Linear Logic

Dale Miller
Penn State University
In September:
INRIA Futurs/Saclay and Ecole Polytechnique

Outline

1. Roles of logic in specifying computation
2. Focus on proof search (logic programming)
3. Two linear logic programming languages
4. Various examples and reasoning techniques
5. Specification of security protocols

Roles of Logic in the Specification of Computation

In the specification of computational systems, logics are generally used in one of two approaches.

Computation-as-model: Here, computations are mathematical structures representing computations via nodes, transitions, and state (for example, Turing machines, etc). Logic is used in an external sense to make statements *about* those structures. E.g. Hoare triples, BAN logic.

Computation-as-deduction: Here, logical deduction is used to model computation directly.

Functional programming. Programs are proof and computation is modeled using proof normalization (λ-conversion, cut-elimination).

Logic programming. Programs are theories and computation is the search for (cut-free) sequent proofs. For example, we encode the fact that the identifier x has value 5 as the atomic formula $(x\ 5)$ or the fact that agent Alice has memory M as $(a\ M)$ (both x and a are predicates). The dynamics of computation are encoded in the changes to sequents that occur during the search for a proof. Linear logic support rich changes in sequents during proof search.

Observations

Logic programming has proved more expressive than function programming in handling syntax with bound variables (solved in LP for 15 years, FP is still working on it). Logical frameworks, used to encode logical systems, are examples of logic programming languages.

If you add more logical connectives (say, LL connectives), in FP you don't get new programs but more types. In LP you get more programs. In particular, LL has made a big impact on logic programming design: each new connective is a new combinator for new programs.

Adding logical connectives *modularly* enriches an LP's expressiveness. For example, the interplay between higher-orders and modules is not affected by adding linear resources/concurrency. To get the meaning, just read off the proof theory.

Cut-elimination provides for a formal basis for justifying this modularity.

Kowalski's equation revisited

$$\text{Algorithm} = \text{Logic} + \text{Control}.$$

This equation makes the important point that there is a gap between first-order Horn clause specifications and algorithmic specifications. Unfortunately, this equation has been elaborated into:

$$\begin{aligned} \text{Programming} = \text{Logic} \quad &+ \quad \text{Control} + \text{I/O} + \\ &+ \quad \text{Higher-order programming} \\ &+ \quad \text{Data abstractions} \\ &+ \quad \text{Modules} \\ &+ \quad \text{Concurrency} + \ldots \end{aligned}$$

Such extensions are generally *ad hoc*: logic, which was the motivation and the intriguing starting point, is now put in a minor ghetto. Questions about how various features interact start to dominate the language design. If static analysis is done on purely logical expressions, it can be done deeply and richly. On the mess above, it is severely restricted or impossible.

A goal of declarative programming

A more interesting project would be to get closer to the goal:

$$\text{Programming} = \text{Logic}.$$

If this equation is at all possible, then one will certainly need to rethink what is meant by "Programming" and by "Logic".

In these lectures, we explore an interpretation of "Logic" that makes use of elements of higher-order logic and linear logic.

Three Logics

Classical Logic: A logic for "truth." Think of truth tables or models *a la* Tarski. Truth is not dynamic: it is fixed.

$$\vdash p \vee \neg p$$

Intuitionistic Logic: A logic of proof and construction. Think of type theory or the λ-calculus.

$$\not\vdash p \vee \neg p$$

Linear Logic: A logic of resources. Think of multiset rewriting, vending machines, etc.

- Two quarter can become a cup of coffee and a dime.
- A process can become its continuation and a network message.

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:

1. It should have some deep meta-theoretical properties such as cut-elimination and/or sound and complete model theory. That is, it should be the basis for declarative programming.

2. The interpretation of logical connectives in goals should have a fixed “search” semantics: that is, the interpretation of logical connectives is independent of context. This interpretation of logical connectives is a central feature of logic programming.

Because of this latter properties, logic programming is sometimes referred to as called proof search.

Goal Directed Search

To construct a proof of the sequent $\Delta \longrightarrow G$, where G is not atomic, use the right introduction rule.

$$\frac{\Delta \longrightarrow G_1 \quad \Delta \longrightarrow G_2}{\Delta \longrightarrow G_1 \mathbin{\&} G_2} \qquad \frac{\Delta, D \longrightarrow G}{\Delta \longrightarrow D \Rightarrow G}$$

A logic makes a abstract logic programming language if such a simple search strategy is complete.

