October 1988

Logic Programming Based on Higher-Order Hereditary Harrop Formulas

Dale Miller
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-77.

Logic Programming Based on Higher-Order Hereditary Harrop Formulas

Abstract

Hereditary Harrop formulas are an extension to Horn clauses in which the body of clauses can contain implications and universal quantifiers. These formulas can further be extended by embedding them in a higher-order logic; that is, by permitting quantification over function symbol occurrences and some predicate symbol occurrences, and by replacing first-order terms with simply typed λ-terms. Our justification for considering this rich extension of Horn clause theory as a satisfactory logic programming language is provided by a proof-theoretic notion we call "uniform proofs". This notion will be defined and motivated. This extended language can provide very natural and direct implementations of various kinds of abstraction mechanisms. For example, higher-order hereditary Harrop formulas (hohh) can be used to support aspects of modular programming, abstract data types, and higher-order programming.

We have designed and built a logic programming system which implements hohh in much the same way Prolog implements first-order Horn clauses. This language and its interpreter, collectively called λProlog, will be described. We will present several example programs where λProlog provides a much more immediate and satisfactory implementation language than first-order Prologs. These examples are taken from theorem proving and program transformation. Finally, we will describe some aspects of our implementation of λProlog.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-77.

LOGIC PROGRAMMING BASED ON HIGHER-ORDER HEREDITARY HARROP FORMULAS

Dale Miller

MS-CIS-88-77
LINC LAB 135

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

October 1988

Slides presented at the Advanced School on Foundations of Logic Programming, 19 - 23 September 1988, Alghero, Sardinia, Italy

---

Acknowledgements: This research was supported in part by NSF grants CCR-87-05596, MCS-8219196-CER, IRI84-10413-AO2, DARPA grant NOOO14-85-K-0018, and U.S. Army grants DAA29-84-K-0061, DAA29-84-9-0027.

LOGIC PROGRAMMING BASED ON HIGHER-ORDER HEREDITARY HARROP FORMULAS

Dale Miller
Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389 USA

Slides given at the Advanced School on Foundations of Logic Programming, 19 – 23 September 1988, Alghero, Sardinia, Italy.

Logic programming based on higher-order hereditary Harrop formulas

Dale Miller
Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389
dale@cis.upenn.edu
(215)898-1593

Hereditary Harrop formulas are an extension to Horn clauses in which the body of clauses can contain implications and universal quantifiers. These formulas can further be extended by embedding them in a higher-order logic; that is, by permitting quantification over function symbol occurrences and some predicate symbol occurrences, and by replacing first-order terms with simply typed λ-terms. Our justification for considering this rich extension of Horn clause theory as a satisfactory logic programming language is provided by a proof-theoretic notion we call "uniform proofs". This notion will be defined and motivated. This extended language can provide very natural and direct implementations of various kinds of abstraction mechanisms. For example, higher-order hereditary Harrop formulas (hohh) can be used to support aspects of modular programming, abstract data types, and higher-order programming.

We have designed and built a logic programming system which implements hohh in much the same way Prolog implements first-order Horn clauses. This language and its interpreter, collectively called λProlog, will be described. We will present several example programs where λProlog provides a much more immediate and satisfactory implementation language than first-order Prologs. These examples are taken from theorem proving and program transformation. Finally, we will describe some aspects of our implementation of λProlog.

Lecture I
Introduction

Table of Contents

Colleagues

Duke University

Gopalan Nadathur	CS	assistant professor

Carnegie Mellon University

Conal Elliott	CS	graduate student
Frank Pfenning	CS	research scientist

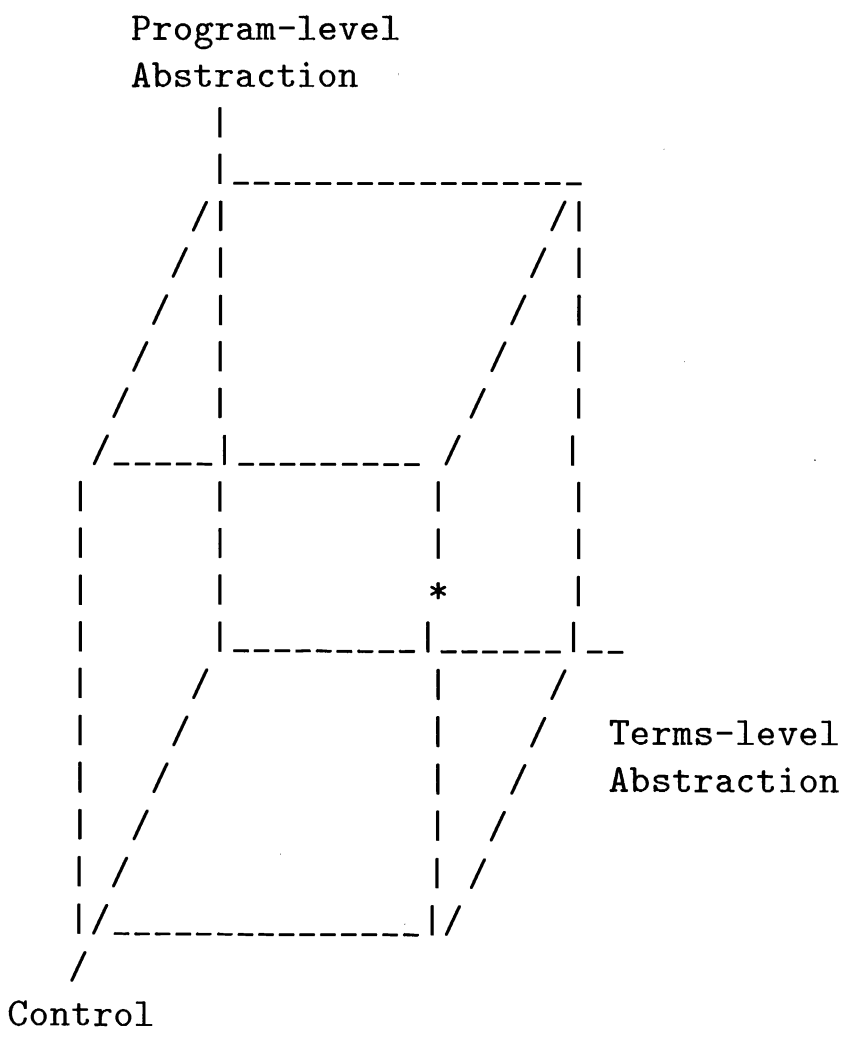
University of Edinburgh

James Harland	CS	graduate student

University of Pennsylvania

Amy Felty	CIS	graduate student
Elsa Gunter	CIS	post doc
John Hannan	CIS	graduate student
Dale Miller	CIS	assistant professor
Remo Pareschi	CIS	graduate student
Andre Scedrov	Math	associate professor

Three Dimensions for Extending Horn Clauses



First-order Horn clauses form the origin.

References

[1] P. Andrews, Resolution in Type Theory, Journal of Symbolic Logic, 36 (3), 414 – 432, 1971.

[2] P. Andrews, An Introduction to Mathematical Logic and Type Theory, Academic Press, 1986.

[3] A. Church, A Formulation of the Simple Theory of Types, Journal of Symbolic Logic 5(1940), 56 – 68.

[4] A. Felty and D. Miller, Specifying theorem provers in a higher-order logic programming language, 9th International Conference on Automated Deduction, Argonne, IL, 23 – 26 May 1988.

[5] J. Gallier and W. Snyder, Higher-Order Unification Revisited: Complete Sets of Transformations. Submitted to a special issue of the *Journal of Symbolic Computation* (1988).

[6] J. Hannan and D. Miller, Enriching a meta language with higher-order features, Workshop on Meta Programming in Logic Programming, Bristol, UK, June 1988.

[7] J. Hannan and D. Miller, Uses of higher-order unification for implementing program transformers, Fifth Symposium on Logic Programming, August 1988, Seattle, Washington.

[8] R. Harrop, Concerning Formulas of the types $A \rightarrow B \vee C$, $A \rightarrow (Ex)B(x)$ in Intuitionistic Formal Systems. Journal of Symbolic Logic, 25(1), 1960, 27 — 32.

[9] G. Huet, A Unification Algorithm for Typed λ-Calculus, Theoretical Computer Science 1, 1975, 27 – 57.

[10] G. Huet and B. Lang, Proving and Applying Program Transformations Expressed with Second-Order Patterns.

Outline

Lecture I

Introduction

Lecture II

Higher-Order Horn Clauses:
Definition, Examples, and Theory

Lecture III

Higher-Order Unification and
a Generalization of SLD-Resolution

Lecture IV

Hereditary Harrop Formulas and
Uniform Proofs

Lecture V

An Approach to Modules and Lexical Scoping

Lecture VI

Higher-Order Hereditary Harrop Formulas

Goals of These Lectures

To probe the essential logical character of various notions of abstractions in logic programming.

- higher-order functions
- abstract data types
- modules

To describe computational aspects of higher-order logic.

To present some relationships between proof theory and logic programming.

To propose an extension to the logic of Horn clauses that maintains many of its computational aspects.

To present a programming language, λProlog, built on this extension.

To use the proposed extensions to provide new programming language features.

Extensions to Logic Programming

Amalgamate Prolog with other languages.

Modify existing interpreters to add new functionality.

Extend the logical foundations of Prolog.

- Increase the role of negation
- Increase the role of equality
- Quantify over more syntactic categories
- Add more logical primitives to queries

Extensions to the Logic of Logic Programming

We shall consider two kinds of extensions in these talks.

Quantificational extension

- adding quantification over predicate and/or function symbols
- higher-order Horn clauses
- terms extended with λ-terms

Propositional extension

- adding additional connectives to goals and program clauses
- hereditary Harrop formulas
- intuitionistic provability models computations

Analysis of "Good" Extensions

Extensions must maintain a certain match between an operational interpretation and the logical interpretation of connectives within goals.

Programs, Goals $\Longleftrightarrow$ Logical Formulas

Solving a Goal $\Longleftrightarrow$ Logical Provability

Logical connectives are to have search-related meanings, for example, the properties listed below such hold.

- $\mathcal{P} \vdash G_1 \vee G_2$ if and only if $\mathcal{P} \vdash G_1$ or $\mathcal{P} \vdash G_2$.
- $\mathcal{P} \vdash \exists x\, G$ if and only if for some t, $\mathcal{P} \vdash G[t/x]$.

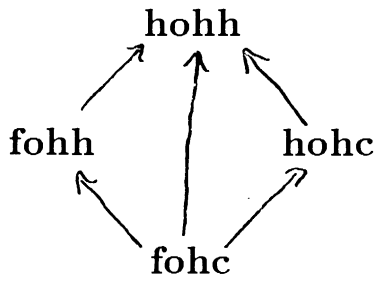
Four Abstract Logic Programming Languages

fohc First-order Horn clauses with classical or intuitionistic provability

hohc Higher-order Horn clauses with classical or intuitionistic provability

fohh First-order hereditary Harrop formulas with intuitionistic provability

hohh Higher-order hereditary Harrop formulas with intuitionistic provability



$\longrightarrow$ denotes containment

The Programming Language λProlog

λProlog is a programming language built on top of **hohh**. An interpreter for this language uses a depth-first discipline for both clauses and (pre)unifier selections.

λProlog extends Prolog by providing

- higher-order programming
- λ-terms as data structures
- stacked-based mechanism for introducing and discharging program clauses
- scoping mechanism for constants
- modules and local importing
- abstract data types

An overview of λProlog can be found in [23].

Implementations of λProlog

LP2.6 (August 1987, UPenn, Miller and Nadathur)

LP2.7 (July 1988, Duke and UPenn, Miller and Nadathur) Available in C-Prolog and Quintus Prolog version (4100 lines of code). Does not implement the full dynamic module facility anticipated by the theory. Does provide a depth-first implementation of full higher-order unification. Sources and several complete examples are in the distribution, which is available from

Gopalan Nadathur
Computer Science Department
Duke University
Durham, NC 27706 USA
(gopalan@cs.duke.edu)

eLP (expected Winter 88, CMU) Written in Common Lisp. Will be used as a meta language within the ERGO program development project. Should implement the full theory of **hohh** as well as certain enhancements. Implementation being done by Conal Elliott and Frank Pfenning.

Acta Informatica 11, (1978), 31 – 55.

[11] D. Miller, Proofs in Higher-Order Logic, Ph. D. dissertation, Carnegie Mellon University, Mathematics Department, August 1983.

[12] D. Miller, A logical analysis of modules in logic programming, to appear in the Journal of Logic Programming.

[13] D. Miller, A theory of modules for logic programming, IEEE Symposium on Logic Programming, Salt Lake City, September 1986.

[14] D. Miller, Lexical Scoping and Abstract Data Types in Logic Programming, in preparation.

[15] D. Miller, Unification Under A Mixed Prefix, unpublished draft, 50 pages.

