Logic in Logic Programming: Sequent Calculus, Higher-Orders, and Linear Logic

Fourth International School for
Computer Science Researchers
Acireale, Sicily
29 June – 3 July 1992

Dale Miller
Computer Science Department
University of Pennsylvania
Philadelphia, PA 19104–6389 USA
dale@cis.upenn.edu

Some corrections have been made on 5 July 1992.

Sequents

Let Δ and Γ be finite (possibly empty) multisets of propositional formulas.
A sequent is a pair

$$\Delta \longrightarrow \Gamma,$$

where Δ is the *antecedent* (*left*) and Γ is the *succedent* (*right*).
The intended interpretation of $\Delta \longrightarrow \Gamma$ is "If all the formulas in Δ hold, then some formula in Γ holds." That is,

$$B_1, \ldots, B_n \longrightarrow C_1, \ldots, C_m$$

denotes approximately the formula

$$(B_1 \wedge \cdots \wedge B_n) \supset (C_1 \vee \cdots \vee C_m).$$

Soundness of the "Rule of Cases"

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

A Sequent Proof System: Structural Rules

Multiset union:

$$\Gamma, B := \Gamma \cup \{B\} \qquad \Gamma_1, \Gamma_2 := \Gamma_1 \cup \Gamma_2$$

Contraction

$$\frac{\Gamma, B, B \longrightarrow \Delta}{\Gamma, B \longrightarrow \Delta}\, C_L \qquad \frac{\Gamma \longrightarrow \Delta, B, B}{\Gamma \longrightarrow \Delta, B}\, C_R$$

Weakening

$$\frac{\Gamma \longrightarrow \Delta}{\Gamma, B \longrightarrow \Delta}\, W_L \qquad \frac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \Delta, B}\, W_R$$

The figure

$$\frac{\Gamma \longrightarrow \Delta}{\Gamma' \longrightarrow \Delta'}$$

means that zero or more applications of weakening and contraction rules have been used.

A Sequent Proof System: Introduction Rules

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

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

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

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

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

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

$$\frac{\Delta \longrightarrow \Gamma, \bot}{\Delta \longrightarrow \Gamma, B} \bot\text{-R} \qquad \frac{}{B \longrightarrow B} \text{ initial}$$

A Sequent Proof System: The Cut-Rule

$$\frac{\Delta' \longrightarrow \Gamma', B \qquad B, \Delta'' \longrightarrow \Gamma''}{\Delta', \Delta'' \longrightarrow \Gamma', \Gamma''}\ \text{cut}$$

For example, having a proof of

$$\Delta' \longrightarrow B \quad \text{and} \quad \Delta'', B \longrightarrow C$$

entails having a proof of $\Delta', \Delta'' \longrightarrow C$. Thus, if $\longrightarrow$ is seen as an implication, then cut corresponds to modus ponens. The formula B is used as a lemma.

Obviously, the use of the word "cut" here has nothing to do with the control primitive called cut in Prolog.

Examples of Sequent Proofs

$$\dfrac{\dfrac{p \longrightarrow p \qquad \dfrac{q \longrightarrow q \qquad r \longrightarrow r}{q, q \supset r \longrightarrow r}}{p, q, p \supset (q \supset r) \longrightarrow r}}{\dfrac{p, p \wedge q, p \supset (q \supset r) \longrightarrow r}{\dfrac{p \wedge q, p \wedge q, p \supset (q \supset r) \longrightarrow r}{\dfrac{p \wedge q, p \supset (q \supset r) \longrightarrow r}{p \supset (q \supset r) \longrightarrow (p \wedge q) \supset r}}}}$$

$$\dfrac{\dfrac{p \longrightarrow p \qquad \bot \longrightarrow \bot}{p, p \supset \bot \longrightarrow \bot}}{p \longrightarrow (p \supset \bot) \supset \bot}$$

$$\dfrac{\dfrac{\dfrac{p \longrightarrow p}{p \longrightarrow \bot, p}}{\longrightarrow p \supset \bot, p} \qquad \dfrac{\bot \longrightarrow \bot}{\bot \longrightarrow p}}{\dfrac{(p \supset \bot) \supset \bot \longrightarrow p, p}{(p \supset \bot) \supset \bot \longrightarrow p}}$$

Some Definitions

A tree of inference rules is a *proof* of its root if all leaves are initial.
A proof is *atomically closed* if for every initial sequent $B \longrightarrow B$, the formula B is atomic or $\bot$.
A proof is *cut-free* if it contains no occurrences of the cut rule.

C-proof	a sequent proof (classical)
I-proof	a C-proof where all sequents have singleton succedents (intuitionistic)
M-proof	an I-proof with no occurrences of the $\bot$-R rule. (minimal)

$\Delta \vdash_C \Gamma$	if $\Delta \longrightarrow \Gamma$ has a C-proof.
$\Delta \vdash_I \Gamma$	if $\Delta \longrightarrow \Gamma$ has an I-proof.
$\Delta \vdash_M \Gamma$	if $\Delta \longrightarrow \Gamma$ has an M-proof.

Negation is defined as $\neg B := B \supset \bot$.
$\bot$ is not an atomic formula.

Searching for Sequent Proofs

Given a sequent, find a proof of it, if possible.

- Classical propositional logic is NP-complete.
- Intuitionistic propositional logic is P-space complete.

A problem with cut

$$\frac{\Delta' \longrightarrow \Gamma', B \qquad B, \Delta'' \longrightarrow \Gamma''}{\Delta', \Delta'' \longrightarrow \Gamma', \Gamma''}\ \text{cut}$$

Notice that this is the only inference rule that does not have the *subformula property*: any formula occurring in a premise is a subformula of a formula occurring in the conclusion. Such a property clearly helps to constrain the search for proofs.

To use cut during a bottom-up search, we must "invent" the new formula B. Such invention is generally very difficult.

Search: Splitting Contexts

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

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

To use these rules, the multisets Δ', Δ'' and Γ', Γ'' must be divided into the pair of multisets Δ' and Δ'', and Γ' and Γ''.

Given a multiset of n elements, there are 2^n ways to do this split. Liberal applications of weakening and contraction (if available) can simplify this problem.

$$\dfrac{\dfrac{\Gamma \longrightarrow \Delta, B \qquad \Gamma \longrightarrow \Delta, C}{\Gamma, \Gamma \longrightarrow \Delta, \Delta, B \wedge C}}{\Gamma \longrightarrow \Delta, B \wedge C}$$

Search: Problems with Structural Rules

Contraction and weakening can be used at any point in a search. Contraction can be used arbitrarily often (in the first-order setting, no bound on contractions can be set).

Explicit weakening can be removed by using initial sequents of the form

$$\Delta, B \longrightarrow B, \Gamma.$$

Some forms of contractions can be factored into inference rules by modifying some rules. For example, consider the following modified introduction rules for conjunction.

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

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

Search: Permutations of Inference Rules

The order in which inference rules are applied is often not important. Consider the following two proof fragments.

$$\dfrac{\dfrac{a, b, c, \Delta \longrightarrow \Gamma \qquad a, b, d, \Delta \longrightarrow \Gamma}{a, b, c \vee d, \Delta \longrightarrow \Gamma}\ \vee\text{-L}}{a \wedge b, c \vee d, \Delta \longrightarrow \Gamma}\ \wedge\text{-L}$$

$$\dfrac{\dfrac{a, b, c, \Delta \longrightarrow \Gamma}{a \wedge b, c, \Delta \longrightarrow \Gamma}\ \wedge\text{-L} \qquad \dfrac{a, b, d, \Delta \longrightarrow \Gamma}{a \wedge b, d, \Delta \longrightarrow \Gamma}\ \wedge\text{-L}}{a \wedge b, c \vee d, \Delta \longrightarrow \Gamma}\ \vee\text{-L}$$

Thus, $\vee$-L *permutes over* $\wedge$-L.

