Lecture Notes on
From Proof Systems to Programming Languages

15-317: Constructive Logic
Frank Pfenning

Lecture 10
Thursday, February 16, 2023

1 Introduction

We have seen a strong correspondence between logic and computation, according to the following table:

Intuitionistic Logic	**Computation**
Proposition	Type
Introduction rule	Constructor
Elimination rule	Destructor
Proof	Program
Proof reduction	Computation step

This gives us a lot of confidence that we are designing both logical and computational rules in a manner that has a strong philosophical foundation. The influence goes both ways. For example, Prawitz's [1965] important book on natural deduction, for all its good qualities, could be more elegantly rewritten using notation and concepts from Church's [1936] λ-calculus after Howard's [1969] discovery of the correspondence between them.

Despite this strong correspondence and mutual influence, we also have to be aware of some differences. We'll investigate two of the three today, and another in the next lecture.

1. Proof reduction, especially the way we have looked at it so far using only *local reduction*, applies whenever an introduction for a connective is immediately followed by its elimination. In programming language terms, this corresponds to a destructor immediately applied to a constructor. But for programming languages we need more: we also need to impose a "strategy" or "discipline" of reduction so that computation proceeds in a predictable way. This is because we very much care about the *cost* of computation, say in terms of time or space, which was not so much a concern in the early days of intuitionistic mathematics.

2. For programming languages we have to decide which outcomes of computation we can *directly observe* and which we cannot. Almost all languages, including functional languages like ML or Haskell, take an *extensional view* of functions which means we cannot directly observe their structure. Instead, we can apply functions and observe

their results. On the other hand, values of concrete types such as booleans or natural numbers are directly observable and therefore *intensional* in nature. There do exist corresponding concepts in proof theory, and we will return to them later in the course.

3. Both types and programs use recursion, something we have not yet considered from the proof-theoretic point of view. We postpone their study to the next lecture, although we will take first steps in that direction at this end of this lecture.

2 Reduction

Recall the steps of local reduction, which we write on proof terms as $M \longrightarrow M'$.

Reduction			Type	Proposition
$(\lambda x.\, M)\, N$	$\longrightarrow$	$[N/x]M$	$(A \to B)$	$(A \supset B)$
$\mathbf{fst}\langle M, N\rangle$	$\longrightarrow$	M	$(A \times B)$	$(A \wedge B)$
$\mathbf{snd}\langle M, N\rangle$	$\longrightarrow$	N		
$\mathbf{case}(\mathbf{inl}\, M, x.\, N, y.\, P)$	$\longrightarrow$	$[M/x]N$	$(A + B)$	$(A \vee B)$
$\mathbf{case}(\mathbf{inr}\, M, x.\, N, y.\, P)$	$\longrightarrow$	$[M/y]P$		
no reductions for $\langle\,\rangle$			$\mathbf{1}$	$\top$
no reductions for $\mathbf{abort}\, M$			$\mathbf{0}$	$\bot$

There are no reductions for $\langle\,\rangle$ because there is no *destructor* for type **1**, and no reductions for **abort** M because there is no *constructor* for type **0**.

Thinking of reduction as a binary judgment on terms, the rules in the table above would be *axioms* in the sense that they would be rules with no premises. Those axioms by themselves are not sufficient to define computation. For example, under the definitions

$$\begin{array}{lcll} \mathsf{bool} & = & 1 + 1 & (\sim \top \vee \top) \\ \mathsf{true} & = & \mathbf{inl}\,\langle\,\rangle & \\ \mathsf{false} & = & \mathbf{inr}\,\langle\,\rangle & \end{array}$$

we might expect

$$\mathbf{snd}\,(\mathbf{fst}\,\langle\,\langle \mathsf{true}, \mathsf{false}\rangle, \mathsf{true}\,\rangle) \longrightarrow^2 \mathsf{false}$$

where $\longrightarrow^2$ means reduction in two steps. However, with the axioms so far we are stuck, because at the top level we have **snd** applied to a term that's not a pair.

This means we need rules that allow us to apply reductions to subterms. This example suggests the following two

$$\frac{M \longrightarrow M'}{\mathbf{fst}\, M \longrightarrow \mathbf{fst}\, M'} \qquad \frac{M \longrightarrow M'}{\mathbf{snd}\, M \longrightarrow \mathbf{snd}\, M'}$$

We also need *multistep reduction*. Mathematically, we say that $\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$.