With a multiple-conclusion sequent setting, we would like to generalize this to be simultaneous right introductions.

$$\frac{\Delta, D \longrightarrow G_1, G_3, \Gamma \qquad\qquad \Delta, D \longrightarrow G_2, G_3, \Gamma}{\Delta \longrightarrow G_1 \mathbin{\&} G_2, D \Rightarrow G_3, \Gamma}$$

If enough introduction rules permute, you can "simulate" simultaneous introduction.

Zoo of linear logic connectives

$\multimap$	linear implication (lollipop)
$!$	reuse modal (bang)
$?$	dual modal (why not)
$\Rightarrow$	intuitionistic implication, $A \Rightarrow B \equiv {!}\,A \multimap B$
$\&$	additive conjunctions (with)
$\otimes$	multiplicative conjunctions (tensor)
$\oplus$	additive disjunctions
$\parr$	multiplicative disjunctions (par)
$\top$	additive truth: $\top \mathbin{\&} A \equiv A$
$\mathbf{1}$	multiplicative truth: $\mathbf{1} \otimes A \equiv A$
0	additive false: $0 \oplus A \equiv A$
$\bot$	multiplicative false: $\bot \parr A \equiv A$
$(-)^\bot$	negation
$\forall, \exists$	quantifiers

A Series of Logic Programming Languages

Horn clauses (Prolog) Unrestricted use of $\{\&, \top\}$ but $\forall, \Rightarrow$ are restricted to the top-level only. For example, $\forall \bar{x}(G \Rightarrow A)$.

Hereditary Harrop formulas (λProlog) Unrestricted use of $\{\forall, \Rightarrow, \&, \top\}$. For example, $\forall x((Bx \mathbin{\&} \forall y(Cxy \Rightarrow Dy)) \Rightarrow Dx)$.

Lolli (a linear refinement of λProlog) Unrestricted use of $\{\forall, \multimap, \Rightarrow, \&, \top\}$. For example, $\forall x((Bx \multimap \forall y(Cxy \Rightarrow Dy)) \Rightarrow Dx)$.

Linear Objects (LO) Unrestricted use of $\{\&, \top, \parr, \bot\}$ with only top-level occurrences of $\forall$, $\multimap$.

$$\forall \bar{x}(G \multimap A_1 \parr \cdots \parr A_n)(n \geq 1)$$

Forum Unrestricted use of $\{\forall, \multimap, \Rightarrow, \&, \top, \parr, \bot\}$. For example,

$$\forall x(Bx \multimap \forall y(Cxy \Rightarrow Dy) \Rightarrow Dx \parr Bx)$$

Forum is a presentation of all of linear logic

The linear logic connectives missing from Forum are definable.

$$B^\perp \equiv B \multimap \perp \qquad 0 \equiv \top \multimap \perp \qquad 1 \equiv \perp \multimap \perp$$

$$!\,B \equiv (B \Rightarrow \perp) \multimap \perp \qquad B \oplus C \equiv (B^\perp \mathbin{\&} C^\perp)^\perp \qquad B \otimes C \equiv (B^\perp \mathbin{⅋} C^\perp)^\perp$$

$$\exists x.B \equiv (\forall x.B^\perp)^\perp$$

The collection of connectives in Forum are not minimal. For example, $?$ and $⅋$, can be defined in terms of the remaining connectives.

$$?\,B \equiv (B \multimap \perp) \Rightarrow \perp \quad \text{and} \quad B \mathbin{⅋} C \equiv (B \multimap \perp) \multimap C$$

A proof system for intuitionistic logic

A sequent $\Gamma \longrightarrow C$ contains a set of formulas Γ and a single formula C.

$$\frac{}{\Gamma, B \longrightarrow B}\ \mathit{initial} \qquad \frac{}{\Gamma \longrightarrow \top}\ \top R$$

$$\frac{\Gamma, B_1 \longrightarrow C}{\Gamma, B_1 \,\&\, B_2 \longrightarrow C}\ \&L \qquad \frac{\Gamma, B_2 \longrightarrow C}{\Gamma, B_1 \,\&\, B_2 \longrightarrow C}\ \&L \qquad \frac{\Gamma \longrightarrow B \qquad \Gamma \longrightarrow C}{\Gamma \longrightarrow B \,\&\, C}\ \&R$$

$$\frac{\Gamma \longrightarrow B \qquad \Gamma, C \longrightarrow E}{\Gamma, B \Rightarrow C \longrightarrow E}\ \Rightarrow L \qquad \frac{\Gamma, B \longrightarrow C}{\Gamma \longrightarrow B \Rightarrow C}\ \Rightarrow R$$

$$\frac{\Gamma, B[t/x] \longrightarrow C}{\Gamma, \forall x.B \longrightarrow C}\ \forall L \qquad \frac{\Gamma \longrightarrow B[y/x]}{\Gamma \longrightarrow \forall x.B}\ \forall R,$$

provided that y is not free in the lower sequent.

$$\frac{\Gamma' \longrightarrow B \qquad \Gamma, B \longrightarrow C}{\Gamma' \longrightarrow C}\ \mathit{cut}, \quad \text{provided } \Gamma \subseteq \Gamma'.$$

Given that left-hand contexts are sets, if the pattern matches Γ, B, it might be the case that $B \in \Gamma$.