The Cut-Elimination Theorem

A sequent $\Delta \longrightarrow \Gamma$ has a C-proof (resp., I-proof, M-proof) if and only if $\Delta \longrightarrow \Gamma$ has a cut-free C-proof (I-proof, M-proof). [Gentzen, 1935]

This theorem is proved by permuting cuts upwards through a proof. Consider for example the following two proof fragments.

$$\dfrac{\dfrac{\Delta_1 \longrightarrow B, \Gamma_1 \quad \Delta_2 \longrightarrow C, \Gamma_2}{\Delta_1, \Delta_2 \longrightarrow B \wedge C, \Gamma_1, \Gamma_2} \quad \dfrac{\Delta_3, B \longrightarrow \Gamma_3}{\Delta_3, B \wedge C \longrightarrow \Gamma_3}}{\Delta_1, \Delta_2, \Delta_3 \longrightarrow \Gamma_1, \Gamma_2, \Gamma_3}\ \text{cut}$$

$$\dfrac{\dfrac{\Delta_1 \longrightarrow B, \Gamma_1 \qquad \Delta_3, B \longrightarrow \Gamma_3}{\Delta_1, \Delta_3 \longrightarrow \Gamma_1, \Gamma_3}\ \text{cut}}{\Delta_1, \Delta_2, \Delta_3 \longrightarrow \Gamma_1, \Gamma_2, \Gamma_3}$$

Removing cuts in this way can cause proofs to grow very large (super-exponential).

A Simplified Sequent System

Let Γ and Δ denote sets of formula with Γ, B denoting the set union $\Gamma \cup \{B\}$. Notice that Γ, a matches with the set $\{a, b, c\}$ in two ways: $\Gamma := \{b, c\}$ and $\Gamma := \{a, b, c\}$.

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

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

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

$$\frac{\Delta \longrightarrow \Gamma_1, B \qquad C, \Delta \longrightarrow \Gamma_2}{B \supset C, \Delta \longrightarrow \Gamma_1, \Gamma_2} \supset\text{-L}$$

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

$$\frac{\Delta \longrightarrow \Gamma, \bot}{\Delta \longrightarrow \Gamma, B} \bot\text{-R} \qquad \frac{}{B, \Delta \longrightarrow B, \Gamma} \text{ initial}$$

Natural Deduction

The existence of permutations suggests that sequent proofs contain too much information. More compact representations of proofs are possible.

Classical Logic	expansion trees
Intuitionistic Logic	natural deduction
Linear Logic	proof nets

View the statement $\Delta \longrightarrow B$ as meaning: there exists an argument from the assumptions of Γ to the conclusion B. The notion of argument can be formalized using natural deduction.

Cut-elimination for I-proofs corresponds (roughly) to *normalization* for natural deduction (not studied here).

Natural Deduction (continued)

There are three cut-free proofs of the sequent

$$a, a \supset b, a \supset (b \supset c) \longrightarrow c.$$

All three proofs provide essentially the same argument:

$$\dfrac{\dfrac{g : a \supset (b \supset c) \qquad x : a}{(g\ x) : b \supset c} \qquad \dfrac{f : a \supset b \qquad x : a}{(f\ x) : b}}{(g\ x\ (f\ x)) : c}$$

As we shall see, this natural deduction proof is isomorphic to the simply typed λ-term

$$\lambda x : a\ \lambda f : a \to b\ \lambda g : a \to b \to c.\ (g\ x\ (f\ x)).$$

Lecture 2:

Sequents for Quantificational Logic

and

A Definition of Logic Programming

Implicational Fragment of Intuitionistic Logic

$$\frac{\Gamma \longrightarrow B \qquad C, \Gamma \longrightarrow E}{B \supset C, \Gamma \longrightarrow E} \supset\text{-L}$$

$$\frac{B, \Gamma \longrightarrow C}{\Gamma \longrightarrow B \supset C} \supset\text{-R}$$

$$\frac{}{B, \Delta \longrightarrow B} \text{ initial}$$

The propositional Horn clause

$$a_1 \wedge \ldots \wedge a_n \supset a_0$$

can be written in this fragment of logic as

$$a_1 \supset \ldots \supset a_n \supset a_0.$$

Let p be a propositional letter and let Γ be a set of Horn clauses. Proofs of $\Gamma \longrightarrow p$ contain only the inference rules $\supset$-L and initial.

Simply Typed λ-Terms

This small proof system can be used to give simple types to certain λ-terms. Consider sequents of the form

$$t_1 : \tau_1, \ldots, t_n : \tau_n \longrightarrow t_0 : \tau_0$$

where $t_0, \ldots, t_n$ are λ-terms and $\tau_0, \ldots, \tau_n$ are "types", that is, propositional formulas using only $\supset$ (function type).

$$\frac{\Delta \longrightarrow t : \alpha \qquad ft : \beta, \Delta \longrightarrow s : \gamma}{f : \alpha \supset \beta, \Delta \longrightarrow s : \gamma}$$

$$\frac{x : \alpha, \Delta \longrightarrow t : \beta}{\Delta \longrightarrow \lambda x.t : \alpha \supset \beta}$$

$$\frac{}{\Delta, t : \alpha \longrightarrow t : \alpha}$$

Types for Quantificational Logics

i	individuals (terms)
o	booleans (formulas)
$i \to i \to i$	function of 2 arguments
$(i \to i) \to i$	"functional" of 1 arguments
$i \to o$	predicate of 1 argument
$(i \to o) \to o$	predicate of predicates of 1 argument

Permitting other non-boolean primitive types is straightforward.

The order of a type is the count of the nesting of arrows (implications) to the left.

$$order(i) = 0$$
$$order(i \to i) = 1$$
$$order(i \to i \to i) = 1$$
$$order((i \to i) \to i) = 2$$
$$order(i \to (i \to i) \to i) = 2$$

Signatures

Signatures (or *type assignments*) are finite sets

$$c_1 : \tau_1, \ldots, c_n : \tau_n$$

where $c_1, \ldots, c_n$ are distinct tokens and $\tau_1, \ldots, \tau_n$ are types (propositional formulas over $\rightarrow$).

When a token is declared in a signature, we shall consider it to be a constant.

t is a Σ-*term of type* τ if $\Sigma \longrightarrow t : \tau$ is provable using the proof system for simply typed λ-terms.

If Σ contains only "Horn clause" types and if $\Sigma \longrightarrow t : \alpha$ is provable (for α a primitive type) then t is a first-order term.

Σ-Formulas

Let Σ_0 be the following signature for the logical connectives.

$$\begin{aligned}
\bot &: o \\
\wedge &: o \to o \to o \\
\vee &: o \to o \to o \\
\supset &: o \to o \to o \\
\forall_\tau &: (\tau \to o) \to o \\
\exists_\tau &: (\tau \to o) \to o
\end{aligned}$$

(for all simply types τ not containing o). B is a Σ-*formula* if

$$\Sigma_0, \Sigma \longrightarrow B : o$$

is provable.

Abbreviate $\forall_\tau(\lambda x.B)$ and $\exists_\tau(\lambda x.B)$ as $\forall_\tau x.B$ and $\exists_\tau x.B$, respectively.

Sequents for Quantificational Logic

Let Σ be a signature and let $\Delta \cup \Gamma$ be a finite set of Σ-formulas. The triple

$$\Sigma \; ; \; \Delta \; \longrightarrow \; \Gamma$$

is a sequent for quantificational logic.

The notation $\Sigma + (c \; : \; \tau)$ is meaningful only if Σ does not assign a type to c, in which case it means $\Sigma \cup \{c : \tau\}$.

Take a proof system for propositional logic and attach "Σ ;" to all sequents in it. For example,

$$\frac{\Sigma \; ; \; B, \Delta \; \longrightarrow \; \Gamma \qquad \Sigma \; ; \; C, \Delta \; \longrightarrow \; \Gamma}{\Sigma \; ; \; B \vee C, \Delta \; \longrightarrow \; \Gamma} \; \vee\text{-L}$$

Inference Rules for Quantifiers

$$\frac{\Sigma \longrightarrow t : \tau \qquad \Sigma\ ;\ \Delta, B[t/x] \longrightarrow \Gamma}{\Sigma\ ;\ \Delta, \forall_\tau x\ B \longrightarrow \Gamma}\ \forall\text{-L}$$

$$\frac{\Sigma \longrightarrow t : \tau \qquad \Sigma\ ;\ \Delta \longrightarrow \Gamma, B[t/x]}{\Sigma\ ;\ \Delta \longrightarrow \Gamma, \exists_\tau x\ B}\ \exists\text{-R}$$

$$\frac{\Sigma + c : \tau\ ;\ \Delta, B[c/x] \longrightarrow \Gamma}{\Sigma\ ;\ \Delta, \exists_\tau x\ B \longrightarrow \Gamma}\ \exists\text{-L}$$

$$\frac{\Sigma + c : \tau\ ;\ \Delta \longrightarrow \Gamma, B[c/x]}{\Sigma\ ;\ \Delta \longrightarrow \Gamma, \forall_\tau x\ B}\ \forall\text{-R}$$

Here, c is not declared in Σ. Such an occurrence of c is called an *eigen-variable* of the proof.

$$\frac{\Sigma\ ;\ \Delta \longrightarrow \Gamma, B}{\Sigma\ ;\ \Delta \longrightarrow \Gamma, B'}\ \lambda \qquad \frac{\Sigma\ ;\ B, \Delta \longrightarrow \Gamma}{\Sigma\ ;\ B', \Delta \longrightarrow \Gamma}\ \lambda$$

where B and B' differ only up to α, β, and η conversions. Generally this rule will not be written and we treat formulas in sequents as equivalence classes modulo λ-conversion.