[16] D. Miller and G. Nadathur, Higher-order logic programming, Proceedings of the Third International Logic Programming Conference, London, June 1986, 448 – 462.

[17] D. Miller and G. Nadathur, Some uses of higher-order logic in computational linguistics, 24th Annual Meeting of the Association for Computational Linguistics, New York, June 1986.

[18] D. Miller and G. Nadathur, A logic programming approach to manipulating formulas and programs, IEEE Symposium on Logic Programming, San Franciso, September 1987.

[19] D. Miller, G. Nadathur, and A. Scedrov, Hereditary Harrop formulas and uniform proofs systems, Second Annual Symposium on Logic in Computer Science, Cornell University, June 1987, 98 — 105. Theorem 3 is wrong. It is corrected in the [20].

[20] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov, Uniform proofs as a foundation for logic programming, submitted May 1988.

[21] G. Nadathur, A Higher-Order Logic as the Basis for Logic Programming, Ph. D. dissertation, University of Pennsylvania, May 1987.

[22] G. Nadathur and D. Miller, Higher-order Horn clauses, submitted April 1988.

[23] G. Nadathur and D. Miller, An overview of λProlog, Fifth Symposium on Logic Programming, August 1988, Seattle, Washington.

[24] F. Pfenning, Partial polymorphic type inference and higher-order unification, Proceedings of the ACM Lisp and Functional Programming Conference, 1988.

[25] F. Pfenning and C. Elliot, Higher-order abstract syntax, Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation, 1988.

[26] L. Paulson, Natural Deduction as Higher-Order Resolution, Journal of Logic Programming 3(3), 1986, 237 – 258.

Meta Mathematical Properties of Some Logics

First-order logic

- Valid formulas are precisely theorems (soundness and completeness).
- Theorem are described syntactically via axioms and inference rules.
- Valid formulas are described semantically via models.

Second-order "logic"

- Valid formulas are those provable in the standard model of the integers.
- Gödel showed that there is no (reasonable) syntactic characterization of these valid formulas.
- Second-order "logic" is more mathematics than logic.

Higher-order logic

- Syntactic tools are used to describe the nature of predicate and function quantification.
- Typed λ-calculus is generally used to denote terms of higher-type (see Church [3]).
- Theorems are described syntactically: some approaches have complete model theories, some do not.

An Example of Higher-Order Reasoning

(1) $\forall B\ (B \subseteq open \supset open(\bigcup B))$

open set axiom

(2) $\forall z\ (Az \supset \exists G\ (open(G) \wedge Gz \wedge G \subseteq A))$

assumption

(3) $\{G \mid G \subseteq A \wedge open(G)\} \subseteq open$

simple

(4) $open(\bigcup\{G \mid G \subseteq A \wedge open(G)\})$

Modus Ponens 1, 3

(5) $\bigcup\{G \mid G \subseteq A \wedge open(G)\} \subseteq A$

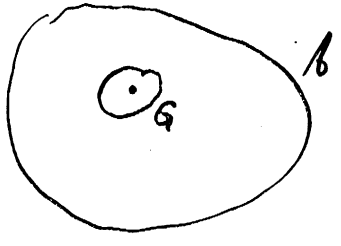
simple

(6) $A \subseteq \bigcup\{G \mid G \subseteq A \wedge open(G)\}$

simple (uses 2)

(7) $open(A)$

4, 5, 6, and extensionality



Another Example of Higher-Order Reasoning

believes(John, "The sun rises in the east.")

$\vdash$ "The sun rises in the east."$\equiv$ "Nixon lied."

Therefore, believes(John, "Nixon lied.")

The problem illustrated above is generally addressed by embellishing the underlying logic to provide an analysis of the *intensionality* of a proposition.

However, a weak enough logic can also block such conclusions.

Extensional / Non-Extensional / Intensional

Extensionality: Predicates (and function) are equal if they have the same extensions. Generally assumed in mathematics.

Direct higher-order extensions to first-order logic do not guarantee extensionality. Axioms such as

$$\forall x[Px \equiv Qx] \supset P = Q$$

must be added explicitly to get an extensional logic.

Intensional logics, such as those of Montague and Gallin, are embellishments of extensional higher-order logics with extra constants (e.g. intensional operators, modal operators) and with additional axioms and inference rules.

With regard to such "semantic" issues, we shall focus on a very weak higher-order extension to first-order logic.

Higher-Order Logic as an Object and Meta Language

Original examination of higher-order logic was to formalize mathematics and then to study the resulting formalism to conclude properties of mathematics. See Church [3] and Andrews [2].

Higher-order logic can makes a very interesting and powerful meta language. Some recent work has focused on the following three areas.

- theorem provers: Felty and Miller [4], Paulson [26], Pfenning [24]
- program transformers: Hannan and Miller [6] and [7], Miller and Nadathur [18], Pfenning and Elliott [25].
- natural language semantics: Miller and Nadathur [17].

Simply Typed λ-Terms

Types:

- A set of ground types $\{o, b_1, \ldots, b_n\}$
- All types $\alpha \to \beta$ where α and β are types.

Terms:

$$c \mid x \mid \lambda x.t \mid (t_1\ t_2)$$

Equivalence of Terms:

α	$\lambda x.(fx) \equiv_\alpha \lambda y.(fy)$
β	$(\lambda g.\lambda x.(g\ x))\ f \equiv_\beta \lambda x.(f\ x)$
η	$\lambda x.(fx) \equiv_\eta f$

Functions are expressions of functional type, that is, of type $\alpha \to \beta$.

Predicate are functional expressions of target type o, that is, of type $\alpha_0 \to \cdots \to \alpha_n \to o$.

Propositions are expressions of type o.

An Example of λ-Conversion

$$\lambda x(x+1)$$
$$\lambda f \lambda z(f(fz))$$

$$(\lambda f \lambda z(f\ (f\ z)))\ \lambda x(x+1)\ c$$
$$\lambda z((\lambda x(x+1))(\lambda x(x+1)z))\ c$$
$$\lambda z((\lambda x(x+1))(z+1))\ c$$
$$\lambda z((z+1)+1)\ c$$
$$((c+1)+1)$$

Adding Logic to λ-Terms

$\neg_{o \to o}$	negation
$\vee_{o \to (o \to o)}$	disjunction
$\wedge_{o \to (o \to o)}$	conjunction
$\supset_{o \to (o \to o)}$	implication
$\forall_{(\alpha \to o) \to o}$	universal α-set recognizer
$\exists_{(\alpha \to o) \to o}$	non-empty α-set recognizer

$\forall(\lambda x\ P)$ is abbreviated as $\forall x\ P$.

$\exists(\lambda x\ P)$ is abbreviated as $\exists x\ P$.

Type association is to the right: $o \to (o \to o)$ is written more simply as $o \to o \to o$.

Formulas as λ-Terms

"Every man loves a woman."

$$\forall x\ (man(x) \supset \exists y\ (woman(y) \wedge loves(x, y)))$$

$$\forall \lambda x((man\ x) \supset \exists \lambda y((woman\ y) \wedge (loves\ x\ y)))$$

"uncle whose children are doctors"

$$\lambda x((uncle\ x) \wedge (\forall \lambda y((child\ x\ y) \supset (doctor\ y))))$$

Another Example of λ-Conversion

$$\bigcup ::= \lambda B \lambda x \exists G\ (BG \wedge Gx)$$

$$\subseteq ::= \lambda P \lambda Q \forall x\ (Px \supset Qx)$$

$$\{G \mid G \subseteq A \wedge (open\ G)\} ::= \lambda G.\ G \subseteq A \wedge (open\ G)$$

$$\bigcup \{G \mid G \subseteq A \wedge (open\ G)\}$$

$$[\lambda B \lambda x \exists G\ (B\ G \wedge G\ x)][\lambda G.\ G \subseteq A \wedge (open\ G)]$$

$$\lambda x \exists G\ [\lambda G.\ G \subseteq A \wedge (open\ G)]G \wedge Gx$$

$$\lambda x \exists G\ [G \subseteq A \wedge (open\ G) \wedge Gx]$$

$$\lambda x \exists G\ [\forall x\ [Gx \supset Ax] \wedge (open\ G) \wedge Gx]$$

An Informal Description of $\mathcal{T}$

All constants and variables have simple types.

Quantification over predicates and function is permitted.

Axioms and inference rules for the classical (resp. intuitionistic) version of $\mathcal{T}$ are those of classical (resp. intuitionistic) first-order logic plus the inference rule of λ-conversion:

If A λ-converts to A' and $\vdash A$, then $\vdash A'$.

This is roughly equivalent to thinking of equality of terms as being modulo λ-conversion.

Axioms of extensionality, description, choice and infinity are not used in $\mathcal{T}$.

See Church 1940 [3] and Andrews 1986 [2] for more about this kind of higher-order logic.

Meta theoretic results:

Cut-Elimination	Schutte, Tait, Takahashi, Girard
Resolution	Andrews 1971 [1]
Unification	Huet 1975 [9]
Herbrand Theorem	Miller 1983 [11]

Lecture II
Higher-Order Horn Clauses: Definition, Examples, and Theory

Table of Contents

Which Formulas Should Be Considered Higher-Order Horn Clauses?

Most certainly, the following should be examples of higher-order Horn clauses.

$$mappred\ P\ nil\ nil$$
$$(P\ X\ Y) \wedge (mappred\ P\ L\ K) \supset (mappred\ P\ (cons\ X\ L)\ (cons\ Y\ K))$$

$$mapfun\ F\ nil\ nil$$
$$(mapfun\ F\ L\ K) \supset (mapfun\ F\ (cons\ X\ L)\ (cons\ (F\ X)\ K))$$

Here the types for the four non-logical constants would be something like the following:

$$\begin{aligned} nil &: list \\ cons &: i \to list \to list \\ mappred &: (i \to i \to o) \to list \to list \to o \\ mapfun &: (i \to i) \to list \to list \to o \end{aligned}$$

Can the Head of a Clause Be a Predicate Variable?

Consider the following two formulas:

$$\forall P\ \forall X\ ((q\ X) \supset (P\ X))$$
$$(q\ a)$$

From these two clauses, any formula is provable. To prove an arbitrary formula, say r, use the instance $P \mapsto \lambda x.r$ and $X \mapsto a$ to get

$$(q\ a) \supset r.$$

These clauses are, thus, inconsistent.

The predicate head of a Horn clause describes which procedure that clause is helping to define.

Higher-Order Horn Clauses

Let $\mathcal{H}^+$ be the set of all λ-normal formulas built from non-logical constants, variables, and the logical constants *true*, $\wedge$, $\vee$ and $\exists$.

Let G be a syntactic variable for propositions in $\mathcal{H}^+$.

Let A be a syntactic variable for propositions in $\mathcal{H}^+$ with non-logical constants as their head. Such formulas are called *atoms.*

A *higher-order Horn clause* is the universal closure of a formula of the form $G \supset A$ or simply A.

Let $\mathcal{P}$ be a syntactic variable for sets of Horn clauses.

A Possible Problem

Consider a proof of $\exists Y\ pY$ from the higher-order Horn clause

$$\forall Q\ (Q \supset pa)$$

There is a proof with answer substitution $Y \mapsto a$. The instance of this Horn clause used in this proof is

$$true \supset pa.$$

There is another proof, however, which yields no answer substitution. First, instantiate x with $\neg pb$ to get the formula

$$\neg pb \supset pa$$

which is equivalent (classically) to the disjunction

$$pb \vee pa.$$

The formula $\exists Y\ pY$ is then provable with the "disjunctive" answer substitution $Y \mapsto a$ or $Y \mapsto b$.

The higher-order substitution instance of a higher-order Horn clause is not necessarily a higher-order Horn clause.

An Approach to Solving This Problem

Notice that if $s, t \in \mathcal{H}^+$ then $[x := s]t \in \mathcal{H}^+$, that is, $\mathcal{H}^+$ is closed under substitutions from $\mathcal{H}^+$.

Thus, higher-order Horn clauses are closed under instantiations from $\mathcal{H}^+$.

Approach: If G is provable from a set of higher-order Horn clauses then it is provable by a proof whose only substitution terms are taken from $\mathcal{H}^+$.

Thus, $\mathcal{H}^+$ is the *Herbrand Universe* for higher-order Horn clauses.

Positive Instances

Let $\mathcal{P}$ be a set of higher-order Horn clauses.

Let $[\mathcal{P}]$ be the smallest set of higher-order Horn clauses such that

- $\mathcal{P} \subseteq [\mathcal{P}]$, and
- if $\forall x\ D \in [\mathcal{P}]$ and $t \in \mathcal{H}^+$ is closed and the same type as x, then $[x := t]D \in [\mathcal{P}]$.