Backchaining as left-introduction rules

$$\dfrac{\dfrac{\Gamma \longrightarrow B \qquad \Gamma \longrightarrow C}{\Gamma \longrightarrow B \mathbin{\&} C}\;\&R \qquad \dfrac{}{\Gamma, A \longrightarrow A}\;\mathit{initial}}{\Gamma, B \mathbin{\&} C \Rightarrow A \longrightarrow A}\;\Rightarrow L$$

$$\dfrac{\Gamma \longrightarrow B \qquad \dfrac{\Gamma \longrightarrow C \qquad \dfrac{}{\Gamma, A \longrightarrow A}\;\mathit{initial}}{\Gamma, C \Rightarrow A \longrightarrow A}\;\Rightarrow L}{\Gamma, B \Rightarrow C \Rightarrow A \longrightarrow A}\;\Rightarrow L$$

Of course, $B \Rightarrow (C \Rightarrow A) \equiv (B \mathbin{\&} C) \Rightarrow A$ is the familar curry/uncurry equivalence. The formula $(B \mathbin{\&} C) \Rightarrow A$ is written in Prolog as

```
A :- B, C.
```

Expressive strength: changes in context

Consider a cut-free proof of the sequent $\Gamma \longrightarrow A$. Let $\Gamma' \longrightarrow A'$ be a sequent somewhere in this proof. (Assume that A and A' are atomic formulas.)

In general, $\Gamma \subseteq \Gamma'$. Contexts can only grow as one moves up through a proof.

If Γ is a set of Horn clauses, then, in fact, $\Gamma' = \Gamma$. Thus, proof search with Horn clauses is basically flat and does not allow abstractions (eg, modules).

The atoms A and A' can be related arbitrarily. Thus, most of the dynamics of a computation must be captured within atoms; that is, within non-logical contexts. Thus, the dynamics is out of the reach of logical principles (modus ponens, cut-elimination, etc).

Linear logic will allow much greater ability to code dynamics within logical contexts.

Proof system for a fragment of linear logic

$$\frac{}{B \longrightarrow B}\ initial \qquad \frac{}{\Delta \longrightarrow \top}\ \top R$$

$$\frac{\Delta, B_i \longrightarrow C}{\Delta, B_1 \,\&\, B_2 \longrightarrow C}\ \&L\ (i = 1, 2) \qquad \frac{\Delta \longrightarrow B \qquad \Delta \longrightarrow C}{\Delta \longrightarrow B \,\&\, C}\ \&R$$

$$\frac{\Delta_1 \longrightarrow B \qquad \Delta_2, C \longrightarrow E}{\Delta_1, \Delta_2, B \multimap C \longrightarrow E}\ \multimap L \qquad \frac{\Delta, B \longrightarrow C}{\Delta \longrightarrow B \multimap C}\ \multimap R$$

$$\frac{\Delta, B_1, B_2 \longrightarrow C}{\Delta, B_1 \otimes B_2 \longrightarrow C}\ \otimes L \qquad \frac{\Delta_1 \longrightarrow B \qquad \Delta_2 \longrightarrow C}{\Delta_1, \Delta_2 \longrightarrow B \otimes C}\ \otimes R$$

$$\frac{\Delta \longrightarrow C}{\Delta, !B \longrightarrow C}\ !W \qquad \frac{\Delta, !B, !B \longrightarrow C}{\Delta, !B \longrightarrow C}\ !C \qquad \frac{\Delta, B \longrightarrow C}{\Delta, !B \longrightarrow C}\ !D \qquad \frac{!\Delta \longrightarrow B}{!\Delta \longrightarrow !B}\ !R$$

$$\frac{\Delta, B[t/x] \longrightarrow C}{\Delta, \forall x.B \longrightarrow C}\ \forall L \qquad \frac{\Delta \longrightarrow B[y/x]}{\Delta \longrightarrow \forall x.B}\ \forall R,$$

provided that y is not free in the lower sequent.

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

Goal-directed search in linear logic

The notion of "goal-directed proofs" can be formalized using the following technical notion: A cut-free proof is *uniform* if every sequent with a non-atomic right-hand side is the conclusion of a right-introducion rule.