An important subtext of today's lecture is that we communicate information via inference rules. If you open proceedings of programming languages conferences you will see that inference rules are indeed ubiquitous. So we should consider how to define multistep reduction via inference rules. Here is one approach, expressing directly the reflexive and transitive closure idea.

$$\frac{M \longrightarrow M'}{M \longrightarrow^* M'}\ \mathsf{step} \qquad \frac{}{M \longrightarrow^* M}\ \mathsf{refl} \qquad \frac{M \longrightarrow^* M' \quad M' \longrightarrow^* M''}{M \longrightarrow^* M''}\ \mathsf{trans}$$

There are other perfectly good definitions. For example, we could use the following two rules instead:

$$\frac{}{M \longrightarrow^* M}\ \mathsf{refl} \qquad \frac{M \longrightarrow M' \quad M' \longrightarrow^* M''}{M \longrightarrow^* M''}\ \mathsf{trans}^+$$

Or we could replace the second premise in the original trans rule with a single step. These three definitions would be extensionally equivalent in that $M \longrightarrow^* M'$ relates the same M and M' in all three forms. But *rule induction* over the three forms of definitions would be different induction principles. Once we prove, at the metalevel, that they are extensionally equivalent we can freely move back and forth between them as we see fit.

Back to reduction: at the moment, we have the axioms given to us by local reduction and so-called congruence rules for **fst** and **snd**. Which other rules do we need? In order to decide that we need to consider *observability*.

3 Observation

One fundamental decision for (almost?) all programming languages is that we cannot directly observe the structure of functions that are returned as the result of computation. For example, Standard ML of New Jersey

```
- val f = fn x => fn y => x;
val f = fn : 'a -> 'b -> 'a
```

happily gives us the most general type of $\lambda x.\, \lambda y.\, x$ but it prints the function just as `fn`. If we continue by apply this, say, to `1` we still do not get much information, just that the answer is still just `fn`.

```
- val g = f 1;
stdIn:3.5-3.14 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val g = fn : ?.X1 -> int
```

It also gives us a warning, but let's ignore that since it is a consequence of an advanced feature of SML not relevant here. When we apply it to two arguments instead, we get the expected answer.

```
- val one = f 1 2;
val one = 1 : int
```

Hiding the definition of functions has a number of important consequences:

- It allows the language (like ML) to compile functions to efficient low-level binary code because it never has to display such code. The main requirement is that its observable behavior (input to output) is the same as that of the function the programmer wrote.
- It affords a degree of abstraction because we can always replace a function with another one (say, improving efficiency or just refactoring the code) without disturbing the code using the function. As long as we preserve its observable *behavior* (that is, the argument/result relation) such a change is otherwise invisible.
- A function by itself is already a *value*, that is, an outcome of a computation. We do not need to apply any reduction underneath an λ-abstraction. If we decide to do so, it would be a program transformation, not a computation.

From these considerations it should be clear that we actually need two distinct notions: $M \longrightarrow M'$ (M reduces to M', which we started on) and $M\ \mathit{value}$ (M is a value, that is, the outcome of a computation). Because the outcome of a computation is a value we often refer to functional computation as *evaluation*.

Before considering further rules for these judgments, it is helpful to write out the theorems we eventually want to prove about them. Similarly, we considered local reductions and expansions as our guide when we wrote the introduction and elimination rules for the logical connectives.

The first is easy, coming from proof theory: if we reduce a deduction of $A\ \mathit{true}$ we want to obtain (more direct) deduction of the same judgment $A\ \mathit{true}$. On terms, this is called *type preservation*.

Theorem 1 (Type Preservation) *If $M : A$ and $M \longrightarrow M'$ then $M' : A$.*

It also make sense from the programming perspective: if our term M denotes, say, and integer, we want its value to be an integer and not a value of some other type like a Boolean.

The second is about the relationship between values and reduction: we want values to characterize precisely the outcome of computations. In the theory of programming languages, this is called *progress*.

Theorem 2 (Progress) *If $M : A$ then either $M \longrightarrow M'$ or $M\ \mathit{value}$ (but not both).*

The disjointness condition on the two judgments is sometimes separated out as a different theorem.

Finally, in our particular language (although not in general), we like the next step in a computation to be uniquely determined.