Provability from Horn Clauses

Theorem: Let $G_1, G_2, A, \exists x\ Bx \in \mathcal{H}^+$ each be closed propositions. Let $\mathcal{P}$ be a set of higher-order Horn clauses. Let $\vdash_{\mathcal{T}}$ be classical provability over $\mathcal{T}$. The following are true:

- $\mathcal{P} \vdash_{\mathcal{T}} true$.
- $\mathcal{P} \vdash_{\mathcal{T}} G_1 \wedge G_2$ if and only if $\mathcal{P} \vdash_{\mathcal{T}} G_1$ and $\mathcal{P} \vdash_{\mathcal{T}} G_2$.
- $\mathcal{P} \vdash_{\mathcal{T}} G_1 \vee G_2$ if and only if $\mathcal{P} \vdash_{\mathcal{T}} G_1$ or $\mathcal{P} \vdash_{\mathcal{T}} G_2$.
- $\mathcal{P} \vdash_{\mathcal{T}} \exists x\ B$ if and only if there is a closed formula $t \in \mathcal{H}^+$ such that $\mathcal{P} \vdash_{\mathcal{T}} [x := t]B$.
- $\mathcal{P} \vdash_{\mathcal{T}} A$ if and only if $A \in [\mathcal{P}]$ or there is a $G \supset A \in [\mathcal{P}]$ and $\mathcal{P} \vdash_{\mathcal{T}} G$.

Proof: See Nadathur's dissertation [21] or the joint paper [22]. See also [16].

Some λProlog Syntax

The syntax of terms is similar to that for Lisp (functions are represented as curried expression). Major differences are:

- λ-abstraction is written with an infix \.
- Lists are written as in Prolog.

```
(reduce (lambda (x y) (x+y)) '(1 2 3) 0)=6
      (reduce X\Y\ (X + Y) [1,2,3] 0 6)
```

The syntax of clauses and goals is similar to that for Prolog. The major difference is the possibility of having explicit existential quantification in goals.

```
?- sigma Y\(generate X Y, test Y Z).
    ∃
```

The syntax of type declarations is similar to that for ML.

```
type  nil     list.
type  cons    i -> list -> list.
type  mappred (i -> i -> o) ->
               list -> list -> o.
type  mapfun  (i -> i) -> list -> list -> o.
```

Polymorphic Typing

Types are a language of first-order terms that is separate from the language of λ-terms.

Primitive types:

```
o, int, string, ...
```

Type constructors:

```
(list int), (pair int string),
      (list (pair int int)), ...
```

Functional types:

```
int -> int,  int -> (list int) -> o, ...
```

Polymorphic types: Allow first-order variables in type expressions.

```
type [_|_]  A -> (list A) -> (list A)
type []     (list A)
type pair   A -> B -> (pair A B)
```


The Mappred Program

```
type mappred (A -> B -> o) ->
                  (list A) -> (list B) -> o.

mappred P nil nil.
mappred P [X|L1] [Y|L2] :- P X Y,
                           mappred P L1 L2.
```

The predicate variable P appears both as an argument and as taking arguments. Consider the following simple clauses:

```
type age  person -> int -> o.

age bob 23.
age sue 24.
age ned 23.
```

and now consider the following query:

```
?- mappred X\Y\(age X Y) [ned, bob, sue] L.
```

This query essentially asks for the ages of the individuals ned, bob and sue. An answer substitution for L is [23, 23, 24].

The Sublist Program

```
type sublist (A -> o) ->
             (list A) -> (list A) -> o.

sublist P [X|L] [X|K] :- P X, sublist P L K.
sublist P [X|L] K :- sublist P L K.
sublist P [] [].

type have_age  (list person) ->
               (list person) -> o.

have_age L K :-
    sublist Z\(sigma X\(age Z X)) L K.

type same_age  (list person) ->
               (list person) -> o.

same_age L K :- sublist Z\(age Z A) L K.
```


Flexible Goals

```
P bob 23.
```

One answer to this query is the substitution `(X\Y\(age X Y))` for `P`. Many other substitutions are also valid. Let `G` be any provable closed query. The substitution `X\Y\G` for `P` is a legal answer substitution.

For example, substituting

```
X\Y\ (memb 4 [3,4,5])
```

for `P` is also an answer substitution.

Constraining Flexible Goals

Such queries are essentially ill-posed. The range of a predicate quantifier should be restricted by the programmer. For example,

```
type primrel  (person -> o) -> o.
type rel      (person -> o) -> o.
type mother   person -> o.
type wife     person -> o.

primrel mother.
primrel wife.
rel R :- primrel R.
rel X\Y\ (sigma Z\(R X Z , S Z Y)) :-
        primrel R , primrel S.
mother jane mary.
wife john jane.
```

The query

```
?- rel R, R john mary,
```

has the unique answer substitution for `R`

```
X\Y\(sigma Z\(wife X Z, mother Z Y))
```


Interpretations for Higher-Order Horn Clauses

An *interpretation* is any set of closed, atomic propositions in $\mathcal{H}^+$.

The following compositional definition of satisfaction is problematic.

- $I \models true$
- $I \models G$ if G is atomic and $G \in I$.
- $I \models G_1 \vee G_2$ if $I \models G_1$ or $I \models G_2$.
- $I \models G_1 \wedge G_2$ if $I \models G_1$ and $I \models G_2$.
- $I \models \exists x\, B$ if there is a closed term $t \in \mathcal{H}^+$ such that $I \models [x := t]B$.

The problem with this definition is that the recursion in the last line is not well-founded: the formula $[x := t]B$ can have more logical connectives that the formula $\exists x\, B$.

$$\exists P\,(Pa)$$
$$P \mapsto \lambda z(\exists P\,(Pa) \wedge q)$$
$$\exists P\,(Pa) \wedge q$$

A Non-Compositional Notion of Satisfaction

Let I be an interpretation and G a proposition in $\mathcal{H}^+$. Write $I \not\models G$ if there is a sequence of formulas

$$G_1, \ldots, G_n = G$$

such that for $i = 1, \ldots, n$, either

- G_i is true, or
- $G_i \in I$, or
- $G_i = G' \wedge G''$ and $\{G', G''\} \subseteq \{G_1, \ldots, G_{i-1}\}$, or
- $G_i = G' \vee G''$ and G' or $G'' \in \{G_1, \ldots, G_{i-1}\}$, or
- $G_i = \exists x\ G'$ and there is a $t \in \mathcal{H}^+$ such that $[x := t]G' \in \{G_1, \ldots, G_{i-1}\}$.

A Least Fixpoint Interpretation

Let $\mathcal{P}$ be a given set of higher-order Horn clauses. Define the following function from interpretations to interpretations:

$$T_{\mathcal{P}}(I) := \{A \mid A \in [\mathcal{P}] \text{ or } G \supset A \in [\mathcal{P}] \text{ and } I \not\models G\}.$$

It is not difficult to see that $T_{\mathcal{P}}$ is monotone and continuous on the set of all interpretations.

The least fixpoint of $T_{\mathcal{P}}$ is therefore

$$T^{\infty}_{\mathcal{P}}(\emptyset) := \bigcup_{n=0}^{\infty} T^n_{\mathcal{P}}(\emptyset).$$

It is this subset of $\mathcal{H}^+$ that we think of as being determined by $\mathcal{P}$, and we call it the *denotation* of $\mathcal{P}$.

Theorem: Let $G \in \mathcal{H}^+$ be a closed proposition. Then $T^{\infty}_{\mathcal{P}}(\emptyset) \not\models G$ if and only if $\mathcal{P} \vdash_{\mathcal{T}} G$. See [21].

The Mapfun Program

Consider the following program

```
type mapfun  (A -> B) ->
             (list A) -> (list B) -> o.

mapfun F [X|L1] [(F X)| L2] :-
         mapfun F L1 L2.
mapfun F [] [].
```

and consider the following query

```
?- mapfun X\(g a X) [a, b] L,
```

The answer substitution for L is

```
[(g a a), (g a b)]
```

An interpreter would need to form the terms `((X\(g a X)) a)` and `((X\(g a X)) b)` and then reduce these terms using the rules of λ-conversion.

The Mapfun Program in "Reverse"

Consider the following query:

```
?- mapfun F [a, b] [(g a a), (g a b)].
```

There is precisely one answer for this query, namely the substitution `X\(g a X)` for `F`. The unification problem `(F a)` and `(g a a)` needs to be solved here. There are four unifiers for `F`:

```
X\(g X X),  X\(g a X),
X\(g X a),  X\(g a a).
```

If any but the second is selected first, the choice of unifier would need to be backtracked over.

Notice that the following qoal is not provable:

```
mapfun F [a, b]  [c, d].
```

There is no "function" (that is, λ-term) which maps `a` to `c` and maps `b` to `d`.

λ-terms as Data Structures

λ-terms capture the *higher-order abstract syntax* of objects like formulas and programs [25].

$$\forall x\ (p(x) \vee q(x))$$

```
(all X\ ((p X) or (q X)))
```

$$sum\ m\ n \quad == \quad if\ (m = 0)\ then\ n \\ else\ sum\ (m-1)\ (n+1)$$

```
(fixpt Sum\M\N\
       (cond (M = 0) N
             (Sum (M - 1) (N + 1))))
```


The Advantage of Such a Representation

The equivalence of the the two formulas

$$\forall x\ (p(x) \vee q(x)) \text{ and } \forall y\ (p(y) \vee q(y))$$

is captured by the α-convertibility of

```
(all X\ ((p X) or (q X)))
(all Y\ ((p Y) or (q Y)))
```

Substitution is implemented by β-reduction. For example, the result of instantiating $\forall y\ (p(y) \vee q(y))$ with $f(a)$ is the represented by the λ-normal form of

```
(X\ ((p X) or (q X))) (f a)
```

Higher-order unification implements sophisticated pattern matching. Consider unifying an expression against the following two higher-order templates:

```
(all X\ ((P X) or (Q X)))
  (all X\ (P or (Q X)))
```

Programs as Data Objects

Programs have a rich structure:

- variable bindings (for formal parameters)
- function bindings (for defining new functions)

Consider using Lisp as the meta-language:

- Use Lisp's notation for λ-terms to represent programs.
- The only primitive mechanisms for manipulating such terms are CAR, CDR, CONS.
- Lisp implementations produce obscure descriptions of program analysis.

Need more sophisticated analysis techniques

Programs as λ-terms

Consider a simple functional language with a conditional operator, lists, and recursion. The append program might appear as

```
fun append K L =
    (if (null K) L
        (cons (car K) (append (cdr K) L)))
```

By introducing new constants to denote each programming language construct, we can represent this program by the the term

```
fix F\K\L\ (if (null K) L
                (cons (car K) (F (cdr K) L)))
```

- Bindings in the object language are represented by bound variables (abstractions) in the meta language
- Two object level programs differing only in renaming of bound variables are treated as equivalent terms.
- Substitution for formal parameters in the object language is acheived by β-reduction.

Analyzing the Append Program

Consider uinifying the code for append

```
fix F\K\L\ (if (null K) L
               (cons (car K) (F (cdr K) L)))
```

against the template

```
fix F\M\N\ (if (C M) (G M N)
                     (H (F (K M) N) M))
```

with free variables C, G, H, and K. It unifies with the "append" term with the substitution

```
C -->  X\ (null X)
G -->  X\Y\ Y
H -->  X\Y\ (cons (car Y) X)
K -->  X\ (cdr X)
```

Unification such as this provides a new method of analyzing program structure. It is very different from representing programs as lists and manipulating them using CAR or CDR in Lisp or first-order unification and =.. in Prolog.

Lecture III
Higher-Order Unification and a Generalization of SLD-Resolution

Table of Contents

Higher-Order Unification

Given any two (simply typed) terms s and t of the same type, the task of finding a substitution σ, if one exists, such that $\sigma(s) = \sigma(t)$, is known as *higher-order unification.*

[A better name is simply typed λ-term unification modulo $\alpha\beta\eta$-conversion.]

Some characteristics of higher-order unification:

- It is a semi-decidable problem (even for just second-order unification).
- If unifiers exists, there is not necessarily a most general unifier. In fact, there may be infinitely many independent unifiers.
- General non-redunant search can only be achieved for *pre-unifiers* and not unifiers.
- Some unification problems, called *flexible-flexible* problems, can produce so many unifiers that solving them is best delayed. Flexible-flexible problems are treated as constraints.

Some References on Higher-Order Unification

Huet in [9] gave the first full description of higher-order unification.

Gallier and Snyder in [5] redo Huet's approach using the *sets of transformations* of Herbrand-Martelli-Montanari.

Miller in [15] considers higher-order unification in the presence of a mixed prefix, *i.e.* admitting quantifier alternations.

Elliott's Ph. D. thesis at Carnegie Mellon University will be on extensions to higher-order unification.

Some Structural Properites of Higher-Order Unification

Dependence on an abstraction. A term t is *dependent on its i^{th} abstraction* if a λ-normal form of t is of the form

$$\lambda x_1 \ldots \lambda x_i \; t'$$

and x_i is free in t'. t' may be a of functional type itself.