Empty Types?

Generally in first-order logic it is assumed that there alway exist terms over a given signature. This is not necessarily true here. For example, there are no Σ-terms of type i for

$$\Sigma = \{p : i \to o, f : i \to i\}.$$

In a sense, the type i is *empty*.

Notice that the sequent

$$\Sigma' + p : i \to o \; ; \; \forall_i x.px \;\longrightarrow\; \exists_i x.px$$

is provable if and only if there exists a Σ'-term.

In a higher-order setting there are good reasons to consider empty types.
Formalizations of classical logic generally consider only signatures Σ for which there are Σ-terms.

Unification and the Sequent Calculus

Attempting to use free or "logic" variables with unification to delay the selection of substitution terms in building proofs is complicated by the fact that signatures may vary with proofs.
Let $\Sigma = \{a : i, p : i \to o\}$.

$$\dfrac{\Sigma \longrightarrow X : i \qquad \dfrac{\dfrac{\Sigma + b : i \;;\; p\ X \longrightarrow p\ b}{\Sigma + b : i \;;\; \longrightarrow p\ X \supset p\ b}}{\Sigma \;;\; \longrightarrow \forall y(p\ X \supset p\ y)}}{\Sigma \;;\; \longrightarrow \exists_i x \forall_i y(p\ x \supset p\ y)}$$

Here, X denotes of "logical variable" (not a variable of the logic). It is impossible to complete this proof: X must be instantiated to b but b is not a Σ-term.

Logic variables introduced before *eigen-variables* are introduced (in a bottom-up reading) cannot be instantiated with those eigen-variables.

First-Order Horn Clauses are First-Order in Two Senses

First-order Horn clauses can be written as

$$\forall x_1 \ldots \forall x_n (A_1 \wedge \ldots \wedge A_m \supset A_0)$$

where $n \geq 0, m \geq 0$. Here, $A_i's$ are atomic formulas and quantification is over primitive types.
Thus, quantification is *first-order*. Constants range over the types

$$\tau ::= o \mid i \mid i \rightarrow \tau,$$

where o denotes booleans (formulas) and i denotes individuals.
First-order Horn Clauses can also be defined as

$$D ::= A \mid A \supset D \mid \forall x.D.$$

Notice that implications are allowed to be nested to the right but not to the left (just as in τ.)
Thus, logical connective structure is *first-order*.

Logic Programming Considered Abstractly

Programs and *goals* are written using logic syntax.

Computation is the process of "proving" that a given goal follows from a given program.

The notion of "proving" should satisfy at least two properties:

- It should have such meta-theoretic properties as cut-elimination and/or sound and complete model theory. That is, it should be the basis for *declarative programming*.
- The interpretation of logical connectives in goals should have a fixed "search" semantics: that is, the interpretation of logical connectives is independent of context. We shall argue that this is a central feature of *logic programming*.

Our analysis here will be blind to issues of control and unification.

Search Semantics for the Logical Connectives

Let the notation $\Sigma; \mathcal{P} \vdash_O G$ denotes the fact that some idealized interpreter succeeds when given a signature Σ, a program $\mathcal{P}$, and a goal G. The following are intended to fix the interpretation of logical connectives in goal formulas.

- $\Sigma; \mathcal{P} \vdash_O G_1 \wedge G_2$ iff $\Sigma; \mathcal{P} \vdash_O G_1$ and $\Sigma; \mathcal{P} \vdash_O G_2$.
- $\Sigma; \mathcal{P} \vdash_O G_1 \vee G_2$ iff $\Sigma; \mathcal{P} \vdash_O G_1$ or $\Sigma; \mathcal{P} \vdash_O G_2$.
- $\Sigma; \mathcal{P} \vdash_O \exists_\tau x.G$ iff there is a Σ-term t of type τ such that $\Sigma; \mathcal{P} \vdash_O G[t/x]$.
- $\Sigma; \mathcal{P} \vdash_O D \supset G$ iff $\Sigma; \mathcal{P}, D \vdash_O G$.
- $\Sigma; \mathcal{P} \vdash_O \forall_\tau x.G$ iff for any token c not in Σ, $\Sigma + c : \tau; \mathcal{P} \vdash_O G[c/x]$.

Uniform Proofs

Uniform proofs are an attempt to formalize this notion of "search semantics".

A sequent proof Ξ is *uniform* if Ξ is an I-proof and whenever a sequent occurrence in Ξ has a non-atomic righthand side, that sequent occurrence is the conclusion of a right-introduction rule.

In other works, when building proofs bottom-up, do right rules before left rules, and do left rules only when the righthand side is atomic.

The search for uniform proofs is *goal-directed* (*succedent-directed*).

Roughly speaking: A logic can be considered an "abstract logic programming language" if restricting to uniform proofs does not lose completeness.

Abstract Logic Programming Languages

A triple $\langle \mathcal{D}, \mathcal{G}, \vdash \rangle$ is an *abstract logic programming language* (ALPL) if

- $\mathcal{D}$ and $\mathcal{G}$ are sets of formulas
- $\vdash$ is a provability relation using sequents, and
- if Σ is a signature, and $G \in \mathcal{G}$ and $\mathcal{P}$ is a finite subset of $\mathcal{D}$, and $\mathcal{P} \cup \{G\}$ is a set of Σ-formulas, then

  $\Sigma; \mathcal{P} \vdash G$ iff $\Sigma \; ; \; \mathcal{P} \longrightarrow G$ has a uniform proof.

Example: Horn Clauses

- Let $\mathcal{D}_1$ be the set of first-order Horn Clauses.
- Let $\mathcal{G}_1$ be the set of conjunctions of atomic formula.
- Let $\vdash_1$ be either $\vdash_C$, $\vdash_I$, or $\vdash_M$.

Then $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_1 \rangle$ is an ALPL.

This fact can be proved by converting any cut-free proof of $\Sigma \; ; \; \mathcal{P} \longrightarrow G$ into a uniform proof by using enough permutations.

Uniform Proofs Involving Horn Clauses

Let $\mathcal{P}$ be a set of Horn clauses. A uniform proof of $\Sigma\ ;\ \mathcal{P} \longrightarrow G$ never contains a sequent that has an implication or a universal quantifier in the succedent. As a result, all sequents in such a proof have the same signature and the same program.

Thus, in logic programming based on Horn clauses, both the program and the set of constants remain constant during the search for a proof.

This has the advantage that implementations can be relatively static and that unification does not need to be concerned with occurrences of eigenvariables.

The disadvantage is that programs and signatures are global: modular programming and abstract data types are not accounted for in Horn clauses.

Examples of non-ALPLs

The following sequents do not have uniform proofs (signatures are not displayed)

$$p \vee q \longrightarrow q \vee p$$
$$[p(a) \wedge p(b)] \supset q \longrightarrow \exists x(p(x) \supset q)$$
$$p \supset q(a), \neg p \supset q(b) \longrightarrow \exists x.q(x)$$
$$\longrightarrow p \vee (p \supset q)$$

although

$$p \vee q \vdash_M q \vee p$$
$$[p(a) \wedge p(b)] \supset q \vdash_C \exists x(p(x) \supset q)$$
$$p \supset q(a), \neg p \supset q(b) \vdash_C \exists x.q(x)$$
$$\vdash_C p \vee (p \supset q)$$

Lecture 3:

Hereditary Harrop Formulas

and

Their Uses in Programming

Harrop Formulas

A *Harrop Formula* is a formula that has no strictly positive occurrences of $\vee$ and $\exists$.

$$H ::= A \mid B \supset H \mid H_1 \wedge H_2 \mid \forall_\tau x.H$$

where A ranges over atomic formulas and B ranges over arbitrary formulas.

Theorem. Let $\mathcal{P}$ be a finite set of (closed) Harrop formulas. Then all the following hold.

