Lecture 1: Using logic to specify computation

Dale Miller
Penn State University

September 2000

Abstract

Computational systems are often given their meaning in terms of operational semantics and these, in turn, can be formalized in meta-languages based on logics. Recent work has shown that higher-order intuitionistic and linear logics can play important roles in providing high-level and declarative specifications of operational semantics. Higher-order logic can be used to provide for a declarative treatment of binding operators within syntax using lambda-tree syntax or higher-order abstract syntax. Intuitionistic and linear logics allow for elegant methods for specifying computations situated within contexts, such as typing context, state, and concurrent computation threads.

One use of logical specifications is to provide prototype implementations of the specified systems via a general purpose interpreter for the logical system. Another use for such specifications is to form the mathematical basis for deriving formal consequences of that system's behavior: for example, proving that a given evaluation strategy is determinate and type safe or proving that a certain bisimulation is a congruence.

In these lectures we will present a formal logical system that allows one to reason about other logical systems. In particular, we describe an enrichment of intuitionistic logic that allows for the use of induction and of definitions (a proof theoretic approach to the closed world assumption). The use of various object-logics (classical, intuitionistic, linear) and the need to do induction in the presence of binding within syntax are all novelties of this approach. We show how these novelties impact on the development of an actual theorem prover, IRIS, currently being developed at Penn State. Many examples of reasoning within this system will be presented.

Joint work with Raymond McDowell and Jeremie Wajs.

Some roles of logic in computational specifications

- Computation-as-model
  - Computations are seen as mathematical models
  - Logic is used externally to state properties about computations.
- Computation-as-deduction
  - Logic formulas and proofs are used to represent computable values. The dynamics of these proofs model computation. There are two common choices.
  - *Proof reduction.* A program is seen as a (non-normal) proof of its type and computation involves reduction to normal form. Functional programming can be characterized this way.
  - *Proof search.* State is modeled by the structure of formulas in a sequent and computation is modeled by the dynamics of proof search. Logic programming can be characterized this way.

Inference rules as specifications

Specification of meaning in computational systems often make use of inference rules to describe judgments about such systems. For example: the definition of Standard ML, the average paper in a conference or journal on programming language design, semantics, or analysis, etc. Structured operational semantics; natural semantics.

As a concrete example, consider the following four inference rules to describe a dynamic semantics (evaluation) and static semantics (typing) for the untyped λ-calculus.

$$\frac{M \Downarrow \lambda x.R \qquad R[N/x] \Downarrow V}{M\,N \Downarrow V} \qquad \frac{}{\lambda x.R \Downarrow \lambda x.R}$$

$$\frac{\Gamma \vdash M\colon A \to B \qquad \Gamma \vdash N\colon A}{\Gamma \vdash M\,N\colon B} \qquad \frac{\Gamma, x\colon A \vdash R\colon B}{\Gamma \vdash \lambda x.R\colon A \to B}$$

Proviso 1: $R[N/x]$ is not a constructor of the system but intended to mean the result of substituting U for x in R.

Proviso 2: The last rule is restricted so that x is not declared in Γ.

Logic formulas as specifications

Of course, inference rules have been used to specify *logics* as well: for example, sequent calculus, natural deduction, or axiomatic systems.

Considerable recent research has gone into showing that once one defines suitable logics (such as intuitionistic or linear logic), the logic can be used to encode directly other inference rules. For example:

$$\forall M, N, V, U, R\ [\mathit{eval}\ \ M\ (\lambda x.R) \wedge \mathit{eval}\ \ (R[N/x])\ V \supset \mathit{eval}\ \ (M\ \ N)\ V]$$

$$\forall R\ [\mathit{eval}\ \ (\lambda x.R)\ (\lambda x.R)]$$

$$\forall \Gamma, M, N, A, B[(\Gamma \vdash \mathit{typeof}\, M\ (A \rightarrow B)) \wedge (\Gamma \vdash \mathit{typeof}\, N\ \ A) \supset (\Gamma \vdash \mathit{typeof}\,(M\ N)\ B)]$$

$$\forall \Gamma, R, A, B\ [(\Gamma, x{:}\, A \vdash \mathit{typeof}\, R\ B) \supset (\Gamma \vdash \mathit{typeof}\,(\lambda x.R)\ (A \rightarrow B))]$$

While this kind of encoding is obvious, natural, and immediate, it is also wrong, or at least it is if we considered these formulas as Horn clauses (either in first-order or higher-order logic).