The term

$$t_0 \; = \; \lambda u \lambda v \lambda w \lambda h \; (F \; u \; h(G \; v))$$

is dependent on its first, second and fourth abstractions but not its third.

Some Additional Properties

Dependency Invariance. Let t be a term that is dependent on its i^{th} abstraction. If t λ-converts to s, then s is dependent on its i^{th} abstraction.

That is, dependence on an abstraction is well-defined with respect to term equality.

Dependency and Substitution. Let t be a term and σ a substitution. If $\sigma(t)$ is dependent on its i^{th} abstraction, then t is dependent on its i^{th} abstraction.

That is, abstraction dependencies cannot be introduced by substitution.

For example, let

$$t = \lambda x \lambda y.(F \; x)$$
$$\sigma = [F \mapsto (c \; y)]$$

Then $\sigma(t) = \lambda x \lambda z.(c \; y \; x)$.

Nesting of Abstractions

Nested Dependency. Let t be a λ-normal term, let σ be a substitution, such that

$$\begin{aligned} t &= \lambda x_1 \ldots \lambda x_n.t' \\ \sigma(t) &= \lambda x_1 \ldots \lambda x_n.t'' \end{aligned}$$

Let $x_i, x_j \in \{x_1, \ldots, x_n\}$.

If every occurrence of x_i in t' is in the scope of an occurrence of x_j in t'

then every occurrence of x_i in t'' is in the scope of an occurrence of x_j in t''.

That is, the "in the scope of" relationship between bound variables does not change under substitution.

Consider again the term

$$t_0 = \lambda u \lambda v \lambda w \lambda h.(F\ u\ (h(G\ v)))$$

For any substitution σ (for F and G), every occurrence of v in the term $\sigma(t_0)$ will be in the scope of h.

An Example

Consider the term (higher-order template)

$$t_0 = \lambda u \lambda v \lambda w \lambda h.(F\ u\ (h\ (G\ v)))$$

which we will try to unify with each of the terms

$$\begin{aligned} t_1 &= \lambda u \lambda v \lambda w \lambda h.((2 * w) + h(3 * v)) \\ t_2 &= \lambda u \lambda v \lambda w \lambda h.((2 * u) + (3 * v)) \\ t_3 &= \lambda u \lambda v \lambda w \lambda h.((2 * u) + h(3 * v)) \end{aligned}$$

- For any substitution σ, $\sigma(t_1)$ is dependent upon its third abstraction (w) while t_0 is not dependent of its third abstraction. Hence, t_1 does not unify with t_0.
- Since all occurrences of v in t_0 are restricted to be in the scope of h and since v is not so restricted in t_2, t_0 does not unify with t_2.
- t_3 does unify with t_0, with substitution $\sigma =$

$$[F \mapsto \lambda x \lambda y.((2 * x) + y), \quad G \mapsto \lambda x.(3 * x)]$$

See [7] for more of this kind of analysis.

A Simple Tail Recursion Schema

Consider the following schema (open higher-order term):

$$(fix\ \lambda f \lambda x \lambda y\ (if\ (C\ x\ y)\ (B\ x\ y)\ (f\ (E_1\ x\ y)\ (E_2\ x\ y))))$$

From our properties, we have the following constraints on closed instances of this term:

- They are terms denoting recursive program of two arguments and the body of the program must be an *if* expression.
- No recursive calls (f) are possible in the "conditional" and "then" parts of the program.
- There is exactly one recursive call in the "else" part of the program and it occurs at the top-level.

Matching the Tail Recursive Schema

$$(fix\ \lambda f \lambda x \lambda y\ (if\ (C\ x\ y)\ (B\ x\ y)\ (f\ (E_1\ x\ y)\ (E_2\ x\ y))))$$

The following term representing the append program does not unify with this schema:

$$(fix\ \lambda f \lambda k \lambda l\ (if\ (null\ k)\ l\ (cons\ (car\ k)\ (f\ (cdr\ k)\ l))))$$

The following term representing the reverse program,

$$(fix\ \lambda f \lambda k \lambda l\ (if\ (null\ k)\ l\ (f\ (cdr\ k)\ (cons\ (car\ k)\ l))))$$

does unify with this schema with substitution

$$\begin{aligned} C &\mapsto \lambda x \lambda y.(null\ x) \\ B &\mapsto \lambda x \lambda y.y \\ E_1 &\mapsto \lambda x \lambda y.(cdr\ x) \\ E_2 &\mapsto \lambda x \lambda y.(cons\ (car\ x)\ y) \end{aligned}$$

A More General Tail Recursion "Template"

```
type tail_rec_body  ((A1 -> A2 -> A3) ->
                     A1 -> A2 -> A3) -> o.
type tailrec        (A1 -> A2 -> A3) -> o.

tailrec (fix Prog):- tail_rec_body Prog.

tail_rec_body (F\X\Y\ (H X Y)).
tail_rec_body (F\X\Y\ (F (G X Y) (H X Y))).
tail_rec_body
  (F\X\Y\ (if (C X Y)
              (H1 F X Y) (H2 F X Y))) :-
        tail_rec_body H1, tail_rec_body H2.
```

For more analysis and an extension of this λProlog program see [18]. See also Huet and Lang [10].

The Structure of λ-normal Terms

All λ-normal terms can be put into the form

$$t = \lambda x_1 \ldots \lambda x_n \ (h \ e_1 \cdots e_m)$$

where $n, m \geq 0$ and $(h \ e_1 \cdots e_m)$ is of primitive type.

The list $x_1, \ldots, x_n$ is called the *binder.*

The variable or constant h is called the *head.*

The terms $e_1, \ldots, e_m$ are the *arguments.*

If h is a constant or a member of the binder, the term is *rigid.*

Otherwise, h is a variable not a member of its binder and the term is *flexible.*

Disagreement Pairs of λ-terms

A disagreement pair is a pair of two λ-normal terms of the same type. Given α- and η-conversions, two such terms can be rewritten into equivalent terms with the same binder. Thus we write disagreement pairs as

$$\lambda x_1 \cdots \lambda x_n \ \langle h\ e_1 \ldots e_{m_1},\ k\ f_1 \ldots f_{m_2} \rangle$$

Disagreement pairs fall into three classes:

rigid-rigid	both terms are rigid
flexible-rigid	one term is flexible and one rigid. We assume the first one listed is flexible, otherwise swap them.
flexible-flexible	both terms are flexible

Simplifying Rigid-Rigid Pairs

Consider the rigid-rigid disagreement pair

$$\lambda x_1 \cdots \lambda x_n \ \langle h\ e_1 \ldots e_{m_1}, k\ f_1 \ldots f_{m_2} \rangle.$$

This pair is not unifiable if h is not identical to k.

Thus for this pair to be unifiable then $h = k$ and $m_1 = m_2 = m$ and the list of disagrement pairs

$$\lambda x_1 \cdots \lambda x_n \ \langle e_1, f_1 \rangle, \quad \ldots, \quad \lambda x_1 \cdots \lambda x_n \ \langle e_m, f_m \rangle$$

are all simultaneously unifiable.

If the types of h and k are different, then they must be unifiable. Use the mgu of the type expressions.

A list of disgreement pairs can either be recognized as non-unifiable or can be simplified to an equivalent unification problem with only flexible-rigid or flexible-flexible pairs.

Processing Flexible-Rigid Pairs

Given the flexible-rigid disagreement pair

$$\lambda x_1 \cdots \lambda x_n \ \langle h \ e_1 \ldots e_{m_1}, \ k \ f_1 \ldots f_{m_2} \rangle.$$

There are two possible and incomparable ways to get $(h \ e_1 \ldots e_{m_1})$ to have rigid head k after substitution *and* normalization.

Imitate — The flexible term gets k as its head directly. This can work only if k is not in the binder.

$$h \mapsto \lambda w_1 \ldots \lambda w_{m_1} \ (k \ (h_1 w_1 \ldots w_{m_1}) \ldots (h_{m_2} w_1 \ldots w_{m_1}))$$

Project — The flexible term gets k as its head indirectly by projecting one of the arguments $f_1, \ldots, f_{m_2}$ into the head position.

$$h \mapsto \lambda w_1 \ldots \lambda w_{m_1} \ (w_i \ (h_1 w_1 \ldots w_{m_1}) \ldots (h_p w_1 \ldots w_{m_1}))$$

where $1 \leq i \leq m_1$ and $p \geq 0$ is determined by the type of w_i.

Example 1: Occurs Check

Consider the unification problem

$$X = (F \ X)$$

where both X and F are variables. Notice that X occurs free in $(F \ X)$. Does this unification problem have a solution?

Yes. In fact two general ones, namely

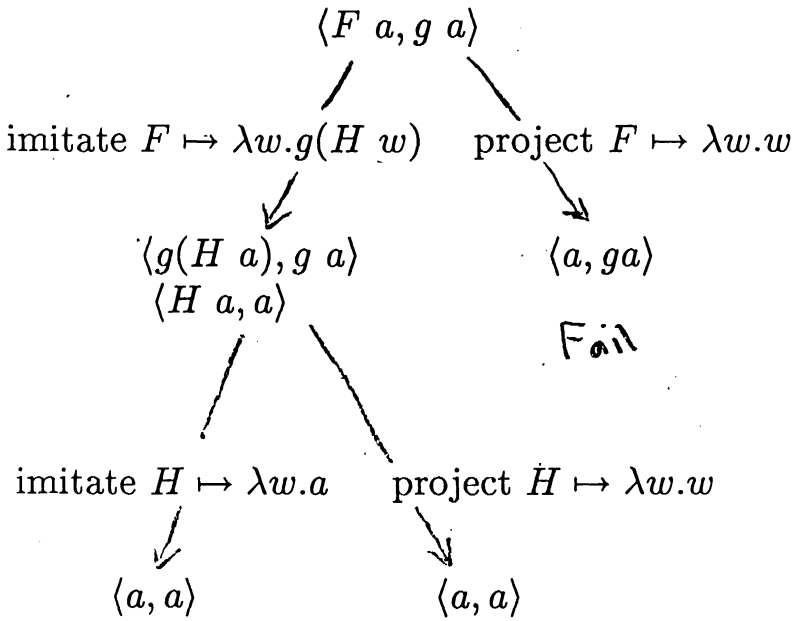
$$F \mapsto \lambda w.w, \quad \text{and} \quad F \mapsto \lambda w.X.$$

There are generalizations of the first-order occurs check that can be used in the higher-order setting to recognize failing unification problems.

Of course, the unification problem $X = t$ where t is a term not containing X free has the single unifier $X \mapsto t$.

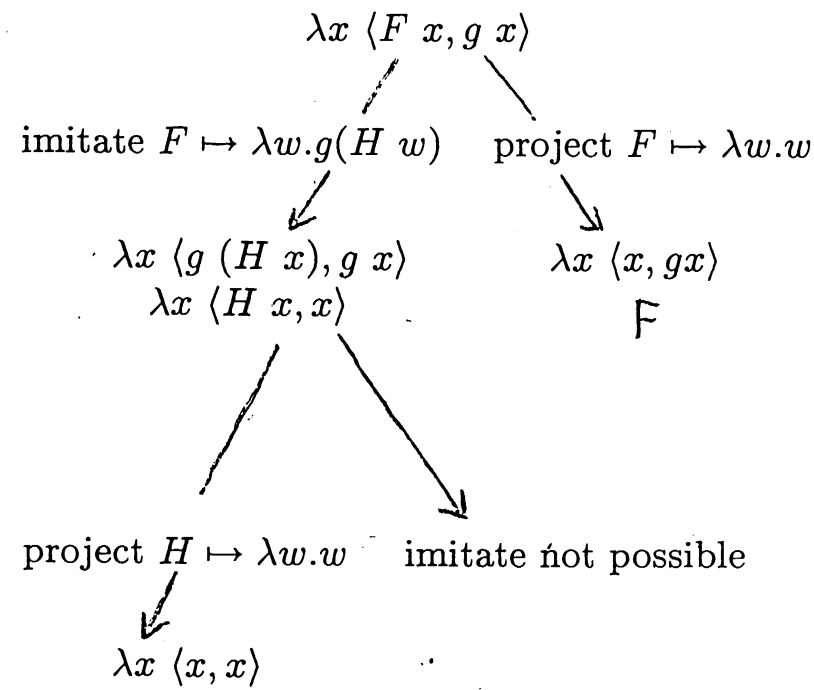
Example 2

Let F be a variable of type $i \to i$, g a constant of type $i \to i$, and a a constant of type i.