Theorem 3 (Small-Step Determinism)
If $M : A$, $M \longrightarrow M'$, and $M \longrightarrow M''$ then $M' = M''$.

4 Reductions and Values

With these design considerations settled, we now come back to our original goal of writing rules for $M \longrightarrow M'$ and also M *value*. We do this type-by-type, because the considerations are largely orthogonal due to our care in defining proofs and typing.

Pairs $A \times B$ ($\sim$ $A \wedge B$). It turns out that a language a priori has two kind of pairs: those with **fst** and **snd** as destructors, and one with a **case** destructor. This second one we explored in Assignment 1 where we wrote it as $A \spadesuit B$, although a more common notation is $A \otimes B$ or even just $A \times B$. Since in intuitionistic logic we have $(A \wedge B) \supset (A \otimes B)$ and $(A \otimes B) \supset (A \wedge B)$, usually only one is considered even if their behavior in a programming language is different: the version of first and second projections should be *lazy* while the one with a case-like elimination should be *eager*. In the terminology of proof theory, $A \wedge B$ is *negative* while $A \otimes B$ is *positive*. We cannot justify this very well at this point, but just think of **fst** and **snd** as ways to observe the components of a pair which is considered a value. So we have:

$$\frac{}{\langle M, N\rangle \ \mathit{value}}$$

$$\frac{}{\textbf{fst}\,\langle M, N\rangle \longrightarrow M} \qquad \frac{}{\textbf{snd}\,\langle M, N\rangle \longrightarrow N}$$

$$\frac{M \longrightarrow M'}{\textbf{fst}\,M \longrightarrow \textbf{fst}\,M'} \qquad \frac{M \longrightarrow M'}{\textbf{snd}\,M \longrightarrow \textbf{snd}\,M'}$$

It may be worth your time to consider how these rules satisfy the three desired theorems, even if we don't formally proof them (at least at this point).

Functions $A \to B$ ($\sim$ $A \supset B$). As we said in the previous section, the structure of functions is not observable so they are values, no matter what the body of the function is.

$$\frac{}{\lambda x.\, M \ \mathit{value}}$$

For function applications we need rules to reduce the function or the argument, because neither might be a value. Different languages take different approaches here: Standard ML evaluates first M then N, OCaml evaluates first N then M, and Haskell evaluates M but not N.

We chose left-to-right evaluation, where the argument has to be reduced to value. Again, we can't fully justify this choice now, but it comes from $A \supset B$ being a so-called *negative connective* which forces A to be *positive*. We are not aware of a strong proof-theoretic argument of whether to evaluate left-to-right or right-to-left.

$$\frac{M \longrightarrow M'}{M\,N \longrightarrow M'\,N} \qquad \frac{M \ \mathit{value}}{M\,N \longrightarrow M\,N'}$$

Note the premise in the second rule that enforces determinism. Similarly, we need such a premise for function application.

$$\frac{N\ \mathit{value}}{(\lambda x.\, M)\, N \longrightarrow [N/x]M}$$

Sums $A + B$ ($\sim$ $A \vee B$). Values of sum type are observable. For example, with the definition of Booleans as $1 + 1$ we would like the two values to be **inl** $\langle\,\rangle$ and **inr** $\langle\,\rangle$, one of which should represent true and the other false. Therefore they have to be "eager" in this sense:

$$\frac{M\ \mathit{value}}{\mathbf{inl}\, M\ \mathit{value}} \qquad \frac{M\ \mathit{value}}{\mathbf{inr}\, M\ \mathit{value}}$$

So here we have to be able to reduce the terms underneath the *constructors* in order to reach a value.

$$\frac{M \longrightarrow M'}{\mathbf{inl}\, M \longrightarrow \mathbf{inl}\, M'} \qquad \frac{M \longrightarrow M'}{\mathbf{inr}\, M \longrightarrow \mathbf{inr}\, M'}$$

Also, the subject of a case needs to be reduced until we reach a value.

$$\frac{M \longrightarrow M'}{\mathbf{case}(M, x.\, N, y.\, P) \longrightarrow \mathbf{case}(M', x.\, N, y.\, P)}$$

$$\frac{M\ \mathit{value}}{\mathbf{case}(\mathbf{inl}\, M, x.\, N, y.\, P) \longrightarrow [M/x]N} \qquad \frac{M\ \mathit{value}}{\mathbf{case}(\mathbf{inr}\, M, x.\, N, y.\, P) \longrightarrow [M/x]P}$$

To preserve determinism, we force the case subject to be a value.