In intuitionistic logic, the following are provable but not with uniform proofs:

$$p \vee q \Rightarrow q \vee p \qquad\qquad \exists x.px \Rightarrow \exists x.px$$

$$q \mathbin{\&} (q \Rightarrow (r \vee s)) \Rightarrow (r \vee s)$$

In linear logic, the following are provable but not with uniform proofs:

$$p \otimes q \multimap q \otimes p \qquad\qquad !\,p \multimap\, !\,p$$

$$q \multimap (q \multimap (r \otimes s)) \multimap (r \otimes s)$$

For these reasons, $\vee$, $\exists$, $\otimes$, and ! are not generally allowed unrestricted in logic programming languages.

Proof system for Lolli

$$\frac{}{\Gamma; A \longrightarrow A}\ \mathit{initial} \qquad \frac{\Gamma, B; \Delta, B \longrightarrow C}{\Gamma, B; \Delta \longrightarrow C}\ \mathit{absorb} \qquad \frac{}{\Gamma; \Delta \longrightarrow \top}\ \top R$$

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

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

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

$$\frac{\Gamma; \Delta, B[t/x] \longrightarrow C}{\Gamma; \Delta, \forall x.B \longrightarrow C}\ \forall L \qquad \frac{\Gamma; \Delta \longrightarrow B[y/x]}{\Gamma; \Delta \longrightarrow \forall x.B}\ \forall R$$

provided that y is not free in the lower sequent.

$$\frac{\Gamma'; \Delta_1 \longrightarrow B \qquad \Gamma; \Delta_2, B \longrightarrow C}{\Gamma'; \Delta_1, \Delta_2 \longrightarrow C}\ \mathit{cut} \qquad \frac{\Gamma'; \emptyset \longrightarrow B \qquad \Gamma, B; \Delta \longrightarrow C}{\Gamma'; \Delta \longrightarrow C}\ \mathit{cut}!$$

Two forms of the cut rule for $\mathcal{L}$. Both rules have the proviso that $\Gamma \subseteq \Gamma'$.

How to Toggle a Switch

Occurrences of $\otimes$-R can be allows (as with $\exists$-R, $\vee$-R, etc).

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

Consider the Lolli (Forum) program:

$toggle\ G \circ\!\!- \exists u \exists v[sw\ v \otimes flip\ v\ u \otimes (sw\ u \multimap G)].$
$flip\ on\ \ off.$
$flip\ off\ on\ .$

Assume that these three clauses are members of Ψ.

$$\dfrac{\dfrac{\dfrac{}{\Psi; sw\ on \longrightarrow sw\ on} \quad \dfrac{\dfrac{}{\Psi; flip\ on\ \ off \longrightarrow flip\ on\ \ off}}{\Psi; \longrightarrow flip\ on\ \ off} \quad \dfrac{\dfrac{\vdots}{\Psi; \Delta, sw\ off \longrightarrow G}}{\Psi; \Delta \longrightarrow sw\ off \multimap G}}{\dfrac{\Psi; \Delta, sw\ on \longrightarrow sw\ on \otimes flip\ on\ \ off \otimes (sw\ off \multimap G)}{\Psi; \Delta, sw\ on \longrightarrow \exists v \exists u[sw\ v \otimes flip\ v\ u \otimes (sw\ u \multimap G)]}}}{\Psi; \Delta, sw\ on \longrightarrow toggle\ G}$$

Count up or down?

Here are two specifications of a counter object, which includes encapsulated state and two methods.

$$\begin{aligned} E_1 = \exists r[\quad & (r\ 0) \otimes \\ & !\forall K \forall V (get\ V\ K \circ\!\!- r\ V \otimes (r\ V \multimap K)) \otimes \\ & !\forall K \forall V (inc\ V\ K \circ\!\!- r\ V \otimes (r\ (V+1) \multimap K))] \end{aligned}$$

$$\begin{aligned} E_2 = \exists r[\quad & (r\ 0) \otimes \\ & !\forall K \forall V (get\ (-V)\ K \circ\!\!- r\ V \otimes (r\ V \multimap K)) \otimes \\ & !\forall K \forall V (inc\ (-V)\ K \circ\!\!- r\ V \otimes (r\ (V-1) \multimap K))] \end{aligned}$$

These specification are encoded using continuation passing style: the variable K ranges over continuations.

If we write E_1 as $\exists r(R_1 \otimes !\, R_2 \otimes !\, R_3)$, then using simple "curry/uncurry" equivalences in linear logic can rewrite $E_1 \multimap G$ as the equivalent formula $\forall r(R_1 \multimap R_2 \Rightarrow R_3 \Rightarrow G)$.