Answer substitutions: $F \mapsto \lambda w.ga$ and $F \mapsto \lambda w.gw$

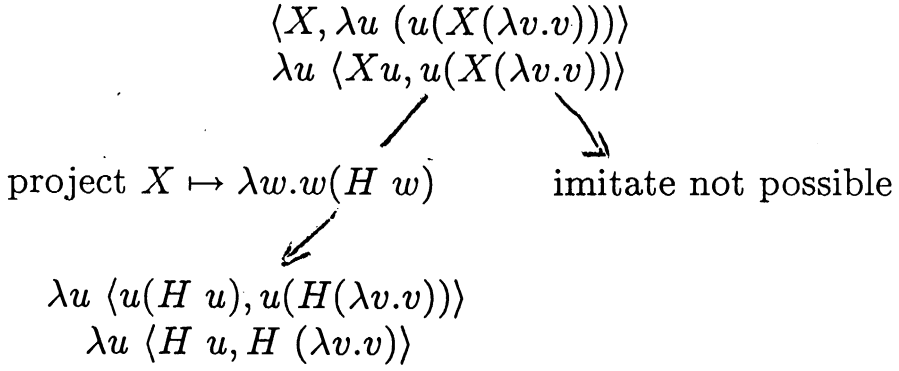
Example 3



Answer substitution: $F \mapsto \lambda w.gw$

Example 4

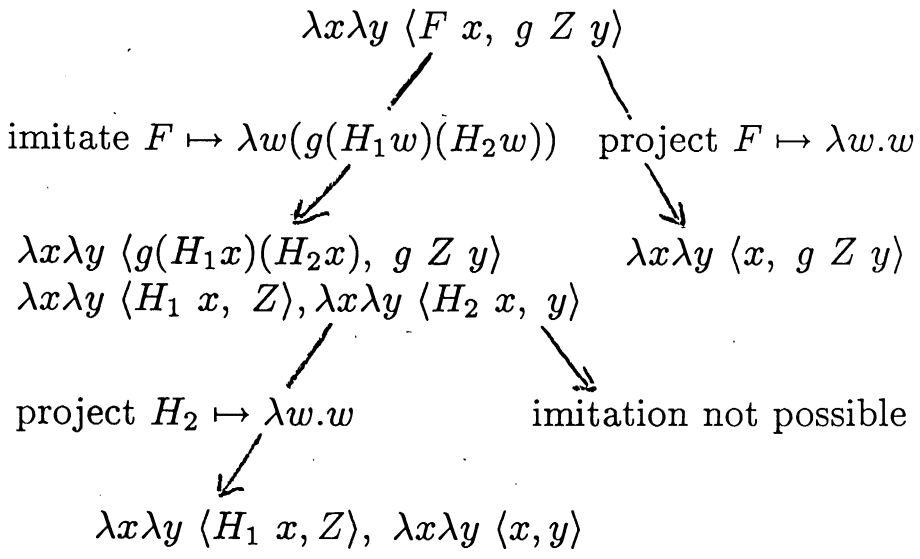
Let X be a variable of type $(i \to i) \to i$.



This final disagreement pair is flexible-flexible. This has the solution $H \mapsto \lambda w.Y$ which yields $X \mapsto \lambda w.w\ Y$ as an answer substitution.

Example 5

Let g be a constant of type $i \to i \to i$, let F be a variable of type $i \to i$, and let Z be a variable of type i.



The rigid-rigid disagrement pair above is not unifiable.

Generalizing SLD-Resolution

The state of a resolution-style theorem prover is the following:

- a program $\mathcal{P}$ which is a finite set of higher-order Horn clauses.
- a list of 4-tuples $\langle \mathcal{G}, \mathcal{U}, \theta, \mathcal{V} \rangle$ where
  - $\mathcal{G}$ is a list of goals that need to be proved,
  - $\mathcal{U}$ is a list of disagreement pairs that need to be unified,
  - θ is a substitution, and
  - $\mathcal{V}$ is a list of free variables including all those free in $\mathcal{G}$, $\mathcal{U}$, and θ.

$\mathcal{P}$-Derivation

$\langle \mathcal{G}_2, \mathcal{U}_2, \theta_2, \mathcal{V}_2 \rangle$ is *$\mathcal{P}$-derived* from $\langle \mathcal{G}_1, \mathcal{U}_1, \theta_1, \mathcal{V}_1 \rangle$ if $\mathcal{U}_1$ is simplified and not a failed unification problem and:

Goal reduction step: $\theta_2 = \emptyset$, $\mathcal{U}_2 = \mathcal{U}_1$, and there is a goal formula $G \in \mathcal{G}_1$ ($\mathcal{G}' := \mathcal{G}_1 - \{G\}$) s.t.

- G is $G_1 \wedge G_2$ and $\mathcal{G}_2 = \mathcal{G}' \cup \{G_1, G_2\}$ and $\mathcal{V}_2 = \mathcal{V}_1$, or
- G is $G_1 \vee G_2$ and, for $i = 1$ or $i = 2$, $\mathcal{G}_2 = \mathcal{G}' \cup \{G_i\}$ and $\mathcal{V}_2 = \mathcal{V}_1$, or
- G is $\exists x \ P$ and for some variable $y \notin \mathcal{V}_1$, $\mathcal{V}_2 = \mathcal{V}_1 \cup \{y\}$ and $\mathcal{G}_2 = \mathcal{G}' \cup \{[x := y]P\}$.

$\mathcal{P}$-Derivation (continued)

Backchaining step: Let $G \in \mathcal{G}_1$ be a rigid atom, and let $D \in \mathcal{P}$ be such that $D \equiv \forall x_1 \ldots \forall x_n \ (G' \supset A)$ for some sequence of new variables $x_1, \ldots, x_n$. Then $\theta_2 = \emptyset$, $\mathcal{V}_2 = \mathcal{V}_1 \cup \{x_1, \ldots, x_n\}$, $\mathcal{G}_2 = \mathcal{G}_1 - \{G\} \cup \{G'\}$, and let $\mathcal{U}_2$ be the simplified form of $\mathcal{U}_1 \cup \{\langle G, A \rangle\}$.

Unification step: $\mathcal{U}_1$ is not a solved set and for some flexible-rigid pair $\langle F_1, F_2 \rangle \in \mathcal{U}_1$ there is an imitation or projection substitution term, call it θ_2, and $\mathcal{G}_2 = \theta_2(\mathcal{G}_1)$, $\mathcal{U}_2$ is the simplified form of $\theta_2(\mathcal{U}_1)$, and $\mathcal{V}_2$ is updated by the new variables in θ_2.

See Nadathur's dissertation [21] or the joint paper [22].

Fixing Choices in $\mathcal{P}$-Derivations

Impose a depth-first discipline on the following choices.

- Goals processed in left-to-right order.
- Left disjuncts attempted before right disjunct.
- Clauses tried in top-down fashion.
- Reduce unification problems to flexible-flexible prior to solving goals.
- Do imitations prior to projections. [This is a switchable option in LP2.7.]
- Postpone flexible goals as well as flexible-flexible disagreement pairs. [Flexible goals not postponed in LP2.7.]

Lecture IV
Hereditary Harrop Formulas and Uniform Proofs

Table of Contents

Characterizing Proofs from Horn Clauses

Every goal is attempted with respect to the same

- program clauses, and
- constants.

That is, there are no scoping mechanisms available for either program clauses or constants.

Such scoping mechanism would, however, provide natural mechanisms for modular programming and abstract datatypes.

There are natural interpretations of implications and universal quantification in goals that can provide these scoping mechanisms.

In particular, implicational goals can be used to assume and discharge program clauses and universal goals can be used to assume and discharge constants.

Search Semantics for the Connectives

Let $\mathcal{P} \vdash_O G$ mean G *succeeds* given $\mathcal{P}$.

The *intended* success/failure semantics for each connective may then be given by the following:

AND $\mathcal{P} \vdash_O G_1 \wedge G_2$ only if $\mathcal{P} \vdash_O G_1$ and $\mathcal{P} \vdash_O G_2$

OR $\mathcal{P} \vdash_O G_1 \vee G_2$ only if $\mathcal{P} \vdash_O G_1$ or $\mathcal{P} \vdash_O G_2$

INSTANCE $\mathcal{P} \vdash_O \exists x\ G$ only if $\mathcal{P} \vdash_O G[t/x]$ for some term t

AUGMENT $\mathcal{P} \vdash_O D \supset G$ only if $\mathcal{P} \cup \{D\} \vdash_O G$

GENERIC $\mathcal{P} \vdash_O \forall x\ G$ only if $\mathcal{P} \vdash_O G[c/x]$ for some constant c that does not appear in $\mathcal{P}$ or in G.

Cut-Free Sequential Proofs

$$\frac{\Gamma \longrightarrow \Delta, B \qquad \Gamma \longrightarrow \Delta, C}{\Gamma \longrightarrow \Delta, B \wedge C}\ \wedge\text{-R} \qquad \frac{B, C, \Delta \longrightarrow \Theta}{B \wedge C, \Delta \longrightarrow \Theta}\ \wedge\text{-L}$$

$$\frac{B, \Delta \longrightarrow \Theta \qquad C, \Delta \longrightarrow \Theta}{B \vee C, \Delta \longrightarrow \Theta}\ \vee\text{-L}$$

$$\frac{\Gamma \longrightarrow \Delta, B}{\Gamma \longrightarrow \Delta, B \vee C}\ \vee\text{-R} \qquad \frac{\Gamma \longrightarrow \Delta, C}{\Gamma \longrightarrow \Delta, B \vee C}\ \vee\text{-R}$$

$$\frac{\Gamma \longrightarrow \Theta, B \qquad C, \Gamma \longrightarrow \Delta}{B \supset C, \Gamma \longrightarrow \Delta \cup \Theta}\ \supset\text{-L} \qquad \frac{B, \Gamma \longrightarrow \Theta, C}{\Gamma \longrightarrow \Theta, B \supset C}\ \supset\text{-R}$$

$$\frac{\Gamma, [x/t]P \longrightarrow \Theta}{\Gamma, \forall x\ P \longrightarrow \Theta}\ \forall\text{-L} \qquad \frac{\Gamma \longrightarrow \Theta, [x/t]P}{\Gamma \longrightarrow \Theta, \exists x\ P}\ \exists\text{-R}$$

$$\frac{\Gamma, [x/y]P \longrightarrow \Theta}{\Gamma, \exists x\ P \longrightarrow \Theta}\ \exists\text{-L} \qquad \frac{\Gamma \longrightarrow \Theta, [x/y]P}{\Gamma \longrightarrow \Theta, \forall x\ P}\ \forall\text{-R}$$

$$\frac{\Gamma \longrightarrow \Theta, \bot}{\Gamma \longrightarrow \Theta, B}\ \bot\text{-R}$$

$\Gamma \longrightarrow \Delta$ is *initial* if $\Gamma \cap \Delta$ contains an atomic formula. Standard proviso on $\forall$-R and $\exists$-L.

Uniform Sequential Proofs

Definition: A uniform proof is a cut-free, atomically closed sequent proof in which

- at most one formula occurs in the succedent of each sequent, and
- every sequent in the proof that contains a non-atomic formula in its succedent is the lower sequent of the inference figure introducing that formula's top-level connective.

Intuitively, a uniform proof is one in which complex goals are immediately simplified (reading bottom-up).

Definition: $\mathcal{P} \vdash_O G$ if and only if the sequent $\mathcal{P} \longrightarrow G$ has a uniform proof.

Abstract Logic Programming Languages

$\mathcal{L}$	a formulation of logic containing the connectives $\wedge$, $\vee$, $\supset$, $\exists$ and $\forall$ (it may include others, say negation and equality.)
$\vdash_R$	a derivability relation for $\mathcal{L}$-formulas
$\mathcal{G}$	a set of $\mathcal{L}$-formulas (goal formulas).
$\mathcal{D}$	a set of $\mathcal{L}$-formulas (definite or program formulas).

Definition: $\langle \mathcal{D}, \mathcal{G}, \vdash_R \rangle$ is an *abstract logic programming language* (ALPL) if and only if for every finite $\mathcal{P} \subseteq \mathcal{D}$ and $G \in \mathcal{G}$, $\mathcal{P} \vdash_R G$ if and only if $\mathcal{P} \vdash_O G$. See [19] and [20].