- $\Sigma; \mathcal{P} \vdash_I B_1 \wedge B_2$ iff $\Sigma; \mathcal{P} \vdash_I B_1$ and $\Sigma; \mathcal{P} \vdash_I B_2$.
- $\Sigma; \mathcal{P} \vdash_I \exists_\tau x.B$ iff there is a Σ-term t of type τ such that $\Sigma; \mathcal{P} \vdash_I B[t/x]$.
- $\Sigma; \mathcal{P} \vdash_I B_1 \vee B_2$ iff $\Sigma; \mathcal{P} \vdash_I B_1$ or $\Sigma; \mathcal{P} \vdash_I B_2$.
- $\Sigma; \mathcal{P} \vdash_I B_1 \supset B_2$ iff $\Sigma; \mathcal{P}, B_1 \vdash_I B_2$.
- $\Sigma; \mathcal{P} \vdash_I \forall_\tau x.B$ iff for any token c not in Σ, $\Sigma + c : \tau; \mathcal{P} \vdash_I B[c/x]$.

Thus, proofs involving Harrop formulas are "uniform at the root".

Hereditary Harrop Formulas

Hereditary Harrop formulas have *no* positive occurrences of $\vee$ and $\exists$.

$$G ::= A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists_\tau x.G \mid \forall_\tau x.G \mid D \supset G$$

$$D ::= A \mid G \supset D \mid D_1 \wedge D_2 \mid \forall_\tau x.D$$

Let $\mathcal{D}_2$ be the collection of closed D-formulas and let $\mathcal{G}_2$ be the collection of closed G-formulas.

Theorem. $\langle \mathcal{D}_2, \mathcal{G}_2, \vdash_I \rangle$ and $\langle \mathcal{D}_2, \mathcal{G}_2, \vdash_M \rangle$ are essentially the same abstract logic programming language.

Given some simple equivalences, hereditary Harrop formulas can be simplified to just

$$D ::= A \mid D_1 \wedge D_2 \mid D_1 \supset D_2 \mid \forall_\tau x.D,$$

that is, the set freely generated from $\wedge, \supset, \forall_\tau$. (Even the conjunction can be removed.)

Uniform Proofs Involving Hereditary Harrop Formulas

Let $\mathcal{P}$ be a set of hereditary Harrop Formulas. In a uniform proof of $\Sigma\ ;\ \mathcal{P}\ \longrightarrow\ G$, signatures and programs can increase as the search for a proof continues.

Thus, programs and signatures are not global – they are now like stacks. This supports modular programming and abstract data types. Attempting to prove

$$\mathcal{P} \longrightarrow D_1 \supset (G_1 \wedge (D_2 \supset G_2))$$

will causes the two subgoals

$$\mathcal{P}, D_1 \longrightarrow G_1 \quad \text{and} \quad \mathcal{P}, D_1, D_2 \longrightarrow G_2$$

Implementations are more involved since this language is more dynamic. Unification must be modified to handle eigen-variables. The close-world assumption is no longer valid.

Re-implementing Consult

Let `classify`, `scanner`, `misc` be the name of files containing logic programs.
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`.

Logic programs becomes accessible and disappears in a stack-disciplined fashion.

Importing Modules

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

Here, the modules $\mathbf{M_2}$ and $\mathbf{M_3}$ are parametric modules. That is, they can be identified with logic programs containing free variables.

The import keyword in $\mathbf{M_3}$ is elaborated as follows: For each clause of the form

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

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

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

A Mechanism for Abstract Data Types

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:

$$(\exists x\ D) \supset G \quad \equiv \quad \forall x\ (D \supset G)$$

provided x is not bound in G (otherwise, rename x first).
This is intuitionistically (hence, classically) valid.

Stacks as Abstract Data Types

Let $stack$ and $\exists empty\ \exists stk\ \ stack'$ stand for the following expression:

$$\exists empty\ \exists stk\ [\ emptystack(empty) \wedge \forall s \forall x(push(x, s, stk(x, s))) \wedge \forall s \forall x(pop(x, stk(x, s), s))]$$

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

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

```
module stack.
local empty, stk.
emptystack(empty).
push(X,S,stk(X,S)).
pop(X,stk(X,S),S).
```

The Sterile Jar Problem

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

Read the string "`pi x\`" as $\forall x$.

```
        ?- 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
        ?- heated j, in b j, bug b
        ?- heated j
        ?- in b j
        ?- bug b
```

Meta-Level Properties of $\supset$ and $\forall$ Goals

If M is both a goal formula and a definite clause (contains no occurrences of $\vee$ or $\exists$), then

$$\Sigma; \mathcal{P} \vdash M \text{ and } \Sigma; \mathcal{P} \vdash M \supset G \text{ implies } \Sigma; \mathcal{P} \vdash G.$$

Similarly, if

$$\Sigma; \mathcal{P} \vdash \forall_\tau x.G \quad \text{and } t \text{ is a } \Sigma\text{-term of type } \tau,$$

then $\Sigma; \mathcal{P} \vdash G[t/x]$. Here, $G[t/x]$ denotes the λ-normal form of substituting t for x in G. In particular, if new λ-redexes are formed by this substitution, these are also removed.

These results follow from the *cut-elimination* theorem or from appropriate model-theoretic semantics.

For example, if it is provable that `g` is a bug in sterile jar `j`, then it is provable that `g` is dead.

Kripke Models for Propositional Formulas

Consider propositional formulas over just the logical constants $\wedge$ and $\supset$. Let $\langle \mathcal{W}, \leq \rangle$ be a partially ordered set. A Kripke model over $\langle \mathcal{W}, \leq \rangle$ is a mapping K from $\mathcal{W}$ to sets of atomic formulas such that

$$\forall w_1, w_2 \in \mathcal{W}(w_1 \leq w_2 \supset K(w_1) \subseteq K(w_2)).$$

Satisfaction in a Kripke model is defined by induction of the structure of formulas.

- $K, w \models A$ if A is atomic and $A \in K(w)$.
- $K, w \models B_1 \wedge B_2$ if $K, w \models B_1$ and $K, w \models B_2$.
- $K, w \models B_1 \supset B_2$ if forall $w' \in \mathcal{W}$ such that $w \leq w'$, if $K, w' \models B_1$ then $K, w' \models B_2$.

The condition for truth of an implication is strong: not only must the implication be true in the current world w but also in all worlds "above" it.

A Kripke Model as a Canonical Model

When attempting to prove a goal from a program, larger programs may need to be considered. Thus, to assign a meaning to one logic program suggests that meaning must also be assigned simultaneously to all larger programs.

Kripke models provide an ideal setting for assigning meaning.

Let $\mathcal{W}$ be the set of all finite sets of formulas (propositional, over $\wedge$ and $\supset$). Use inclusion $\subseteq$ as the order relation.
Define K_0 as: $K_0(w) = \{A \text{ atomic } | w \vdash A\}$.
Theorem:

Cut-elimination holds for this propositional logic

if and only if

forall $w \in \mathcal{W}$ and formula B, B has a cut-free proof from w if and only if $K_0, w \models B$.

Kripke Models for First-Order Logic

Assign to every world a set of individuals using a function D from worlds to sets of terms so that

$$\forall w_1, w_2 \in \mathcal{W}(w_1 \leq w_2 \supset D(w_1) \subseteq D(w_2)).$$

Add the following rule for satisfaction.

- $K, w \models \forall x.B$ if forall $w' \in \mathcal{W}$ such that $w \leq w'$, if $t \in D(w')$ then $K, w' \models B[t/x]$.

A canonical model for the logic containing $\wedge$, $\supset$, and $\forall$ can built in the following fashion:

- $\mathcal{W}$ is the set of pairs of the form $\langle \Sigma, w \rangle$ where Σ is a signature and w is a finite set of Σ-formulas.
- $\langle \Sigma, w \rangle \leq \langle \Sigma', w' \rangle$ if $\Sigma \subseteq \Sigma'$ and $w \subseteq w'$.
- $D_0(\langle \Sigma, w \rangle)$ is the set of all Σ-terms.
- $K_0(\langle \Sigma, w \rangle)$ is the set of all atomic A such that $\Sigma, w \vdash A$.

A similar theorem to that on the preceding slide can be proved here.