Getting syntax of expressions right

It is common to encode object-level application as meta-level applications.

Object-level abstractions (bindings) are generally solved with various "tricks": string names, de Bruijn numerals, pointers, etc. Bind is complicated and deserves a more declarative treatment.

Bindings occur in expressions (e.g., $\lambda x.R$) and in judgments (e.g., $\Gamma, x{:}\, A \vdash \mathit{typeof}\ R\ B$).

We shall use a technique called *λ-tree syntax* representation or *higher-order abstract syntax*.

1. Meta-level abstractions is used to represent object-level abstractions.
2. Eigenvariable abstractions (proof-level abstractions) are used to represent abstractions in judgments.

Example: Coding untyped λ-terms

To denote (untyped) λ-terms and simple types, we introduce two (meta-level) types, *tm* and *ty*.

Untyped λ-terms are encoded using the constants *abs* of type $(tm \to tm) \to tm$ and *app* of type $tm \to tm \to tm$ for object-level abstraction and application.

$\lambda x.x$	$abs\ \ \lambda x.x$
$\lambda x.(xx)$	$abs\ \ \lambda x\ (app\ x\ x)$
$\lambda x \lambda y.x$	$abs\ \lambda x\ abs\ \ \lambda y.x$
$\lambda x \lambda y.(yx)$	$abs\ \lambda x\ abs\ \ \lambda y\ (app\ y\ x)$

Using such a coding places α-equivalence classes of object-level terms in one-to-one correspondence with $\beta\eta$-equivalence classes in the meta-level.

Object-level types will be built up from a single primitive type using the arrow type constructor; these are denoted at the meta-level by the constants *int*, *real*, *string*, ... of type *ty* and *arr* of type $ty \to ty \to ty$.

A logic specification using meta-level abstractions

$$\forall M, N, V, U, R\ [\mathit{eval}\ \ M\ (\mathit{abs}\ R) \wedge \mathit{eval}\ \ (R\ N)\ V \supset \mathit{eval}\ \ (\mathit{app}\ M\ \ N)\ V]$$

$$\forall R\ [\mathit{eval}\ \ (\mathit{abs}\ R)\ (\mathit{abs}\ R)]$$

$$\forall M, N, A, B\ [\mathit{typeof}\, M\ (A \to B) \wedge \mathit{typeof}\, N\ \ A \supset \mathit{typeof}\,(\mathit{app}\ M\ \ N)\ B]$$

$$\forall R, A, B\ [\forall x[\mathit{typeof}\, x\ A \supset \mathit{typeof}\,(R\ x)\ B] \supset \mathit{typeof}\,(\mathit{abs}\ R)\ (A \to B)]$$

Bindings in expressions are encoded as bindings in the logic's terms and bindings in judgments are encoded using bindings in proofs (eigenvariables).

The first three clauses are Horn clauses; the fourth is not.

Why use an expressive logic in specifications?

Clearly we have moved aspects of objects abstractions to the logic. There are several advantages to doing so.

1. When someone implements the logic correctly, that implementation can be used for a number of object-level specifications.
2. There are a number of quality implementations of logics based on intuitionistic logic: the Teyjus implementation of λProlog, Isabelle, and the Twelf implementation of LF.
3. Properties of the logic (such as cut-elimination) should help in proving properties of object-level specifications.
4. The logic provides an elegant and declarative treatment of syntax (no tricks).

There are some problems with using such a logic. One we focus on in this talk: how does one formally reason about logic specifications? Algebraic techniques do not directly apply. Naive application of induction does not work.

Choices of logics for specifications

If we use simply typed λ-terms and certain higher-order quantification in our meta-logic, then λ-tree syntax can be exploited, mostly independently from the choice of logical quantification.

We shall use the *proof-search-as-computation* paradigm (logic programming) to specify computations. There has been considerable study during the past two decades on what logics support this style of specification naturally.

Classical logic Horn clauses, the weakest of logic programming languages (foundation of Prolog), can be interpreted classically.

Intuitionistic logic Hereditary Harrop formulas, enrich Horn clauses with hypothetical and generic judgments. These are the foundation of λProlog.