Are these counters equivalent?

Observationally equivalent? I.e., proves the same atomic formulas?

Trace equivalent? I.e., calls to the methods of one object's implementation correspond to calls to the methods of the other object's implementation.

In fact, these specifications are *logically* equivalent. The entailments $E_1 \vdash E_2$ and $E_2 \vdash E_1$ are provable in linear logic.

The proof of each of these entailments proceeds (in a bottom-up fashion) by choosing an eigen-variable, say s, to instantiate the existential quantifier on the left-hand specification and then instantiating the right-hand existential quantifier with

$$\lambda x.s\ (-x).$$

The proof of these entailments must also use the equations

$$\{-0 = 0, -(x+1) = -x - 1, -(x-1) = -x + 1\}.$$

Reversing a list in Prolog

Move one item from top of one list to the top of the other list.

```
(1::2::3::nil)     nil.
(2::3::nil)   (1::nil).
(3::nil)   (2::1::nil).
nil     (3::2::1::nil).
```

This can be encoded as the program

```
rv nil (3::2::1::nil).
rv (X::L) M :- rv L (X::M).
```

and the query

```
rv (1::2::3::nil) nil.
```

Not really a good program since it is written for one list only.

Notice that `reverse` is symmetric. Proof: Flip both rows and columns!

A better specification

Put the previously written code into "local block" definitions within another definition. The abstract out `(1::2::3::nil)` and `(3::2::1::nil)` for variables.

$$\forall L, K[$$

$$(\forall rv\ ((\forall M, N, X(rv\ N\ (X :: M) \multimap rv\ (X :: N)\ M)) \Rightarrow rv\ nil\ K \multimap rv\ L\ nil))$$

$$\multimap reverse\ L\ K]$$

The base case is assumed linearly! An attempt to prove

$$reverse\ (1 :: 2 :: 3 :: nil)\ (3 :: 2 :: 1 :: nil)$$

results in the introduction of a new predicate rv and the attempt to prove that from the two clauses

$$rv\ nil\ (3 :: 2 :: 1 :: nil)$$

$$!\forall M, N, X(rv\ N\ (X :: M) \multimap rv\ (X :: N)\ M)$$

it follows that

$$rv\ (1 :: 2 :: 3 :: nil)\ nil.$$

Reverse is symmetric

Theorem. Reverse is symmetric; that is, if $\vdash reverse\ L\ K$ then $\vdash reverse\ K\ L$.

Proof. Assume that $\vdash reverse\ L\ K$. Thus, the body of *reverse*'s clause must be provable.

$$\vdash \forall rv\ ((\forall M, N, X (rv\ N\ (X :: M) \multimap rv\ (X :: N)\ M)) \Rightarrow rv\ nil\ K \multimap rv\ L\ nil)$$

Now, instantiate this quantifier with the term $\lambda x \lambda y.(rv\ y\ x)^{\perp}$, and we have that

$$\vdash (\forall M, N, X ((rv\ (X :: M)\ N)^{\perp} \multimap (rv\ M\ (X :: N))^{\perp}) \Rightarrow (rv\ K\ nil)^{\perp} \multimap (rv\ nil\ L)^{\perp}$$

Using the linear logic equivalence of the contrapositive rule: $p^{\perp} \multimap q^{\perp} \equiv q \multimap p$, we have

$$\vdash (\forall M, N, X (rv\ M\ (X :: N) \multimap rv\ (X :: M)\ N)) \Rightarrow rv\ nil\ L \multimap rv\ K\ nil$$

By universal generalization over rv, we have

$$\vdash \forall rv\ ((\forall M, N, X (rv\ M\ (X :: N) \Rightarrow rv\ (X :: M)\ N)) \Rightarrow rv\ nil\ L \multimap rv\ K\ nil)$$

This matches the body of the reverse problem if we switch `L` and `K`. Thus, we can conclude that $\vdash reverse\ K\ L$.

A brief list of references

- Joshua Hodas and Dale Miller, *Logic programming in a fragment of intuitionistic linear logic*, **Information and Computation**, vol. 110 (1994), no. 2, pp. 327–365.
- Max Kanovich, *The complexity of Horn fragments of linear logic*, **Annals of Pure and Applied Logic**, vol. 69 (1994), pp. 195–241.
- Dale Miller, *Forum: A multiple-conclusion specification language*, **Theoretical Computer Science**, vol. 165 (1996), no. 1, pp. 201–232.
- Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov, *Uniform proofs as a foundation for logic programming*, **Annals of Pure and Applied Logic**, vol. 51 (1991), pp. 125–157.