A Need for λ-Terms in a Logic Program

Consider proving the goal

$$\texttt{?-}\ \forall L\ (append\ (cons\ a\ nil)\ L\ V) \wedge (g\ V).$$

This reduces to first trying

$$\texttt{?-}\ append\ (cons\ a\ nil)\ l\ V$$

where l is new. This is solvable if and only if V unifies with $(cons\ a\ l)$ which is not possible for all the following reasons:

- the constant l is introduced after (in a bottom-up reading) the free variable V;
- l is not permitted to leave its scope;
- the value of V should be independent of the choice of constant used to instantiate $\forall L$.

One can argue, however, that an interesting value has been computed and that the current language is too weak to capture it. That is, the language is really not *closed*.

λ-Abstraction as Discharging of Scoped Constants

Instead, consider the query

$$\texttt{?-}\ \forall L\ (append\ (cons\ a\ nil)\ L\ (V\ L)) \wedge (g\ V).$$

Here, V is a variable of higher-type. This reduces to first trying

$$\texttt{?-}\ append\ (cons\ a\ nil)\ l\ (V\ l)$$

where l is new. This is solvable if and only if

$$(V\ l) \text{ unifies } (cons\ a\ l)$$

which is possible. Of the two possible unifiers

$$V = \lambda w(cons\ a\ w) \text{ and}$$
$$V = \lambda w(cons\ a\ l)$$

only the first is legal.
Notice the first solution is essentially the result of *discharging* the scoped constant l from the term $(cons\ a\ l)$.

Lecture 4:

Computing with λ-Terms

and

Meta-Programming

Structure of First-Order Terms

Let's first review the structure of first-order terms.

$$\Sigma = \{a : i, \quad b : i, \quad f : i \to i, \quad g : i \to i \to i\}$$

$$\frac{\Sigma \vdash X : i}{\Sigma \vdash f\ X : i} \qquad \frac{\Sigma \vdash X : i \qquad \Sigma \vdash Y : i}{\Sigma \vdash g\ X\ Y : i}$$

$$\frac{}{\Sigma \vdash a : i} \qquad \frac{}{\Sigma \vdash b : i}$$

Notice that signatures do not change in these inference rules.

$$\frac{\dfrac{\dfrac{}{\Sigma \vdash a : i}}{\Sigma \vdash f\ a : i} \qquad \dfrac{}{\Sigma \vdash b : i}}{\Sigma \vdash g\ (f\ a)\ b : i}$$

Structure of λ-Terms

$$\Sigma' = \Sigma \cup \{h : (i \to i) \to i\}$$

$$\frac{\Gamma \vdash U : i \to i}{\Gamma \vdash h\ U : i} \qquad \frac{\Gamma,\ x : i \vdash V : i}{\Gamma \vdash \lambda x.V : i \to i}$$

provided that Γ is an extension of Σ' and x is not in Γ.

$$\dfrac{\dfrac{\dfrac{\Sigma',\ x : i \vdash x : i \qquad \dfrac{\Sigma',\ x : i \vdash x : i}{\Sigma',\ x : i \vdash f\ x : i}}{\Sigma',\ x : i \vdash g\ x\ (f\ x) : i}}{\Sigma' \vdash \lambda x.g\ x\ (f\ x) : i \to i}}{\Sigma' \vdash h\ (\lambda x.g\ x\ (f\ x)) : i}$$

Designing a New Notion of Abstract Syntax

First: Recursion over terms with abstraction requires signatures (contexts) to be dynamically augmented.

Second: Equality of terms should be at least α-conversion.

Since terms are not freely generated, simple destructuring is not a sensible operation.

$$\lambda x(fxx) \quad = \quad \lambda y(fyy)$$

$$x \quad (fxx) \quad \neq \quad y \quad (fyy)$$

This, of course, suggests unification modulo α-conversion.

Unification Modulo $\alpha\beta_0$-Conversion

$$\begin{array}{ll} \forall : (i \to b) \to b & r : i \to b \\ \wedge : b \to b \to b & s : i \to b \\ \supset : b \to b \to b & t : b \end{array}$$

$$\forall \lambda x(P \wedge Q) = \forall \lambda y((ry \supset sy) \wedge t)$$

This pair has no unifiers (modulo α-conversion).

$$\forall \lambda x(Px \wedge Q) = \forall \lambda y((ry \supset sy) \wedge t)$$

This pair has one unifier:

$$\{P \mapsto \lambda w(rw \supset sw), Q \mapsto t\}$$

provided a wee bit of β-conversion is permitted.

$$\forall \lambda x([\lambda w(rw \supset sw)x] \wedge t) = \forall \lambda y((ry \supset sy) \wedge t)$$

$$(\lambda x.B)x = B \qquad \beta_0\text{-conversion}$$

Some Matching Examples

$a : i \qquad f : i \to i \qquad g : i \to i \to i$

(1)	$\lambda x \lambda y (f\ (H\ x))$	$\lambda u \lambda v (f\ (f\ u))$
(2)	$\lambda x \lambda y (f\ (H\ x))$	$\lambda u \lambda v (f\ (f\ v))$
(3)	$\lambda x \lambda y (g\ (H\ y\ x)\ (f\ (L\ x)))$	$\lambda u \lambda v (g\ u\ (f\ u))$
(4)	$\lambda x \lambda y (g\ (H\ x)\ (L\ x))$	$\lambda u \lambda v (g\ (g\ a\ u)\ (g\ u\ u))$

(1)	$H \mapsto \lambda w (f\ w)$	
(2)	match failure	
(3)	$H \mapsto \lambda y \lambda x.x$	$L \mapsto \lambda x.x$
(4)	$H \mapsto \lambda x.(g\ a\ x)$	$L \mapsto \lambda x.(g\ x\ x)$

L_λ-Unification

An L_λ-*unification problem* is a finite set of equations between simply typed λ-terms such that occurrences of free variables of higher-order type are applied to distinct λ-bound variables. (Dropping this restriction yields *higher-order unification.*)

Properties

L_λ-unification is decidable and most general unifiers exist if unifiers exist.

L_λ-unification appears to be the simplest extension to first-order unification that "respects" bound variables.

L_λ-unification does not require type information to determine unifiers or the possibility of unifiers.

$\beta\eta$-unification of simply typed λ-terms (sometimes called "higher-order" unification) can be encoded directly as logic programming using only L_λ-unification.

L_λ in a Logic Programming Language

To incorporate λ-terms into a logic programming language we must take (at least) three steps.

- Perform L_λ-unification.
- Permit universal quantification and implications in goals. This permits for dynamically changing contexts.
- Modify the restriction: higher-order variables can be applied to at most distinct variables that are λ-bound or universally quantified negatively. For example,

  $$\forall_i X \forall_{i \to i} F(\forall_i y(p\ y \supset p\ \lambda w(F\ w\ y)) \supset q\ F\ X)$$

  is legal while the following is not legal.

  $$\forall_i X \forall_{i \to i} F(\forall_i y(p\ y \supset p\ \lambda w(F\ w\ X)) \supset q\ F\ X)$$

The Signature of a First-Order Object-Logic

```
kind  term, form  type.

type  all, some   (term -> form) -> form.
type  and, imp    form -> form -> form.

type  a     term.
type  f     term -> term.
type  g     term -> term -> term.
type  p     term -> form.
type  q     term -> term -> form.

type term   term -> o.
type atom   form -> o.

term a.
term (f X)   :- term X.
term (g X Y) :- term X, term Y.
atom (p X)   :- term X.
atom (q X Y) :- term X, term Y.
```

Recognizing Object-Level Horn Clauses

```
type quanfree, conj, hornc   form -> o.

quanfree A :- atom A.
quanfree (and B C) :- quanfree B, quanfree C.
quanfree (imp B C) :- quanfree B, quanfree C.

conj (and B C) :- conj B, conj C.
conj A :- atom A.

hornc A :- atom A.
hornc (imp A G) :- atom A, conj G.
hornc (all C) :- pi x\(term x => hornc (C x)).

?- hornc (all u\(all v\(imp (p u)
                           (and (q v a) (q a u)))))

{C = u\(all v\(imp (p u)(and (q v a)(q a u))))}

term d ?- hornc (all v\(imp (p d)
                            (and (q v a) (q a d))))

{C = v\(imp (p d) (and (q v a) (q a d)))}

term e ?- hornc (imp (p d)
                         (and (q e a) (q a d))))
```