Linear logic *Lolli* is a subset of linear logic than extends hereditary Harrop formulas with linear implication. *Forum* is a presentation of *all* of linear logic that turns out to be logic programming (at least abstractly). Current research focuses on how to make practical programming languages based on these linear logic programming languages.

Sequent calculus in 4 slides

A *sequent* is a triple written as $\Sigma\colon \Delta \longrightarrow \Gamma$ where

Σ: The *signature* of this sequent: a set of type declarations for constants and variables used in the formulas in Δ and Γ.

Δ: The *antecedent* or *left-hand side*.

Γ: The *succedent* or *right-hand side*.

In classical logic, Δ and Γ are generally sets of formulas.

In minimal logic, Δ is a set and Γ a singleton set.

Intuitionistic logic uses minimal logic sequents, except that Γ can also be empty (denoting false).

In linear logic, Δ and Γ are generally multisets of formulas (multiplicity of formulas matter).

Sequent calculus inference rules

Inference rules are divided into three kinds.

Structural rules Often sequents are present where the antecedent and succedent are *lists*. Then the *exchange* rules is used to reorder formula; the *contraction* rule is used to reduce multiplicities; and the *weakening rule* is used to insert extra formulas. By choosing antecedent and succedent as sets, as in intuitionistic logic, we can avoid these rules here.

Cut and Initial rules We present the version of these rules for intuitionistic logic.

$$\frac{}{\Sigma : \Delta, B \longrightarrow B}\ \mathit{initial} \qquad \frac{\Sigma : \Delta_1 \longrightarrow B \quad \Sigma : \Delta_2, B \longrightarrow C}{\Sigma : \Delta_1, \Delta_2 \longrightarrow C}\ \mathit{cut}$$

The rules in both of these categories do not involve logical connectives directly.

Introduction rules Logical connectives are introduced into the right context by *right-introduction rules* and into the left context by *left-introduction rules.*

$$\frac{\Sigma : \Delta \longrightarrow C}{\Sigma : \Delta, \top \longrightarrow C}\ \top\mathcal{L} \qquad \frac{}{\Sigma : \Delta \longrightarrow \top}\ \top\mathcal{R}$$

$$\frac{\Sigma : \Delta, B_i \longrightarrow C}{\Sigma : \Delta, B_1 \wedge B_2 \longrightarrow C}\ \wedge\mathcal{L} \qquad \frac{\Sigma : \Delta \longrightarrow B_1 \qquad \Sigma : \Delta \longrightarrow B_2}{\Sigma : \Delta \longrightarrow B_1 \wedge B_2}\ \wedge\mathcal{R}$$

$$\frac{\Sigma : \Delta \longrightarrow B_1 \qquad \Sigma : \Delta, B_2 \longrightarrow C}{\Sigma : \Delta, B_1 \supset B_2 \longrightarrow C}\ \supset\mathcal{L} \qquad \frac{\Sigma : \Delta, B_1 \longrightarrow B_2}{\Sigma : \Delta \longrightarrow B_1 \supset B_2}\ \supset\mathcal{R}$$

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

Provisos: In $\forall\mathcal{L}$, $\Sigma \vdash t : \tau$ and in $\forall\mathcal{R}]$, y is not declared in Σ.

Following Church, logical expressions will have type o. We restrict quantification also so that the type τ does not contain o (*i.e.*, no predicate quantification).

Backchaining as a derived rule

$$\dfrac{\dfrac{\dfrac{}{p \longrightarrow p}\ initial \quad \dfrac{}{q \longrightarrow q}\ initial}{q, p \longrightarrow p \wedge q}\ \wedge\mathcal{R} \qquad \dfrac{}{r \longrightarrow r}\ initial}{\dfrac{(p \wedge q) \supset r, q, p \longrightarrow r}{(p \wedge q) \supset r, q \longrightarrow p \supset r}\ \supset\mathcal{R}}\ \supset\mathcal{L}$$

The following useful inference is derivable from left-introductions:

$$\frac{\mathcal{P} \longrightarrow G}{\mathcal{P}, G \supset A \longrightarrow A}\ BC$$

This can be generalized with universal quantifiers.

$$\frac{\mathcal{P} \longrightarrow \theta G}{\mathcal{P}, \forall \bar{x}(G \supset A') \longrightarrow A}\ BC$$

provided that $\theta A' = A$.