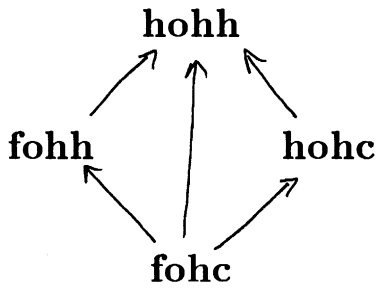
Examples of ALPLs

fohc First-order Horn clauses with classical or intuitionistic provability

hohc Higher-order Horn clauses with classical or intuitionistic provability

fohh First-order hereditary Harrop formulas with intuitionistic provability

hohh Higher-order hereditary Harrop formulas with intuitionistic provability



$\longrightarrow$ denotes containment

Languages Which Are Not Abstract Logic Programming Languages

$$p(a) \vee p(b) \vdash_{I,C} \exists x\, p(x)$$

$$p(a) \vee p(b) \vdash_{I,C} p(b) \vee p(a)$$

$$q \supset p(a), \neg q \supset p(b) \vdash_{C} \exists x\, p(x)$$

No uniform proofs exist in these cases.

$$\dfrac{\dfrac{p(a) \longrightarrow p(a)}{p(a) \longrightarrow \exists x\, p(x)}\,\exists\text{-R} \qquad \dfrac{p(b) \longrightarrow p(b)}{p(b) \longrightarrow \exists x\, p(x)}\,\exists\text{-R}}{p(a) \vee p(b) \longrightarrow \exists x\, p(x)}\,\vee\text{-L}$$

This proof is both classically and intuitionistically valid. It is not, however, uniform.

First-Order Harrop Formulas

$$A := \text{ atomic formula}$$
$$G := \text{ arbitrary formula}$$
$$D := A \mid G \supset D \mid \forall x\ D \mid D_1 \wedge D$$
or
$$D := A \mid G \supset A \mid \forall x\ D \mid D_1 \wedge D_2$$

Theorem (Harrop [8])

Let $\mathcal{H}$ be a set of D-formulas. Then

- If $\mathcal{H} \vdash_I A \vee B$ then $\mathcal{H} \vdash_I A$ or $\mathcal{H} \vdash_I B$.
- If $\mathcal{H} \vdash_I \exists x\ B$ then for some t, $\mathcal{H} \vdash_I [x/t]B$.
- If $\mathcal{H} \vdash_I A \wedge B$ then $\mathcal{H} \vdash_I A$ and $\mathcal{H} \vdash_I B$.
- If $\mathcal{H} \vdash_I A \supset B$ then $A, \mathcal{H} \vdash_I B$.
- If $\mathcal{H} \vdash_I \forall x\ B$ then $\mathcal{H} \vdash_I [x/y]B$ for any new parameter y.

First-Order Hereditary Harrop Formulas

$$A := \text{ atomic formula}$$
$$D := A \mid G \supset A \mid \forall x\ D \mid D_1 \wedge D_2$$
$$G := A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \forall x\ G \mid \exists x\ G \mid D \supset G$$

$\mathcal{D}$	set of closed D-formulas
$\mathcal{G}$	set of closed G-formulas

$\langle \mathcal{F}, \mathcal{D}, \mathcal{G}, \vdash_I \rangle$ is a logic programming language.

$\langle \mathcal{F}, \mathcal{D}, \mathcal{G}, \vdash_C \rangle$ is *not* a logic programming language. For example, there is a classical proof of the sequent

$$(p(a) \wedge p(b) \supset q) \longrightarrow \exists x\ (p(x) \supset q)$$

while there is no uniform proof.

A Classical and Non-Intuitionistic Proof

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{p(a), p(b) \longrightarrow p(a) \quad p(a), p(b) \longrightarrow p(b)}{p(a), p(b) \longrightarrow p(a) \wedge p(b)}\wedge\text{-R} \quad p(a), p(b), q \longrightarrow q}{p(a) \wedge p(b) \supset q, p(a), p(b) \longrightarrow q}\supset\text{-L}}{p(a) \wedge p(b) \supset q, p(a) \longrightarrow q, p(b) \supset q}\supset\text{-R}}{p(a) \wedge p(b) \supset q \longrightarrow p(a) \supset q, p(b) \supset q}\supset\text{-R}}{p(a) \wedge p(b) \supset q \longrightarrow p(a) \supset q, \exists x\,(p(x) \supset q)}\exists\text{-R}}{p(a) \wedge p(b) \supset q \longrightarrow \exists x\,(p(x) \supset q)}\exists\text{-R.}$$

A Nasty Classical Equivalence

$$\begin{aligned} G_1 \vee (D \supset G_2) &\equiv G_1 \vee \neg D \vee G_2 \\ &\equiv \neg D \vee G_1 \vee G_2 \\ &\equiv (D \supset G_1) \vee G_2 \\ &\equiv (D \supset G_1) \vee (D \supset G_2) \end{aligned}$$

The classical equivalence of $p \supset q$ with $\neg p \vee q$ undermines the intended scoping of implications.

$$\dfrac{\dfrac{\dfrac{D \longrightarrow G_1, G_2}{\longrightarrow G_1, D \supset G_2}\supset\text{-R}}{\longrightarrow G_1, G_1 \vee (D \supset G_2)}\vee\text{-R}}{\longrightarrow G_1 \vee (D \supset G_2)}\vee\text{-R}$$

Permitting more than one formula on the right works against our intented interpretation of the logical connectives.

The Sterile Jar Problem

```
type  sterile   jar -> o.
type  bug       insect -> o.
type  dead      insect -> o.
type  heated    jar -> o.
type  in        insect -> jar -> o.
type  j         jar.

sterile Y :- pi X\(bug X=> in X Y=> dead X).
dead X    :- headed Y, in X Y, bug X.
heated j.
```

```
      ?- sterile j
      ?- pi X\(bug X => in X j => dead X)
      ?- bug b => in b j => dead b
bug b ?- (in b j) => (dead b)
in b j ?- dead b
      ?- headed j, in b j, bug b
      ?- headed j
      ?- in b j
      ?- bug b
```


Extending Universal Quantifiers in Goals

Permit $\forall$-quantifiers in goals to quantify functions and predicate symbols.

While this could technically be called a higher-order extension, this extension does not need to be accompanied with λ-terms and higher-order unification to be given a (theoretically) complete implementation. An "essentially first-order" implementation will correctly provide this extension.

This extension simply permit predicates and function symbols to be given scope a long with first-order individuals.

Signatures

Let a *signature* be an association list between tokens and arities (or between tokens and types).

For example, $\Sigma = \{f/1, g/2, a/0, b/0\}$, is a signature.

Let the *Herbrand Universe determined by* Σ, written as $H(\Sigma)$, be the set of all first-order terms built using terms in Σ.

A Σ-formula is a formula all of whose non-logical constants are from Σ.

For the discussion here, we shall permit the confusion of terms from $H(\Sigma)$ with atomic Σ-formulas.

A Non-Deterministic Interpreter

A *state* or our interpreter is a triple $\langle \Sigma, \mathcal{P}, G \rangle$ where

- Σ is the current signature,
- $\mathcal{P}$ is the current program (a set of Σ-formulas), and
- G is the current goal (a Σ-formula).

Defintion: $[\mathcal{P}]_\Sigma$ is the smallest set of **fohh** formulas such

(1) $\mathcal{P} \subseteq [\mathcal{P}]_\Sigma$.

(2) If $D_1 \wedge D_2 \in [\mathcal{P}]_\Sigma$ then $D_1, D_2 \in [\mathcal{P}]_\Sigma$.

(3) If $\forall x \ D \in [\mathcal{P}]_\Sigma$ and $t \in H(\Sigma)$ then $[x := t]D \in [\mathcal{P}]_\Sigma$.

A Non-Deterministic Interpreter (Continued)

$\langle\langle \Sigma, \mathcal{P}, G \rangle\rangle$ denotes the proposition that the interpreter succeeds given the current signature Σ, the current program $\mathcal{P}$, and the goal G.

The interpreter can be describe at a very high-level as follows:

SUCCESS	$\langle\langle \Sigma, \mathcal{P}, true \rangle\rangle$
AND	$\langle\langle \Sigma, \mathcal{P}, G_1 \wedge G_2 \rangle\rangle$ if both $\langle\langle \Sigma, \mathcal{P}, G_1 \rangle\rangle$ and $\langle\langle \Sigma, \mathcal{P}, G_2 \rangle\rangle$.
OR	$\langle\langle \Sigma, \mathcal{P}, G_1 \vee G_2 \rangle\rangle$ if either $\langle\langle \Sigma, \mathcal{P}, G_1 \rangle\rangle$ or $\langle\langle \Sigma, \mathcal{P}, G_2 \rangle\rangle$.
INSTANCE	$\langle\langle \Sigma, \mathcal{P}, \exists x\ G \rangle\rangle$ if for some $t \in H(\Sigma)$, $\langle\langle \Sigma, \mathcal{P}, [x := t]G \rangle\rangle$.
AUGMENT	$\langle\langle \Sigma, \mathcal{P}, D \supset G \rangle\rangle$ if $\langle\langle \Sigma, \mathcal{P} \cup \{D\}, G \rangle\rangle$.
GENERIC	$\langle\langle \Sigma, \mathcal{P}, \forall x\ G \rangle\rangle$ if for some $c \notin \Sigma$, $\langle\langle \Sigma \cup \{c\}, \mathcal{P}, [x := c]G \rangle\rangle$.
BACKCHAIN	$\langle\langle \Sigma, \mathcal{P}, A \rangle\rangle$ (where A is atomic) if either $A \in [\mathcal{P}]_\Sigma$ or $G \supset A \in [\mathcal{P}]_\Sigma$ and $\langle\langle \Sigma, \mathcal{P}, G \rangle\rangle$.

A Deterministic Interpreter

Add a depth-first discipline to backtracking.

Use logical variables (free variables) in BACKCHAIN and INSTANCE instead of guessing at a closed term.

Process conjuncts and disjuncts in a left-to-right order.

When adding a clause during AUGMENT, add it to the top of the list.

In BACKCHAIN, select clauses in a top-down fashion.

How does one handle the problem of quantifier alternation? How to modify unification in the presence of the restriction posed by GENERIC?

Four Implementations of GENERIC

Given the query

$$\mathcal{P}(\bar{x}) \quad ?- \quad \forall y(G(y, \bar{z}))$$

where $\bar{x}$ and $\bar{z}$ are lists of free (logical) variables (possibly overlapping).

(1) Reduce to $\mathcal{P}(\bar{x}) \quad ?- \quad G(c, \bar{z})$ where c is a new constant (added to the current signature). Modify unification to respect the constraint that the variables in $\bar{x}$ and $\bar{z}$ cannot get instantiated with terms containing c.

(2) Reduce to $\mathcal{P}(\bar{x}) \quad ?- \quad G(f(\bar{x}, \bar{z}), \bar{z})$ where f is a skolem function. Unification is unchanged. The occur-check is required to enforce restriction.

(3) Higher-order unification provides a different approach (called *raising*).

(4) Keep an explicit prefix as a constraint.

Raising: A Dual to Skolemization

Notice that inner-most universals are related to λ-abstraction.

$$\vdash \exists x \forall y \, [t_1 = s_1 \wedge \ldots \wedge t_n = s_n]$$

if and only if

$$\vdash \exists x \, [(\lambda y \, t_1 = \lambda y \, s_1) \wedge \ldots \wedge (\lambda y \, t_n = \lambda y \, s_n)]$$

While a prefix can be simplified by having Skolemization introduce new constants of higher-type, prefixes can also be simplified by introducing new variables of higher-type.

$$\vdash \forall x \exists y \forall z . P(x, y, z)$$

if and only if

$$\vdash \exists h \forall x \forall z . P(x, h(x), z)$$

This approach is used in LP2.7. See [15] for complete description and correctness proofs.

Explicit Prefix as a Constraint

Consider quantified sequents for representing current states with free varialbes. The free variables are existentially quantified while members of the signature are universally quantified. The position of an existential quantifiers in the prefix determines which constants can appear in the substitution term for the existentially quantified variable.

For example,

$$\forall x\ \exists y\ \forall z\ \exists u\ (\mathcal{P} \longrightarrow G)$$

describes a state with signature $\{x, z\}$ and where the logical variable y can be instantiated with a term from $H(\{x\})$ and the logical variable u can be instantiated with a term from $H(\{x, z\})$.

INSTANCE and BACKCHAIN add $\exists$-quantifiers to the prefix.

GENERIC adds $\forall$-quantifiers to the prefix.

This approach is used in eLP. See [15] for complete description and correctness proofs.

Lecture V
An Approach to Modules and Lexical Scoping

Table of Contents

Formulas That Are Both Program Clauses and Goal Formulas

Theorem: If M is both a program clause and a goal formula then $\Gamma \vdash_O M \wedge G$ if and only if $\Gamma \vdash_O M \wedge [M \supset G]$.

Such formulas can be stored after being proved to hold. Such storing does not make new goals provable. Instead it possibly provides shorter proofs for existing provable goals.

The *core* of an abstract logic programming language $\langle \mathcal{L}, \mathcal{D}, \mathcal{G}, \vdash_R \rangle$ is the intersection, $\mathcal{D} \cup \mathcal{G}$.

The core of (extended) **fohh** is

$$M := A \mid M \supset A \mid M_1 \wedge M_2 \mid \forall x\, M,$$

where the universal quantification is strictly first-order. This is the fragment of **fohh** that does not contain occurrences of disjunctions of existential quantifiers. It contains **fohc**.

The core for **fohc** is simply the set of closed atomic formulas.