Implementing Object-Level Equality

```
type   copytm    term -> term -> o.
type   copyfm    form -> form -> o.

copytm a a.
copytm (f X) (f U) :- copytm X U.
copytm (g X Y) (g U V) :-
                          copytm X U, copytm Y V.

copyfm (p X) (p U) :- copytm X U.
copyfm (q X Y) (q U V) :-
                          copytm X U, copytm Y V.
copyfm (and X Y) (and U V) :-
                          copyfm X U, copyfm Y V.
copyfm (imp X Y) (imp U V) :-
                          copyfm X U, copyfm Y V.
copyfm (all X) (all U) :-
 pi y\(pi z\(copytm y z => copyfm (X y)(U z))).
copyfm (some X) (some U) s:-
 pi y\(pi z\(copytm y z => copyfm (X y)(U z))).
```

$$[\![t, s : \texttt{term}]\!] = \texttt{copytm}\ t\ s$$

$$[\![t, s : \texttt{form}]\!] = \texttt{copyfm}\ t\ s$$

$$[\![t, s : \tau \texttt{ -> } \sigma]\!] = \forall x \forall y ([\![x, y : \tau]\!] \supset [\![t\ x, s\ y : \sigma]\!])$$

Implementing Object-Level Substitution

```
type subst  (term -> form) -> term -> form -> o.

subst M T N :-
      pi c\(copytm c T => copyfm (M c) N).
```

Here, the first argument of `subst` is an abstraction over formulas. Compare this to the somewhat simpler specification (which is not in L_λ):

```
subst M T (M T).
```

```
type uni_instan  form -> term -> form -> o.

uni_instan (all B) T C :- subst B T C.
```

Using meta-level β-conversion:

```
uni_instan (all B) T (B T).
```

Partial Correctness of `hornc` and `subst`

Theorem. Instantiating a Horn clause with a term results in a Horn clause.

Proof. Assume $\vdash$ `term t` and $\vdash$ `hornc (all w\h)`. Thus,

$\vdash$ `pi x\(term x => hornc h[x/w])`

$\vdash$ `term t => hornc h[t/w]`

$\vdash$ `hornc h[t/w]`

Theorem. If $\vdash$ `subst (w\d) t s` then `s` is `d[t/w]`.

Proof. First note that $\vdash$ `copytm u v` if and only if `u` and `v` are equal terms.

$\vdash$ `subst (w\d) t s`

$\vdash$ `pi x\(copytm x t => copytm d[x/w] s)`

$\vdash$ `copytm t t => copytm d[t/w] s`

$\vdash$ `copytm d[t/w] s`

The converse of this theorem is also easy to prove.

Reversing Substitutions

```
subst F a (q a a)
```

This query yields four answer substitutions for `F`:

```
w\(q w w)   w\(q w a)   w\(q a w)   w\(q a a).
```

```
copytm a a.
copyfm (q X Y)(q U V) :- copytm X U,copytm Y V.
```

```
?- subst F a (q a a).
?- pi c\(copytm c a => copyfm (F c) (q a a)).
copytm c a.  ?- copyfm (F c) (q a a).
     {F c = (q (F1 c) (F2 c))}
copytm c a ?- copytm (F1 c) a, copytm (F2 c) a.
copytm c a ?- copytm (F1 c) a.
     {F1 c = c}   or   {F1 c = a}
      F1 = w\w    or    F1 = w\a
```

Much of full higher-order unification can be understood as reversing substitution in this manner.

Additional Examples

The following programs make use of meta-level β-conversion to do object-level substitution.

```
type double (term -> term) -> term -> term -> o.
double F X (F (F X)).

type mapfun (term -> term) ->
            term list -> term list -> o.

mapfun F nil nil.
mapfun F (cons X L) (cons (F X) K) :-
   mapfun F L K.
```

To make substitution explicit, write instead:

```
type substterm  (term -> term) ->
                  term -> term -> o.
substterm M T N :-
   pi c\(copytm c T => copytm (M c) N).

double F X S :-
   substterm F X T, substterm F T S.
mapfun F (cons X L) (cons T K) :-
   substterm F X T, mapfun F L K.
```

Interpreting Object-Level Horn Clauses

```
type  interp    list form -> form -> o.
type  instan    form -> form -> o.
type  backchain list form -> form -> form -> o.

interp Cs (and B C) :- interp Cs B, interp Cs C.
interp Cs A :- atom A, memb D Cs,
               instan D E, backchain Cs E A.

instan (all A) B :-
           pi x\(copytm x T => instan (A x) B).
instan B C :- quanfree B, copyfm B C.

backchain Cs A A.
backchain Cs (imp A G) A :- interp Cs G.
```

Higher-Order Programming

If the restrictions on higher-order variables in L_λ are removed, then an implementation of the resulting logic requires full meta-level β-conversion and higher-order unification. Some forms of predicate variable quantification can also be permitted. This provides for higher-order programming.

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

The terms "higher-order Horn clauses" and "higher-order hereditary Harrop" formulas permit full meta-level β-conversion and predicate variables quantification.

Lecture 5:

A Brief Introduction to Linear Logic

and a

Linear Refinement of

Hereditary Harrop Formulas

Removing Contraction and Weakening

The two presentations of $\wedge$-L:

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

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

are equivalent rules in the presence of contraction and weakening. If we remove these structural rules, then the notion of conjunction splits into two different connectives:

- $\&$ additive conjunction ("with")
- $\otimes$ multiplicative conjunction ("tensor")

Similarly, disjunction splits into two different connectives

- $\oplus$ additive disjunction ("plus")
- $\sqcup$ multiplicative disjunction ("par")

Linear implication, written as $\multimap$, is treated as a multiplicative connective.

A Proof Systems for Linear Logic

(without the exponentials)

$$\frac{\Delta' \longrightarrow \Gamma', B \qquad \Delta'' \longrightarrow \Gamma'', C}{\Delta', \Delta'' \longrightarrow \Gamma', \Gamma'', B \otimes C} \qquad \frac{B, C, \Delta \longrightarrow \Gamma}{B \otimes C, \Delta \longrightarrow \Gamma}$$

$$\frac{\Delta \longrightarrow \Gamma, B \qquad \Delta \longrightarrow \Gamma, C}{\Delta \longrightarrow \Gamma, B \,\&\, C}$$

$$\frac{B, \Delta \longrightarrow \Gamma}{B \,\&\, C, \Delta \longrightarrow \Gamma} \qquad \frac{C, \Delta \longrightarrow \Gamma}{B \,\&\, C, \Delta \longrightarrow \Gamma}$$

$$\frac{B, \Delta' \longrightarrow \Gamma' \qquad C, \Delta'' \longrightarrow \Gamma''}{B \sqcup C, \Delta', \Delta'' \longrightarrow \Gamma', \Gamma''} \qquad \frac{\Delta \longrightarrow B, C, \Gamma}{\Delta \longrightarrow B \sqcup C, \Gamma}$$

$$\frac{B, \Delta \longrightarrow \Gamma \qquad C, \Delta \longrightarrow \Gamma}{B \oplus C, \Delta \longrightarrow \Gamma}$$

$$\frac{\Delta \longrightarrow \Gamma, C}{\Delta \longrightarrow \Gamma, B \oplus C} \qquad \frac{\Delta \longrightarrow \Gamma, B}{\Delta \longrightarrow \Gamma, B \oplus C}$$

A Proof Systems for Linear Logic (continued)

$$\frac{\Delta' \longrightarrow \Gamma', B \qquad C, \Delta'' \longrightarrow \Gamma''}{B \multimap C, \Delta', \Delta'' \longrightarrow \Gamma', \Gamma''}$$

$$\frac{B, \Delta \longrightarrow \Gamma, C}{\Delta \longrightarrow \Gamma, B \multimap C}$$

$$\frac{}{B \longrightarrow B} \qquad \frac{}{0, \Delta \longrightarrow \Gamma} \qquad \frac{}{\Delta \longrightarrow \top, \Gamma}$$

Here, Δ and Γ are multisets of propositional formulas.

Only a fragment of propositional linear logic is presented here. In particular, the exponentials and linear negation, $(-)^{\perp}$, are not yet addressed.

Examples

$$\dfrac{\dfrac{\dfrac{}{a \longrightarrow a} \qquad \dfrac{}{a \longrightarrow a}}{a \longrightarrow a \,\&\, a}}{\longrightarrow a \multimap a \,\&\, a}$$

$$\dfrac{\dfrac{}{a \longrightarrow a} \qquad \dfrac{\dfrac{}{b \longrightarrow b} \qquad \dfrac{}{c \longrightarrow c}}{b, b \multimap c \longrightarrow c}}{a \multimap b, b \multimap c, a \longrightarrow c}$$

$$\dfrac{\dfrac{}{a \longrightarrow a} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{b \longrightarrow b} \qquad \dfrac{}{d \longrightarrow d}}{b, b \multimap d \longrightarrow d} \qquad \dfrac{}{c \longrightarrow c}}{b \sqcup c, b \multimap d \longrightarrow d, c}}{b \sqcup c, b \multimap d \longrightarrow d \sqcup c}}{a \multimap (b \sqcup c), b \multimap d, a \longrightarrow d \sqcup c}$$

There are no proofs of

$$a \multimap (a \otimes a) \qquad\qquad (a \otimes b) \multimap (a \,\&\, b)$$

or their converses.