Example: path in a graph

adj a b.

adj b c.

adj b d.

adj c d.

$\forall x \forall y(adj\ x\ y \supset path\ x\ y).$

$\forall x \forall y \forall z(adj\ x\ y \wedge path\ y\ z \supset path\ x\ z).$

If written as a program in (lambda) Prolog:

```
adj a b.
adj b c.
adj b d.
adj c d.
path X Y :- adj X Y.
path X Z :- adj X Y, path Y Z.
```

Example: searching a path

Let $\mathcal{P}$ the the set of formulas on the previous slide. Proof search can be used to search for a path in a graph.

$$\dfrac{\dfrac{\dfrac{}{\mathcal{P} \longrightarrow \mathit{adj}\ a\ b}\ \mathit{initial} \qquad \dfrac{\vdots}{\mathcal{P} \longrightarrow \mathit{path}\ b\ d}}{\mathcal{P} \longrightarrow \mathit{adj}\ a\ b \land \mathit{path}\ b\ d}\ \land\mathcal{R}}{\mathcal{P} \longrightarrow \mathit{path}\ a\ d}\ BC$$

Cut-elimination

For a sequent calculus to be called a "logic", we expect the *Cut-Elimination Theorem* to hold.

A sequent $\Sigma\colon \Delta \longrightarrow \Gamma$ is provable if and only if it has a cut-free proof.

This theorem has many important consequences, of which we list a few here.

1. It validates that the two aspects of a logical connective — how it *reason from* and how to *reason to* — match up.
2. Consistency generally is a simple consequence.
3. In first-order systems, cut-free proofs satisfy the sub-formulas property: a formula occurrence any where in a cut-free proof is a subformula of a formula in the root sequent.
4. Allows *modus ponens* to be used: From $\vdash B$ and $\vdash B \supset C$ conclude $\vdash C$.

Modeling computation or mathematics?

Proofs of interesting mathematical theorems require extensive use of cut (lemma, libraries of theorems, etc). Although cuts can be removed *in principle*, it would be ridiculous to do so.

On the other hand, cut-free proofs can be used to describe computations, in the sense that sequences of Turing machine configurations can model computations. We expect cut-free proofs to be more declarative computational traces and hence will be more useful for reasoning about computation than other *ad hoc* mechanisms.

Logic programming as goal-directed search

Abstractly speaking, logic programming can be considered to be *goal-directed* search for cut-free sequent proofs of

$$\Sigma : \mathcal{P} \longrightarrow G$$

where Σ is the current *signature*, $\mathcal{P}$ is the current *program* or *context* and G is the current *goal*.

Goal-directed search: If G is non-atomic then this sequent must be the conclusion of a right-introduction rule. Thus, in a search, do right rules before left rules.

Informally: The “search semantics” of logical connectives is fixed and given by the right introduction rule. The program and signature are only used in searching for proofs of non-logical formulas (atoms).

The technical term *uniform proofs* has been used to formalize this notion of goal-directed search. It has also been generalized to work with multiple conclusion sequents.

If restricting to uniform proofs does not cause incompleteness, then you have a logic programming language.

We will focus on intuitionistic logic

We clearly have two different logics to consider: an object-level logic and a meta-level logic.

As we shall motivate, our meta-level logic will be intuitionistic. It will be our choice for a "formalization of the mathematics of computation".

A wide range of logics can be used for the object-level application. One picks this to best support the object-level program being specified.

For example, linear logic has shown to be valuable for encoding certain aspect of imperative and concurrent, including exceptions and input/output primitives.

In these lectures, we shall limit ourselves to just an intuitionistic object-logic. Others have been considered elsewhere.

A scheme for reasoning about computation

Let $\vdash$ be a provability in a sequent system to encode computation (e.g., intuitionistic or linear logic).

Let $\vdash^+$ be a stronger system with new introduction rules for atomic formulas, to capture induction and definitions.

We shall assume that $\vdash \subseteq \vdash^+$ and that cut-elimination holds for $\vdash$ and $\vdash^+$.

One scheme for reasoning is the following:

$\vdash$	C	A computation is witnessed.
$C \vdash^+$	D	An inference about computation.
$\vdash^+$	D	By cut.
$\vdash$	D	By inspection of a cut-free proof.