Extension Tables

We use the very simple example of the Fibonacci program to illustrate how implicational goals can be used to build "scoped extension tables."

```
fib(0,0).
fib(1,1).
fib(N,F) :- N1 is N-1, N2 is N-2, fib(N1,F1),
            fib(N2,F2), F is F1+F2.


fib(N,M) :-
     memo(0,0) => memo(1,1) => fiba(N,M,2).
fiba(N,M,I) :- memo(N,M).
fiba(N,M,I) :-
     N1 is I-1, N2 is I-2, memo(N1,F1),
     memo(N2,F2), F is F1+F2, I1 is I+1,
     memo(I,F) => fiba(N,M,I1).
```

Example of Lexical Scoping

```
reverse L K :- pi Rev\ (
  (pi L\                    (Rev [] L L),
   pi X\pi L\pi K\pi M\(Rev [X|L] K M :-
                          Rev L K [X|M]))
       => Rev L K [])
```

```
reverse L K :- pi Rev\ (
  (                 (Rev [] K),
   pi X\ pi L\ pi M\(Rev [X|L] M :-
                       Rev L [X|M]))
       => Rev L [])
```


Implementing Fail and Succeed

How do we implement the predicate `fail` that is never provable? One way is to have the program for `fail` be empty.

In this dynamic logic (**fohh**), a programmer may add to the current program clauses that add meaning to `fail`.

The goal

$$\forall p\ .p$$

will always fail: it picks a new predicate name, that is, it is guaranteed to have no program clauses defining it, and then a proof for it is attempted.

Similarly, how do you implement the predicate `succeed` which is to succeed exactly once?

The goal

$$\forall p\ .p \supset p$$

will succeed exactly once.

Minimal Logic Negation

Pick $\perp$ as a special non-logical constant.
Expressions of the form $A \supset \perp$ will be read as $\neg A$.

$$p(a) \wedge p(b) \supset \perp$$

$$p(a)$$

$$?- p(b)$$

$$?- p(b) \supset \perp$$

$$p(a) \wedge p(b) \supset \perp$$

$$p(a)$$

$$p(b)$$

$$?- p(c)$$

See [13] and [12] for more on this kind of negation.

A Simple Database Example

```
enrolled(jane,102).
enrolled(bill,100).
⊥ :- enrolled(X,101),enrolled(X,102).

db :- read(Command), do(Command), db.
do(enter(Fact)) :- Fact => db.
do(retract) :- fail.
do(commit) :- repeat.
do(check(Query)) :-
   (Query, write(yes), nl,!;
    Query => ⊥, write(no),nl,!;
    write('no, but it could be true'),nl).
do(consis) :- (not ⊥, write(yes),!;
               write(no)), nl.

?- db.
?- check(enrolled(jane,102)).
yes
?- check(enrolled(jane,101)).
no
?- check(enrolled(bill,101)).
no, but it could be true
?-
```

Reimplementing Consult

Let `classify`, `scanner`, `misc` be the name of files containing Prolog code.

Consider solving the goal

```
misc => ((classify => (G1, scanner => G2)),
         G3).
```

An interpreter will need to consider showing

- `G1` from `misc` and `classify`,
- `G2` from `misc`, `classify`, and `scanner`, and
- `G3` from `misc`.

"New" code becomes accessible and disappears in a stack-disciplined fashion.

Modules

`module` *ModuleName*.
Declarations of operators, types, modes, etc.
Collection of clauses

For example,

```
module lists.
append([],X,X).
append([U|L],X,[U|M]) :- append(L,X,M).
member(X,[X|L]) :- !.
member(X,[Y|L]) :- member(X,L).
memb(X,[X|L]).
memb(X,[Y|L]) :- memb(X,L).
```

Parametric Modules

```
module sort(Order).
bsort(L1,L2) :-
        append(Sorted,[Big,Small|Rest],L1),
        Order(Big,Small),
        !,
        append(Sorted,[Small,Big|Rest],L3),
        bsort(L3,L2).
bsort(L1,L1).
```


Combining Modules

```
?- lists => sort(<) => bsort([3,2,1],X)

  lists, sort(<) ?- bsort([3,2,1],X)
```

```
module sort(Order).
bsort(L1,L2) :-
   (lists =>
      (append(Sorted,[Big,Small|Rest],L1),
       Order(Big,Small),
       !,
       append(Sorted,[Small,Big|Rest],L3),
       bsort(L3,L2)
      ).
bsort(L1,L1).
```

Importing Modules

module $\mathbf{M_1}$	module $\mathbf{M_2}(x)$	module $\mathbf{M_3}(y,z)$
		import $\mathbf{M_1}$ $\mathbf{M_2}(y)$
$\mathcal{P}_1$	$\mathcal{P}_2(x)$	$\mathcal{P}_3(z)$

For each clause of the form

$$\forall \bar{w}(G \supset A)$$

in $\mathcal{P}_3$ replace it with one of the form

$$\forall \bar{w}((\mathbf{M_1} \wedge \mathbf{M_2}(y)) \supset G) \supset A)$$

See [13] and [12] for several examples of using this form of importing.

Programs as Possible Worlds

Fix the signature Σ and assume that universal quantifiers are removed from all goals and the body of programs.

- Let $\mathcal{W}$ be the set of all programs. This set will be used as the set of possible worlds.
- A function I from $\mathcal{W}$ to a subset of $H(\Sigma)$ is an *interpretation* if

$$\forall w_1, w_2 \in \mathcal{W}[w_1 \subseteq w_2 \supset I(w_1) \subset I(w_2)].$$

- $\langle \mathcal{W}, \subseteq, I \rangle$ is a Kripke model.
- Define each of the following for interpretations I_1 and I_2.

$$I_1 \sqsubseteq I_2 := \forall w \in \mathcal{W}[I_1(w) \subset I_2(w)]$$
$$(I_1 \sqcup I_2)(w) := I_1(w) \cup I_2(w)$$
$$(I_1 \sqcap I_2)(w) := I_1(w) \cap I_2(w)$$

- The set of interpretations is a complete lattice under $\sqsubseteq$.
- The minimal interpretation is $I_\perp$ where $I_\perp(w) = \emptyset$ for all $w \in \mathcal{W}$.

A Continuous Operator on Interpretations

Define $I, w \nVdash G$ as follows:

- $I, w \nVdash \top$.
- $I, w \nVdash A$ if $A \in I(w)$.
- $I, w \nVdash G_1 \wedge G_2$ if $I, w \nVdash G_1$ and $I, w \nVdash G_2$.
- $I, w \nVdash G_1 \vee G_2$ if $I, w \nVdash G_1$ or $I, w \nVdash G_2$.
- $I, w \nVdash D \supset G$ if $I, w \cup \{D\} \nVdash G$.

Define T as a mapping from interpretation to interpretations as follows:

$$T(I)(w) := \{A \mid \text{if } A \in [w]_\Sigma \text{ or } G \supset A \in [w]_\Sigma \text{ and } I, w \nVdash G\}$$

Kripke-model Fixed Point

The least fixed point of T is

$$T^\infty(I_\perp) := T(I_\perp) \sqcup T^2(I_\perp) \sqcup T^3(I_\perp) \sqcup \ldots$$

and has the following properties:

Theorem: If $\mathcal{P}$ is a program and G is a goal formula, then $\mathcal{P} \vdash_I G$ if and only if $T^\infty(I_\perp), \mathcal{P} \nVdash G$. (See [13] and [12].)

Theorem: If G is a goal formula and $\vdash_I G$ then G is true in $T^\infty(I_\perp)$ in the usual Kripke model sense (replace $\nVdash$ with $\models$).

A Mechanism for Abstract Datatypes

Consider solving the goal

$$\exists x\, \forall y\, (D(y) \supset G(x)).$$

- Substitution terms determined for x cannot contain the constant introduced for y.
- $\forall$ provides a means for *hiding* data in modules.

Allow existential quantifiers around program clauses. Such existential quantifiers are interpreted as follows:

$$(\exists x\, D) \supset G \qquad \equiv \qquad \forall x\, (D \supset G)$$

provided x is not bound in G (otherwise, rename x first).

This is intuitionistically (hence, classically) valid.

Stacks as Abstract Datatypes

Let `stack` stand for the following expression:

$$\begin{aligned}\exists empty\, \exists stk\, [\ & emptystack(empty) \wedge \\ & \forall s \forall x(push(x, s, stk(x, s))) \wedge \\ & \forall s \forall x(pop(x, stk(x, s), s))]\end{aligned}$$

$$?- \quad \exists x(stack \supset \exists y[G(x, y)])$$

$$?- \quad \exists x\, \forall empty\, \forall stk\, (stack' \supset \exists y[G(x, y)])$$

Module Definition for Stackes

```
module stack.

kind stack type -> type.
type empty (stack A) -> o.
type pop   A -> (stack A) -> (stack A) -> o.
type push  A -> (stack A) -> (stack A) -> o.

local  emp  (stack A).
local  stk  A -> (stack A) -> (stack A).

empty emp.
pop   X   S          (stk X S).
push  X   (stk X S)  S.
```


Binary Trees As an Abstract Data Types

```
module btreesort Order.
import lists.
local insert   int -> bt -> o.
local traverse btree -> list int -> o.
local root     bt.
local bt       int -> bt -> bt -> bt.
local build    list int -> bt -> o.
type  btsort   list int -> list int -> o.

btsort L K :- build L Bt, traverse Bt K.

build [] T.
build [N|L] T :- insert N T, build L T.

insert N (bt M T S) :- N = M, !.
insert N (bt M T S) :- Order N M, !,
                       insert N T.
insert N (bt M S T) :- insert N T.

traverse root [].
traverse (bt N Left Right) L :-
    traverse Left K, traverse Right J,
    append K [N|J] L.
```

Another Way to Connect Modules

```
module mod1.

p X Y   :- .... .
q X Y Z :- .... .
r X Y   :- .... .


module mod2.

p X Y :- pi p\ pi q\ pi r\
          (mod1 => p X Y).
t X Y :- pi p\ pi q\ pi r\
          (mod1 => q (f X) [] Y).
```


Encapsulating State

```
nodule accounts.

type print_amt    account ->         o -> o.
type wd_money     account -> int -> o -> o.
type add_money    account -> int -> o -> o.
type make_account account -> int -> o -> o.

nake_account Acc Amt G :- pi Reg\ (
  (  (Reg Amt),
     (pi Inc\ (pi H\ (pi Tmp\
        (add_money Acc Inc H :-
           Reg Val, Tmp is (Val + Inc),
           Reg Tmp => H) ))),
     (pi Dec\ (pi H\ (pi Tmp\
        (wd_money Acc Dec H :-
           Reg Val, Tmp is (Val - Dec),
           Reg Tmp => H) ))),
     (pi Acc\ (pi H\ (pi Val\
        (print_amt Acc H :-
           Reg Val, write Val, nl, H)))))
  => G).
```

Encapsulating State (continued)

```
type transactions o.
type quit         o -> o.

transactions :- write ">>- ",
  read Entry, ( Entry = quit, !;
                Entry transactions).

?- transactions.
>>- make_account john 10.
>>- add_money john 5.
>>- print_amt john.
15
>>- wd_money john 14.
>>- print_amt john.
1
>>- quit.
?-
```


A Need for Embedded Implications

Assume that the binary relation compare is defined in the module compound.

```
?- compmod => btreesort compare =>
      btree [3,1,5] L.
?- btreesort (X\Y\(compmod=>compare X Y))
     => btree [3,1,5] L.
?- write "Enter an order relation",
   read Order,
   btreesort Order => btree [3,1,5] L,
   write L.
```

For more on how to get thes program-level abstractions out of (extended) **fohh** see [14].

Lecture VI
Higher-Order Hereditary Harrop Formulas

Table of Contents

How Can Hereditary Harrop Formulas be Made Higher-Order?

This question can loosely be phrased as "Can program-level abstraction can be reflected into terms?" In particular, can modules be embedded inside terms?

There seem to be two general approaches to answering this question.

Dynamic — This approach permits such full reflection. Serious kinds of run time errors, however, can occur. The language is very strong since it contains a kind of `eval` or `apply` operator.

Static — This approach restricts such reflection. As a result, we can prove the that the resulting language has no run time errors. This conservative approach, however, disallows many sensible computations.

This dichotomy, which is illustrated on the following slides, can be dealt with as follows:

- Implement the Dynamic language.
- Prove theorems about the Static language.

The Dynamic Approach

Let A denote atomic formulas of the form

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

where

P is a non-logical constant or variable, and

t_i is a simply typed λ-term perhaps with embedded $\wedge$, $\vee$, $\supset$, $\exists$, and $\forall$.