The Exponentials ! and ?

Without contraction and weakening, linear logic would would be very weak. Contraction and weakening are introduced via logical connectives and not by structural rules.

- ! "of course"
- ? "why not"

$$\frac{B, \Delta \longrightarrow \Gamma}{!B, \Delta \longrightarrow \Gamma} \qquad \frac{\Delta \longrightarrow \Gamma}{!B, \Delta \longrightarrow \Gamma} \qquad \frac{!B, !B, \Delta \longrightarrow \Gamma}{!B, \Delta \longrightarrow \Gamma}$$

$$\frac{\Delta \longrightarrow B, \Gamma}{\Delta \longrightarrow ?B, \Gamma} \qquad \frac{\Delta \longrightarrow \Gamma}{\Delta \longrightarrow ?B, \Gamma} \qquad \frac{\Delta \longrightarrow ?B, ?B, \Gamma}{\Delta \longrightarrow ?B, \Gamma}$$

$$\frac{!\Delta \longrightarrow B, ?\Gamma}{!\Delta \longrightarrow !B, ?\Gamma} \qquad \frac{B, !\Delta \longrightarrow ?\Gamma}{?B, !\Delta \longrightarrow ?\Gamma}$$

The notion of intuitionistic implication $B \supset C$ is coded as

$$!B \multimap C.$$

Examples

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a \longrightarrow a}}{a \,\&\, b \longrightarrow a}}{!(a \,\&\, b) \longrightarrow a}}{!(a \,\&\, b) \longrightarrow !\, a} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{b \longrightarrow b}}{a \,\&\, b \longrightarrow b}}{!(a \,\&\, b) \longrightarrow b}}{!(a \,\&\, b) \longrightarrow !\, b}$$

$$\dfrac{!(a \,\&\, b), !(a \,\&\, b) \longrightarrow !\, a \otimes !\, b}{!(a \,\&\, b) \longrightarrow !\, a \otimes !\, b}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a \longrightarrow a}}{!\, a \longrightarrow a}}{!\, a, !\, b \longrightarrow a} \qquad \dfrac{\dfrac{\dfrac{}{b \longrightarrow b}}{!\, b \longrightarrow b}}{!\, a, !\, b \longrightarrow b}}{!\, a, !\, b \longrightarrow a \,\&\, b}$$

$$\dfrac{!\, a, !\, b \longrightarrow a \,\&\, b}{!\, a, !\, b \longrightarrow !(a \,\&\, b)}$$

$$\dfrac{!\, a, !\, b \longrightarrow !(a \,\&\, b)}{!\, a \otimes !\, b \longrightarrow !(a \,\&\, b)}$$

Set $1 := !\top$ and $\bot := ?0$. Then $a \otimes 1$ is linearly equivalent to a and $a \sqcup \bot$ is linearly equivalent to a.

Set $B^{\bot} := B \multimap \bot$.

Aspects of Intuitionistic Contexts

Theorem Proving

+ Contexts manage hypotheses and eigenvariables elegantly.
− Contraction cannot be controlled naturally.

Linguistics

+ Relative clauses are sentences with noun phrase gaps: $(NP \supset SENT) \supset REL$.
− Gap extraction must be non-vacuous.

Data Bases

+ Contexts can act as databases and support query answering by deduction.
− Contexts cannot naturally be "edited" or updated.

Object State

+ Objects can have their state and methods hidden in a context.
− Updating an object's state is not possible declaratively.

A Linear Refinement of Contexts

In intuitionistic contexts, all formulas can be used any number of times. To make a linear refinement, replace

$$!D_1, \ldots, !D_n \longrightarrow G$$

with the more general

$$R_1, \ldots, R_m, !D_1, \ldots, !D_n \longrightarrow G.$$

Now, there will be multiplicative and additive versions of some of the logical connectives.
If $\otimes$ and ! are permitted to occur freely, many provable sequents would not have uniform proofs.
For example, the sequents

$$a \otimes b \longrightarrow b \otimes a \qquad a, a \multimap !\, b \longrightarrow !\, b$$

are provable in linear logic but do not have uniform proofs.

A Sublanguage for Logic Programming

Consider the set of formulas freely generated from

$$\top, \&, \multimap, \Rightarrow, \forall$$

where $B \Rightarrow C$ is $!\, B \multimap C$.

$$\frac{}{\Gamma; A \longrightarrow A} \qquad \frac{\Gamma, B; \Delta, B \longrightarrow C}{\Gamma, B; \Delta \longrightarrow C} \qquad \frac{}{\Gamma; \Delta \longrightarrow \top}$$

$$\frac{\Gamma; \Delta, B_i \longrightarrow C}{\Gamma; \Delta, B_1 \,\&\, B_2 \longrightarrow C} \qquad \frac{\Gamma; \Delta \longrightarrow B \quad \Gamma; \Delta \longrightarrow C}{\Gamma; \Delta \longrightarrow B \,\&\, C}$$

$$\frac{\Gamma; \Delta_1 \longrightarrow B \quad \Gamma; \Delta_2, C \longrightarrow E}{\Gamma; \Delta_1, \Delta_2, B \multimap C \longrightarrow E} \qquad \frac{\Gamma; \Delta, B \longrightarrow C}{\Gamma; \Delta \longrightarrow B \multimap C}$$

$$\frac{\Gamma; \emptyset \longrightarrow B \quad \Gamma; \Delta, C \longrightarrow E}{\Gamma; \Delta, B \Rightarrow C \longrightarrow E} \qquad \frac{\Gamma, B; \Delta \longrightarrow C}{\Gamma; \Delta \longrightarrow B \Rightarrow C}$$

Introducing Positive Occurrences of $\otimes, \oplus, \mathbf{1}, \top$

$$\dfrac{\Gamma\ ;\ \Delta_1 \longrightarrow P \qquad \dfrac{\Gamma\ ;\ \Delta_2 \longrightarrow Q \qquad \dfrac{}{\Gamma\ ;\ R \longrightarrow R}}{\Gamma\ ;\ \Delta_2, Q \multimap R \longrightarrow R}}{\Gamma\ ;\ \Delta_1, \Delta_2, P \multimap Q \multimap R \longrightarrow R}$$

This suggests that tensors $\otimes$ in goals can be specified using the following higher-order clause.

$$\forall P \forall Q [P \multimap Q \multimap (P \otimes Q)]$$

Similarly, other logical constants can be specified in goal positions.

$$\forall P \forall Q [P \multimap (P \oplus Q)]$$

$$\forall P \forall Q [Q \multimap (P \oplus Q)]$$

$$\top \Rightarrow \mathbf{1}$$

$$\forall P [P \Rightarrow ! P]$$

Embedding Hereditary Harrop Formulas

Girard has presented a mapping of intuitionistic logic into linear logic, part of which is given as:

$$(A)^0 = A, \text{ where } A \text{ is atomic,}$$
$$(\text{true})^0 = \mathbf{1},$$
$$(B_1 \wedge B_2)^0 = (B_1)^0 \,\&\, (B_2)^0,$$
$$(B_1 \supset B_2)^0 = !(B_1)^0 \multimap (B_2)^0$$

A "tighter" translation holds in our setting.

$$(A)^+ = (A)^- = A, \text{ where } A \text{ is atomic}$$
$$(\text{true})^+ = \mathbf{1} \qquad (\text{true})^- = \top$$
$$(B_1 \wedge B_2)^+ = (B_1)^+ \otimes (B_2)^+$$
$$(B_1 \wedge B_2)^- = (B_1)^- \,\&\, (B_2)^-$$
$$(B_1 \supset B_2)^+ = (B_1)^- \Rightarrow (B_2)^+$$
$$(B_1 \supset B_2)^- = (B_1)^+ \multimap (B_2)^-$$

Thus, we should translate `a :- b, c => d.` as

$$[b \otimes (c \Rightarrow d)] \multimap a.$$

How to Toggle a Switch

Let Γ contain the following clauses.

```
toggle(G) :- sw(V), flip(V,U), sw(U) -o G.
flip(on,off).
flip(off,on).
```

$$\dfrac{\dfrac{\Gamma\ ;\ sw(off) \longrightarrow sw(V) \quad \Gamma\ ;\ \longrightarrow flip(V,U) \quad \dfrac{\Gamma\ ;\ \Delta, sw(U) \longrightarrow G}{\Gamma\ ;\ \Delta \longrightarrow sw(U) \multimap G}}{\Gamma\ ;\ \Delta, sw(off) \longrightarrow sw(V) \otimes flip(V,U) \otimes sw(U) \multimap G}}{\Gamma\ ;\ \Delta, sw(off) \longrightarrow toggle(G)}$$

The Modality of !

