triangleq \textit{seq} \textit{N} \textit{L} G_1 \wedge \textit{seq} \textit{N} \textit{L} G_2$$

$$\textit{seq} \ (s \textit{N}) \textit{L} (\textit{A} \supset \textit{G}) \triangleq \textit{seq} \textit{N} (\textit{A} :: \textit{L}) \textit{G}$$

$$\textit{seq} \ (s \textit{N}) \textit{L} (\forall x \textit{B} \textit{x}) \triangleq \forall x \textit{seq} \textit{N} \textit{L} (\textit{B} \textit{x})$$

$$\textit{seq} \ (s \textit{N}) \textit{L} \langle \textit{A} \rangle \triangleq \textit{member} \textit{A} \textit{L}$$

$$\textit{seq} \ (s \textit{N}) \textit{L} \langle \textit{A} \rangle \triangleq \exists b \textit{prog} \textit{A} \textit{b} \wedge \textit{seq} \textit{N} \textit{L} \textit{b}$$

Then $\mathcal{H}^2$ derivability of $\Sigma \vdash \Delta \vdash \textit{G}$ amounts (for suitable $\textit{prog}$ clauses) to the $\mathcal{G}$-provability of $\exists n ((\textit{nat} \textit{n}) \wedge (\textit{seq} \textit{n} \langle \textit{L} \rangle \langle \textit{G} \rangle))$.

We will abbreviate the latter by $\textit{L} \vdash \textit{G}$, assuming a fixed $\Delta$. 

Proving Properties of Specifications

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

For example, given $\text{prog}$ clauses encoding evaluation and typing, subject reduction can be expressed via the statement

$$\forall e \forall v \forall ty \left( \vdash \langle \text{eval} \ e \ v \rangle \supset (\vdash \langle \text{of} \ e \ ty \rangle) \supset (\vdash \langle \text{eval} \ v \ ty \rangle) \right)$$

Proofs of such properties will typically involve induction on the lengths of $\mathcal{H}^2$ derivations

For example, the subject reduction formula is equivalent to

$$\forall e \forall v \forall ty \forall n \left( \text{nat} \ n \land \text{seq} \ n \ \text{nil} \ \langle \text{eval} \ e \ v \rangle \supset (\vdash \langle \text{of} \ e \ ty \rangle) \supset (\vdash \langle \text{eval} \ v \ ty \rangle) \right)$$

If we try to prove this by induction on $\text{nat} \ n$, this will amount to induction on the length of the $\mathcal{H}^2$ derivation of $\text{eval} \ e \ v$

Using Strong Induction on Derivation Lengths

To prove the subject reduction formula, we will actually need to unfold the $\text{seq}$ definition twice

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

To deal with this, we prove instead the formula

$$\forall e \forall v \forall ty \forall n \forall m \left( \text{nat} \ n \land (\text{lt} \ m \ n) \land \text{seq} \ m \ \text{nil} \ \langle \text{eval} \ e \ v \rangle \supset (\vdash \langle \text{of} \ e \ ty \rangle) \supset (\vdash \langle \text{eval} \ v \ ty \rangle) \right)$$

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

$$\forall \Gamma \forall \Delta \forall G \left( \forall X \ \text{member} \ X \ \Gamma \supset \text{member} \ X \ \Delta \right) \supset \Gamma \vdash G \supset \Delta \vdash G$$

**Cut Admissibility**

$$\forall \Gamma \forall A \forall G \left( A :: \Gamma \right) \vdash G \supset \Gamma \vdash A \supset \Gamma \vdash G$$

**Instantiation**

$$\forall \Gamma \forall G \left( \forall x \left( G \ x \right) \vdash (G \ t) \right) \supset \forall t \left( \Gamma \ t \right) \vdash (G \ t)$$

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

$$\left( \forall x \left( \text{of} \ x \ a \right) :: L \vdash \langle \text{of} \ (T \ x) \ B \rangle \supset (L \vdash \langle \text{of} \ u \ a \rangle) \supset L \vdash \langle \text{of} \ (T \ u) \ B \rangle \right)$$

The Abella System

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

- It automatically creates $\text{prog}$ clauses from $\mathcal{H}^2$ specs
- It treats judgements of the form $L \vdash G$ in a special way, building in the processing of $\text{seq}$ and $\text{prog}$ 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 $\lambda$-tree syntax simplifies this task.

- The encoding of $hH^2$ into $G$ must be adequate.
  In particular, theorems proved in $G$ 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 $G$.
  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.