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

$$G ::= A \mid G_1 \vee G_2 \mid G_1 \wedge G_2 \mid \exists x\, G \mid D \supset G \mid \forall x\, G$$

$$D ::= A \mid G \supset A \mid \forall x\, D \mid D_1 \wedge D_2$$

Strengths of the Dynamic Language

Permits predicates substitutions to carry around their own code.

```
?- btreesort X\Y\(compmod => compare X Y)
     => btree [3,1,5] L.
```

After computing a term that denotes a program, make it into an available program.

```
?- transform Spec Prog, Prog => G.
```

Reflection makes meta-interpreters very simple.

```
tl :- nl, read Command, do Command.

do quit.
do (enter Prog) :- Prog => tl.
do (solve Goal) :- (Goal, !, write "Yes";
                                write "No" ),
                   tl.
```

Problems with the Dynamic Language

With negatively occurring predicate variables in goal formulas, it is not possible to guarantee that the current program is always a subset of $\mathcal{D}$. For example, in the goal,

```
?- transform Spec Prog, Prog => G.
```

`transform` could output a formula with a top-level disjunct.

More seriously, some intuitionistic provable goals formulas do not have uniform proofs. The following such goal (in the dynamic language) is due to Pfenning.

$$\exists Q[\forall p \forall q[R(p \supset q) \supset R(Qpq)] \wedge Q(t \vee s)(s \vee t)].$$

Here R is a constant of type $o \rightarrow o$, s and t are constants of type o, Q is a variable of type $o \rightarrow o \rightarrow o$, and p and q are constants of type o.

The only substitution term for Q is $\lambda x \lambda y(x \supset y)$. Any proof of this goal must contain within it a proof of the sequent $t \vee s \longrightarrow s \vee t$.

The Static Language: Higher-Order Hereditary Harrop Formulas

Let A denote atomic formulas of the form

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

where

P	is a nonlogical constant or variable, and
t_i	is a simply typed λ-term perhaps with embedded $\wedge$, $\vee$, $\exists$, and $\forall$ (no $\supset$).

Let A_r denote such a formula where P is a nonlogical constant. These are called *rigid atoms.*

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

$$G ::= A \mid G_1 \vee G_2 \mid G_1 \wedge G_2 \mid \exists x\, G \mid D \supset G \mid \forall x\, G$$

$$D ::= A_r \mid G \supset A_r \mid \forall x\, D \mid D_1 \wedge D_2$$

Then **hohh** $= \langle \mathcal{T}, \mathcal{D}, \mathcal{G}, \vdash_I \rangle$, where

$\mathcal{T}$	denotes our higher-order logic, and
$\vdash_I$	denotes intuitionistic provability.

A Meta Interpreter

```
module interpreter.
import lists.

type  interp  (list o) -> o -> o.
type  instan  o -> o -> o.

interp Cl true.
interp Cl (G1 , G2) :-
   interp Cl G1 , interp Cl G2.
interp Cl (G1 ; G2) :-
   interp Cl G1 ; interp Cl G2.
interp Cl (D => G) :- interp [D|Cl] G.
interp Cl (sigma G) :-
     sigma T\ (interp Cl (G T)).
interp Cl (pi G) :-
     pi X\ (interp Cl (G X)).
interp Cl A :-
     memb Clause Cl, instan Clause Inst,
   ( Inst = A ; Inst = (A :- G),
                interp Cl G ).

instan (pi P) C :- instan (P T) C.
instan C C.
```


Why This Interpreter Does Not Interpret Itself

The clauses which involve "internal quantifiers" are polymorphic. That is, the quantification is over variables of unspecified type.

Consider the `instan` predicate.

```
instan (forall P) C :- instan (P T) C.
instan C C.
```

There is an implicit universal quantification of a type variable for the type of `T` in the first clause. If this program is made into a list of clauses, say

```
[pi C\(pi P\ (instan (forall P) C :-
                        instan (P T) C)),
      pi C\ (instan C C)],
```

to be fed to `interp`, then this implicit type quantification is lost. It is instead existentially quantified by being made a free type variable.

Specifying the Formulas of an Object Logic

The following module provides the signature for formulas of a first-order logic.

```
module logic.

infix   110 and xfy.
infix   110 or  xfy.
infix   120 imp xfy.

kind    i       type.
kind    bool    type.

type    and     bool -> bool -> bool.
type    or      bool -> bool -> bool.
type    imp     bool -> bool -> bool.
type    neg     bool -> bool.
type    forall  (i -> bool) -> bool.
type    exists  (i -> bool) -> bool.
type    false   bool.
```

The formula $\forall x\ \exists y\ (p(y) \supset p(x))$ is written as the term

```
forall X\ (exists Y\ (p X imp p Y)).
```


Negation Normal: Propositional Part

Negation normal formulas are those first-order formulas in which negations have atomic scope.

```
module nnf.
import logic.

type  nnf  bool -> bool -> o.

nnf (A and B) (C and D) :-
      nnf A C, nnf B D.
nnf (A or B) (C or D) :-
      nnf A C, nnf B D.
nnf (A imp B) (C or D) :-
      nnf (neg A) C, nnf B D.
nnf (neg (neg A)) B :-
      nnf A B.
nnf (neg (A and B)) (C or D) :-
       nnf (neg A) C, nnf (neg B) D.
nnf (neg (A or B)) (C and D) :-
      nnf (neg A) C, nnf (neg B) D.
nnf (neg (A imp B)) (C and D) :-
      nnf A C, nnf (neg B) D.
```

Negation Normal: Quantificational Part

```
nnf (forall A) (forall B) :-
      pi X\ (nnf (A X) (B X)).
nnf (exists A) (exists B) :-
      pi X\ (nnf (A X) (B X)).
nnf (neg (forall A)) (exists B) :-
      pi X\ (nnf (neg (A X)) (B X)).
nnf (neg (exists A)) (forall B) :-
      pi X\ (nnf (neg (A X)) (B X)).

nnf A A.
```


Specifying Inference Rules

```
type  proof   sequent -> prf -> o.
type  -->     (list bool) -> bool -> sequent.
infix 100 --> xfy.
```

$$\frac{\Gamma \longrightarrow \Delta, B \qquad \Gamma \longrightarrow \Delta, C}{\Gamma \longrightarrow \Delta, B \wedge C}\ \wedge\text{-R}$$

```
type  and_r  prf -> prf -> prf.

proof (Gamma --> (A and B)) (and_r P1 P2) :-
    proof (Gamma --> A) P1,
    proof (Gamma --> B) P2.
```

Specifying Inference Rules (continued)

$$\frac{\Gamma \longrightarrow \Delta, B}{\Gamma \longrightarrow \Delta, B \vee C}\ \vee\text{-R} \qquad \frac{\Gamma \longrightarrow \Delta, C}{\Gamma \longrightarrow \Delta, B \vee C}\ \vee\text{-R}$$

```
proof (Gamma --> (A or B)) (or_r P) :-
    proof (Gamma --> A) P;
    proof (Gamma --> B) P.
```

$$\frac{\Gamma, [x/t]P \longrightarrow \Theta}{\Gamma, \forall x\, P \longrightarrow \Theta}\ \forall\text{-L} \qquad \frac{\Gamma \longrightarrow \Theta, [x/t]P}{\Gamma \longrightarrow \Theta, \exists x\, P}\ \exists\text{-R}$$

```
proof (Gamma --> (exists A)) (exists_r P) :-
    proof (Gamma --> (A T)) P.
```

$$\frac{\Gamma \longrightarrow \Theta, [x/y]P}{\Gamma \longrightarrow \Theta, \forall x\, P}\ \forall\text{-R}$$

```
proof (Gamma --> (forall A)) (forall_r P) :-
    pi T\ (proof (Gamma --> (A T)) (P T)).

type forall_r  (i -> prf) -> prf
```


Natural Deduction Rules

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

$$\frac{[x/t]A}{\exists x\, A}\ \exists\text{–I} \qquad \frac{[x/y]A}{\forall x\, A}\ \forall\text{–I}$$

```
proof (A and B) (and_i P1 P2) :-
    proof A P1,
    proof B P2.
proof (A or B) (or_i P) :-
    proof A P; proof B P.
proof (exists A) (exists_i P) :-
    proof (A T) P.
proof (forall A) (forall_i P) :-
    pi T\ (proof (A T) (P T)).
```

Specifying the Discharge of Assumptions

$$\frac{\begin{array}{c}(A)\\ B\end{array}}{A \supset B}\ \supset\text{-I}$$

```
proof (A imp B) (imp_i P) :-
    pi PA\ ((proof A PA) =>
                (proof B (P PA))).
type   imp_i  (prf -> prf) -> prf.
```


Inference Rules As Tactics

Atomic Goals

```
type  pgoal  sequent -> prf -> goalexp.
```

```
(pgoal (Gamma --> A) P)
```

$$\frac{\Gamma \longrightarrow \Delta, B \qquad \Gamma \longrightarrow \Delta, C}{\Gamma \longrightarrow \Delta, B \wedge C}\ \wedge\text{-R}$$

```
proof (Gamma --> (A and B)) (and_r P1 P2) :-
     proof (Gamma --> A) P1,
     proof (Gamma --> B) P2.

and_r_tac (pgoal (A and B) (and_i P1 P2))
          (andgoal (pgoal A P1)
                   (pgoal B P2)).
```

Inference Rules As Tactics (continued)

$$\frac{\Gamma, [x/y]P \longrightarrow \Theta}{\Gamma, \exists x\, P \longrightarrow \Theta}\ \exists\text{-L}$$

```
exists_l_tac
  (pgoal (Gamma1 --> A) (exists_i P))
  (allgoal X\ (pgoal ([(B X)|Gamma2] --> A)
                     (P X)))
  :-
    memb_and_rest (exists B) Gamma1 Gamma2.
```


A Goal Reduction Tactical

```
type truegoal goalexp.
type andgoal  goalexp -> goalexp ->
                              goalexp.
type allgoal  (A -> goalexp) -> goalexp.
type maptac   (goalexp -> goalexp -> o) ->
              (goalexp -> goalexp -> o) -> o.

maptac Tac truegoal truegoal.

maptac Tac (andgoal InGoal1 InGoal2)
           (andgoal OutGoal1 OutGoal2) :-
   maptac Tac InGoal1 OutGoal1,
   maptac Tac InGoal2 OutGoal2.

maptac Tac (allgoal InGoal)
           (allgoal OutGoal) :-
  pi T\ (maptac Tac (InGoal T) (OutGoal T)).

maptac Tac InGoal OutGoal :-
       Tac InGoal OutGoal.
```

Tacticals

```
then Tac1 Tac2 InGoal OutGoal :-
   Tac1 InGoal MidGoal,
   maptac Tac2 MidGoal OutGoal.

orelse Tac1 Tac2 InGoal OutGoal :-
   Tac1 InGoal OutGoal;
   Tac2 InGoal OutGoal.

idtac Goal Goal.

repeat Tac InGoal OutGoal :-
   orelse (then Tac (repeat Tac))
          idtac InGoal OutGoal.

try Tac InGoal OutGoal :-
   orelse Tac idtac InGoal OutGoal.

complete Tac InGoal truegoal :-
   Tac InGoal OutGoal,
   goalred OutGoal truegoal.
```


Simplifying Some Goal Expressions

```
goalred (andgoal truegoal Goal) OutGoal :-
    goalred Goal OutGoal.

goalred (andgoal Goal truegoal) OutGoal :-
    goalred Goal OutGoal.

goalred (allgoal T\ truegoal) truegoal.

goalred Goal Goal.
```

Interactive Theorem Proving

```
query (pgoal A P) OutGoal :-
   write A, write "Enter tactic:", read Tac,
   Tac (pgoal A P) OutGoal.

interactive InGoal OutGoal :-
   repeat query InGoal OutGoal.

and_e_query (pgoal C PC)
            (impgoal (proof A (and_e1 P))
               (impgoal (proof B (and_e2 P))
                  (pgoal C PC))) :-
  memo (hyp (A and B) P),
  write "Eliminate this conjunction?",
  write (A and B),
  read "yes".
```

For more examples on building theorem provers in this fashion, see Felty and Miller [4].

The Copy Verification Program

The goal

```
?- copy_ver Tacs Copy In Out
```

attempts to repeatly copy the goal structure in Copy onto the goal In to get the goal Out. Tacs provides the methods for decomposing Copy.

```
copy_ver Tacs (andgoal C1 C2)
                  (andgoal I1 I2) Out :-
  copy_ver Tacs C1 I1 O1,
  copy_ver Tacs C2 I2 O2,
  goalred (andgoal O1 O2) Out.

copy_ver Tacs (allgoal C) (allgoal I) Out :-
  pi T\(copy_ver Tacs (C T) (I T) (O T)),
  goalred (allgoal O) Out.

copy_ver Tacs Copy In Out :-
  memb Tac Tacs,
  Tac Copy NewC, Tac In Mid,
  maptac (copy_ver Tacs NewC) Mid Out.

copy_ver Tacs Copy Goal Goal.
```