When ! appears in a goal, it behaves as a modal operator. Consider the following proof fragment.

$$\dfrac{\dfrac{\dfrac{\Gamma\ ;\ D\ \longrightarrow\ G_1}{\Gamma\ ;\ \emptyset\ \longrightarrow\ D \multimap G_1}}{\Gamma\ ;\ \emptyset\ \longrightarrow\ !(D \multimap G_1)} \qquad \Gamma\ ;\ \Delta\ \longrightarrow\ G_2}{\Gamma\ ;\ \Delta\ \longrightarrow\ !(D \multimap G_1) \otimes G_2}$$

Contrast this to the proof fragment involving the demo-predicate.

$$\dfrac{D\ \longrightarrow\ G_1 \qquad \mathcal{P}\ \longrightarrow\ G_2}{\mathcal{P}\ \longrightarrow\ demo(D, G_1) \wedge G_2}$$

Improving a Theorem

Below is a theorem prover for a propositional intuitionistic object-logic. Here, `erase` is concrete syntax for $\top$.

```
pv (A and B) :- pv B & pv A.
pv (A imp B) :- hyp A -o pv B.
pv (A or  B) :- pv A.
pv (A or  B) :- pv B.
pv G :- hyp (A and B),
        (hyp A -o hyp B -o pv G).
pv G :- hyp (A or  B),
        ((hyp A -o pv G) & (hyp B -o pv G)).
pv G :- hyp (C imp B),
        ((hyp (C imp B) -o pv C) &
         (hyp B -o pv G)).
pv G :- hyp false, erase.
pv G :- hyp G, erase.
```

Permuting a List

```
load nil K      :- unload K.
load (X::L) K   :- (item X -o load L K).
unload nil.
unload (X::L)   :- item X, unload L.

perm L K :-  load L K.
```

Notice that the last clause defining `perm` is not correct enough. There is nothing in its definition that guarantees that when it is called there are no `items` in the bounded context.
A better definition is

```
perm L K  <=  load L K.
```

or (using the "defined" logical constant `bang`)

```
perm L K  :-  bang(load L K).
```

Parsing Relative Clauses

Correct: "whom Mary married ↑"

Correct: "whom Mary believed John married ↑"

Wrong: "whom Mary married Bill" because the gap is not used: vacuous abstraction

Wrong: "whom Mary believed ↑ married Jill" because subject extraction is not permitted here.

```
sent P1 P2    o- !(np P1 P0) x vp P0 P2.
vp P1 P2      o- tv P1 P0 x np P0 P2.
vp P1 P2      o- stv P1 P0 x sbar P0 P2.
np P1 P2      o- pn P1 P2.
sbar (that::P1) P2 o- sent P1 P2.

rel (whom::P1) P2 o-
                  all z\(np z z) -o sent P1 P2.

pn (mary::L) L.
pn (bob::L) L.
pn (jill::L) L.
tv (loves::L) L.
tv (married::L) L.
stv (believes::L) L.
```

How to Split Bounded Contexts?

$$\frac{\Gamma\ ;\ \Delta_1 \longrightarrow B \quad \Gamma\ ;\ \Delta_2, C \longrightarrow E}{\Gamma\ ;\ \Delta_1, \Delta_2, B \multimap C \longrightarrow E}$$

$$\frac{\Gamma\ ;\ \Delta_1 \longrightarrow B \qquad \Gamma\ ;\ \Delta_2 \longrightarrow C}{\Gamma\ ;\ \Delta_1, \Delta_2 \longrightarrow B \otimes C}$$

If $\Delta = \Delta_1, \Delta_2$ contains n items, there are 2^n ways to form the partitions Δ_1 and Δ_2. How can we delay partitioning during search?

An Interpreter for the Propositional Fragment

An *IO-context* is a list made up of formulas, !'ed formulas, or the special symbol del used to denote a place where a formula has been deleted.

$I\{G\}O$: given resources I, a proof of G can be built that returns the resources in O.

If $!R$ is an element of an IO-context, it is never removed. If R (without a !) is an element, it can be replaced by del.

pickR(I, O, R) holds if R is a member of I and O is the result of replacing that occurrence of R with del; or $!\,R$ occurs in I, and I and O are equal.

subtensor(I, O): if O arises from replacing zero or more non-!'ed components of I with del.

An Input/Output Interpreter

$$\frac{}{I\{1\}I} \qquad \frac{\text{subcontext}(O, I)}{I\{\top\}O} \qquad \frac{I\{G\}I}{I\{!\,G\}I}$$

$$\frac{I\{G_1\}M \qquad M\{G_2\}O}{I\{G_1 \otimes G_2\}O} \qquad \frac{I\{G_1\}O \qquad I\{G_2\}O}{I\{G_1 \,\&\, G_2\}O}$$

$$\frac{R{::}I\{G\}\text{del}{::}O}{I\{R \multimap G\}O} \qquad \frac{!\,R{::}I\{G\}\,!\,R{::}O}{I\{R \Rightarrow G\}O}$$

$$\frac{\text{pickR}(I, O, A)}{I\{A\}O} \qquad \frac{\text{pickR}(I, M, G \multimap A) \qquad M\{G\}O}{I\{A\}O}$$

$$\frac{\text{pickR}(I, O, G \Rightarrow A) \qquad O\{G\}O}{I\{A\}O}$$

Notice that these are all first-order Horn clauses.

A Prolog implementation is immediate.

Resource Indexed Models

Let $\langle R, +, 0\rangle$ be a commutative monoid: the *monoid of bounded resources.*

Let $\langle \mathcal{W}, \leq\rangle$ be a partially ordered set: the *set of possible worlds.*

A *(propositional) Kripke interpretation* is an order preserving mapping from $\langle \mathcal{W}, \leq\rangle$ to the powerset of the set atomic formulas.

A *resource indexed model* $\mathcal{M}$ is an R-indexed set of Kripke interpretations, $\{K_r \mid r \in R\}$.

Satisfaction in a structure $\mathcal{M} = \{K_r \mid r \in R\}$ is defined by:

- $K_r, w \models \top$.
- $K_r, w \models A$ if A is atomic and $A \in K_r(w)$.
- $K_r, w \models B_1 \;\&\; B_2$ if $K_r, w \models B_1$ and $K_r, w \models B_2$.
- $K_r, w \models B_1 \multimap B_2$ if $\forall r' \in R, \forall w' \in \mathcal{W}$ if $w \leq w'$ and $K_{r'}, w' \models B_1$ then $K_{r+r'}, w' \models B_2$.
- $K_r, w \models B_1 \Rightarrow B_2$ if $\forall w' \in \mathcal{W}$ if $w \leq w'$ and $K_0, w' \models B_1$ then $K_r, w' \models B_2$.

A Canonical Model Theorem

$\mathcal{M} \models B$ if $\forall w \in \mathcal{W}$, $K_0, w \models B$.
The Kripke interpretation K_0 models truth in the usual intuitionistic sense while K_r models truth that has been moved out-of-phase.

Theorem

Let $\mathcal{W}$ be the set of all finite sets of formulas (over $\top$, &, $\multimap$, and $\Rightarrow$) and let $\leq$ be set inclusion.
Let R be the set of all finite multisets of such formulas and let $+$ be multiset union and 0 be the empty multiset.
Define $\mathcal{M} = \{K_r \mid r \in R\}$ by

$$K_r(w) = \{A \mid A \text{ is atomic and } w; r \vdash_{\mathcal{L}} A\}.$$

Then the equivalence

$$\forall w \in \mathcal{W}\, \forall r \in R(w; r \vdash_{\mathcal{L}} B \text{ iff } K_r, w \models B)$$

holds if and only if the cut rules are admissible in $\mathcal{L}$.