Thus the implication $C \supset D$ in the stronger system ($\vdash^+$) can be used to carry computations to computations.

Proof of type preservation

Assume the formulation of (call-by-name) evaluation and of simple typing that we provided before.

Theorem: If P evaluates to V and P has type T then V has type T.

Proof: Proceed by $n \geq 0$ that if the height of the proof of $\mathit{eval}\ \ P\ V$ is n then for all T, if $\vdash \mathit{typeof}\, P\ T$ then $\vdash \mathit{typeof}\, V\ T$.

The proof of $\mathit{eval}\ \ P\ V$ must end by backchaining on one of the formulas encoding evaluation.

Case 1: Backchaining on the *eval* of *abs*: thus P and V are equal to $\mathit{abs}\ R$, for some R, and the consequent is immediate.

Case2 : Backchaining on the *eval* of *app*: thus P is of the form $(app\ M\ \ N)$ and for some R, there are shorter proofs of $eval\ \ M\ (abs\ R)$ and $eval\ \ (R\,N)\ V$.

Since $\vdash typeof\,(app\ M\ \ N)\ T$, this typing judgment must have been proved using backchaining and, hence, there is a U such that $\vdash typeof\,M\ \ (arr\,U\ \ T)$ and $\vdash typeof\,N\ U$.

Using the inductive hypothesis, we have $\vdash typeof\,(abs\ R)\ (arr\,U\ \ T)$. This formula must have been proved by backchaining on the *typeof* formula for *abs*, and, hence, $\vdash \forall x.[typeof\,x\ U \supset typeof\,(R\,x)\ T]$.

Since our logic of judgments is intuitionistic logic, we can instantiate this quantifier with N and use cut and cut-elimination to conclude that $\vdash typeof\,(R\,N)\ T$. Using the inductive hypothesis a second time yields $\vdash typeof\,V\ T$. QED

Analyzing this informal proof

This proof is clear and natural, and we would like to have a formal reasoning setting that allow proofs quite similar to this in structure. This suggests that the following features would be valuable in the meta-logic.

1. *Two distinct logics.* The *first* logic corresponds to the one written with logical syntax above and would capture judgments, e.g. about typability and evaluation. The *second* logic would formalize the English text in the proof above. Atomic formulas of that logic would be judgments in the object-logic.
2. *Induction* over at least natural numbers.
3. *Instantiation of meta-level eigenvariables.* In the proof above, for example, the meta-level variable P was instantiated in one part of the proof to $abs\ R$ and in another part of the proof to $app\ M\ N$. Instantiation of eigenvariables within a proof does not happen in a usual sequent calculus proofs.

4. *Case analysis of the proof of an assumed judgment.* In the proof above, this was done a few times, leading, for example, from the assumption

$$\vdash \mathit{typeof}\,(\mathit{abs}\ R)\ (\mathit{arr}\ U\ T)$$

to the assumption

$$\vdash \forall x.[\mathit{typeof}\ x\ U \supset \mathit{typeof}\,(R\,x)\ T].$$

The specification of *typeof* allows the implication to go in the other direction, but given the structure of the specification of *typeof*, this direction can also be justified at the meta-level.

In our meta-logic, we accommodate the first two features by specifying an object-logic within the meta-logic and by introducing *natural numbers* and *induction*.

The last two features are accommodated by introducing a notion of *definition* and two sequent calculus rules for the left and right introduction of defined concepts.

The $FO\lambda^{\Delta\mathbb{N}}$ logic

Stands for a "first-order logic for λ-abstraction with definitions and natural number induction". Here, first-order means that there is no quantification over predicates, but there is quantification at other higher-order types.

We pronounce it "fold-n".

$FO\lambda^{\Delta\mathbb{N}}$ is a sequent calculus proof system built using the following components:

- a proof system for intuitionistic logic,
- left and right rules for introducing defined predicates, and
- natural number induction.

The last two items are represented as introduction rules for atoms, thus this is a modular extension to usual introduction rules.

We first present the proof theoretic notion of *definition.*

Iris: A prototyping implementation of $FO\lambda^{\Delta\mathbb{N}}$ in λProlog currently being developed [Wajs&Miller].