Unit 1 ($\sim$ $\top$). The constructor here is simply a value, and there is no destructor and therefore no computation rule.

$$\frac{}{\langle\,\rangle\ \mathit{value}} \qquad \text{(no computation rule for } \mathbf{1}\text{)}$$

Void 0 ($\sim$ $\bot$). There is no constructor (and therefore no value), but we have a rule to reduce the subject of an **abort** construct. Because the subject of an abort has the empty type **0**, it can never actually reduce down to a value.

$$\text{(no value rule for } \mathbf{0}\text{)} \qquad \frac{M \longrightarrow M'}{\mathbf{abort}\, M \longrightarrow \mathbf{abort}\, M'}$$

This completes our investigation into values and computation in a programming language. We would now be in a position to actually prove the theorems of type preservation, progress, and determinism, but we prefer not to go into this detail which can be found in textbooks in theory of programming languages and requires no new techniques beyond those we already introduced (specifically: rule induction). See, for example, Harper [2016].

5 Natural Numbers

Another aspect of the differences between intuitionistic logic and programming languages so far is the presence of *recursion* in programming languages. In logic, we usually study this mostly using the exemplar of natural numbers. From a theoretical perspective, other types like lists, trees, etc. can be coded as natural numbers [Gödel, 1931] or they can be given a direct treatment in which case we are in the transition between logic and type theory. The boundary between these subject is quite fluid, however. This course will focus on logic, but will prepare you for further study in type theory.

Back to the numbers. If we want to write out the propositions we have so far considered, we can do it in so-called EBNF style.

$$\text{Propositions} \quad A \quad ::= \quad P \mid A_1 \wedge A_2 \mid A_1 \supset A_2 \mid A_1 \vee A_2 \mid \top \mid \bot$$

We tend not to do this because it suggest "that's all there is" while our enterprise is inherently an open-ended one, but sometimes it can be useful. In a similar way, we can define the natural numbers as being built from a zero 0 and a successor constructor s.

$$\text{Natural Numbers} \quad n \quad ::= \quad 0 \mid \mathsf{s}\, n$$

Before we get to the judgments of Heyting arithmetic in a later lecture, let's consider some simple judgments on natural numbers and the rules that define them.

As a first example, let's write a judgment $\mathsf{eq}\, m\, n$ which should have a derivation if and only if $m = n$. Nobody mentioned this in lecture, but the simplest form is just an axiom $\mathsf{eq}\, m\, m$. But we can also think about it in terms of breaking down the structure of the natural numbers.

$$\frac{}{\mathsf{eq}\, 0\, 0}\ \mathsf{eq00} \qquad \frac{\mathsf{eq}\, m\, n}{\mathsf{eq}\, (\mathsf{s}\, m)\, (\mathsf{s}\, n)}\ \mathsf{eqss}$$

This only let's us infer that two numbers are equal, but it does not allow us to infer when two numbers are *not* equal. One way to do that is to define an explicit judgment $\mathsf{neq}\, m\, n$. The first two rules are just axioms, because 0 is different from any success.

$$\frac{}{\mathsf{neq}\, 0\, (\mathsf{s}\, n)}\ \mathsf{neq0s} \qquad \frac{}{\mathsf{neq}\, (\mathsf{s}\, m)\, 0}\ \mathsf{neqs0}$$

There is one more case that has to be written as an inference rule: $\mathsf{s}\, m$ is different from $\mathsf{s}\, n$ if m is different from n.

$$\frac{\mathsf{neq}\, m\, n}{\mathsf{neq}\, (\mathsf{s}\, m)\, (\mathsf{s}\, n)}\ \mathsf{neqss}$$

Now we can prove properties about these judgment by rule induction. Here is one example:

Theorem 4 (Disjointness) $\forall m.\, \forall n.\, \neg(\mathsf{eq}\, m\, n \wedge \mathsf{neq}\, m\, n)$

Proof: To prove the negation, we assume $\mathsf{eq}\, m\, n$ and $\mathsf{neq}\, m\, n$ and prove a contradiction. This proof proceeds by rule induction on the derivation $\mathcal{D}$ of $\mathsf{eq}\, m\, n$. Would could also proceed by induction over $\mathcal{E}$ of $\mathsf{neq}\, m\, n$ or by simultaneous induction over both.

Case:

$$\mathcal{D} = \frac{}{\mathsf{eq}\,0\,0}\ \mathsf{eq00}$$

where $m = n = 0$. There is no rule with conclusion $\mathsf{neq}\,0\,0$ even though we have assumed there is a derivation $\mathsf{neq}\,m\,n$, which is a contradiction, as required.

Case:

$$\mathcal{D} = \frac{\begin{array}{c}\mathcal{D}' \\ \mathsf{eq}\,m'\,n'\end{array}}{\mathsf{eq}\,(\mathsf{s}\,m')\,(\mathsf{s}\,n')}\ \mathsf{eqss}$$

where $m = \mathsf{s}\,m'$ and $n = \mathsf{s}\,n'$. We have assumed there is a derivation $\mathcal{E}$ of $\mathsf{neq}\,m\,n$ which in this case has the form $\mathsf{neq}\,(\mathsf{s}\,m')\,(\mathsf{s}\,m')$. There is only one rule for the neq judgment that could have been used to derive this judgment, namely neqss. So there must be a derivation $\mathcal{E}'$ of the premise of this rule of the judgment $\mathsf{neq}\,m'\,n'$. Now we can apply the induction hypothesis to $\mathcal{D}'$ to arrive at a contradiction.

□

This proof uses the important principle of *inversion*: from the shape of a judgment we know to have a derivation, we consider all the possible ways this judgment might have been derived. In the first case above there were zero (giving us a contradiction), in the second case there was one (giving us a subderivation and an opportunity to apply the induction hypothesis).

In this particular proof, because we we derive a contradiction, the computational content is not particularly interesting. Let's consider the counterpart: coverage.

Theorem 5 (Coverage) $\forall m.\,\forall n.\,\mathsf{eq}\,m\,n \vee \mathsf{neq}\,m\,n$

Proof: This time, we are not given a derivation to perform an induction over, but we have m and n. So we can do a simultaneous induction over m and n were we stipulate that both must get smaller to apply the induction hypothesis. (Other forms of induction also work here, but contain a little less information.)

Case: $m = n = 0$. Then $\mathsf{eq00}$ applies and $\mathsf{eq}\,0\,0$.

Case: $m = 0$ and $n = \mathsf{s}\,n'$. Then $\mathsf{neq0s}$ applies and $\mathsf{neq}\,0\,(s\,n')$.

Case: $m = \mathsf{s}\,m'$ and $n = 0$. Then $\mathsf{neqs0}$ applies and $\mathsf{neq}\,(\mathsf{s}\,m')\,0$.

Case: $m = \mathsf{s}\,m'$ and $n = \mathsf{s}\,n'$. By induction hypothesis we know that either $\mathsf{eq}\,m'\,n'$ or $\mathsf{neq}\,m'\,n'$. In the first case we use the rule eqss to conclude $\mathsf{eq}\,(\mathsf{s}\,m')\,(\mathsf{s}\,n')$. In the second case we use the rule neqss to conclude $\mathsf{neq}\,(\mathsf{s}\,m')\,(\mathsf{s}\,n')$.

□

In this last proof we saw the importance of designing the rules to follow the structure of the natural numbers, breaking them down from the conclusion to the premises.

From this latter proof it is easy to extract a decision procedure for equality over natural numbers, written in SML. We assign `true` to equality $\mathsf{eq}\, m\, n$, and `false` to disequality $\mathsf{neq}\, m\, n$.

```
datatype nat = Zero | Succ of nat

(* val eq_neq : nat -> nat -> bool *)
fun eq_neq Zero Zero = true
  | eq_neq Zero (Succ n') = false
  | eq_neq (Succ m') Zero = false
  | eq_neq (Succ m') (Succ n') = eq_neq m' n'
```

References

Alonzo Church and J.B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, May 1936.

Kurt Gödel. Über formal unentscheidbare sätze der Principia Mathematica und verwandter system I. *Monatshefte für Mathematik und Physik*, 38:173–198, December 1931.

Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, second edition, April 2016.

W. A. Howard. The formulae-as-types notion of construction. Unpublished note. An annotated version appeared in: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 479–490, Academic Press (1980), 1969.

Dag Prawitz. *Natural Deduction*. Almquist & Wiksell, Stockholm, 1965.