Lecture Notes on Proofs as Programs

15-317: Constructive Logic
Frank Pfenning

Lecture 4
Thursday, January 26, 2023

1 Introduction

In this lecture we investigate a computational interpretation of constructive proofs and relate it to functional programming. On the propositional fragment of logic this is called the Curry-Howard isomorphism [Howard, 1969]. From the very outset of the development of constructive logic and mathematics, a central idea has been that *proofs ought to represent constructions*. The Curry-Howard isomorphism is only a particularly poignant and beautiful realization of this idea. In a highly influential subsequent paper Martin-Löf [1980] developed it further into a more expressive calculus called *intuitionistic type theory*.

2 Propositions as Types

In order to illustrate the relationship between proofs and programs we introduce a new judgment:

$M : A$ $\quad$ M is a proof term for proposition A

We presuppose that A is a proposition when we write this judgment. We will also interpret $M : A$ as "M *is a program of type* A". These dual interpretations of the same judgment are at the core of the Curry-Howard isomorphism. We either think of M as a syntactic term that represents the proof of A *true*, or we think of A as the type of the program M. As we discuss each connective, we give both readings of the rules to emphasize the analogy.

We intend that if $M : A$ then A *true*. Conversely, if A *true* then $M : A$ for some appropriate proof term M. But we want something more: every deduction of $M : A$ should correspond to a deduction of A *true* with an identical structure and vice versa. In other words we annotate the inference rules of natural deduction with proof terms. The property above should then be obvious. In that way, proof term M of $M : A$ will correspond directly to the corresponding proof of A *true*.

As a point of terminology, we refer to A *true* as a *synthetic judgment* since we need external evidence for it in the form of a natural deduction. On the other hand, $M : A$

is a *analytic judgment* in that it contains its own evidence and we can effectively check it. Martin-Löf [1994] further elaborates on this important distinction.

Conjunction. Constructively, we think of a proof of $A \wedge B$ *true* as a pair of proofs: one for A *true* and one for B *true*. So if M is a proof of A and N is a proof of B, then the pair $\langle M, N\rangle$ is a proof of $A \wedge B$.

$$\frac{M : A \quad N : B}{\langle M, N\rangle : A \wedge B}\ \wedge I$$

The elimination rules correspond to the projections from a pair to its first and second elements to get the individual proofs back out from a pair M.

$$\frac{M : A \wedge B}{\mathbf{fst}\, M : A}\ \wedge E_1 \qquad \frac{M : A \wedge B}{\mathbf{snd}\, M : B}\ \wedge E_2$$

Hence the conjunction $A \wedge B$, as a proposition, corresponds to the product type $A \times B$. And, indeed, product types in functional programming languages have the same property that conjunctions $A \wedge B$ have. Constructing a pair $\langle M, N\rangle$ of type $A \times B$ requires a program M of type A and a program N of type B (as in $\wedge I$). Given a pair M of type $A \times B$, its first component of type A can be retrieved by the projection $\mathbf{fst}\, M$ (as in $\wedge E_1$), its second component of type B by the projection $\mathbf{snd}\, M$ (as in $\wedge E_2$).

In general, the introduction rules for a logical connective correspond to the *constructors* for the corresponding type. Conversely, the elimination rules correspond to *destructors*.

Truth. Constructively, we think of a proof of $\top$ *true* as a unit element that carries no information.

$$\frac{}{\langle\rangle : \top}\ \top I$$

Hence $\top$ corresponds to the unit type $\mathbf{1}$ with one element. There is no elimination rule and hence no further proof term constructs for truth. Indeed, we have not put any information into $\langle\rangle$ when constructing it via $\top I$, so cannot expect to get any information out by an elimination rule.

Implication. Constructively, we think of a proof of $A \supset B$ *true* as a function which transforms a proof of A *true* into a proof of B *true*.

In mathematics and many programming languages, we define a function f of a variable x by writing $f(x) = \ldots$ where the right-hand side "$\ldots$" depends on x. For example, we might write $f(x) = x^2 + x - 1$. In functional programming, we can instead write $f = \lambda x.\ x^2 + x - 1$, that is, we explicitly form a functional object by λ*-abstraction* of a variable (x, in the example).

In the concrete syntax of Standard ML language, $\lambda x.\ M$ is written and $\mathsf{fn}\, x \Rightarrow M$, but we will use the universal and original notation due to Church and Rosser [1936]. In general, we use a dot (".") to separate so-called *bound variable* from its scope. This is exactly the same notion of scope as for hypotheses in natural deduction.

We now use the notation of λ-abstraction to annotate the rule of implication introduction with proof terms.

$$\dfrac{\begin{array}{c}\overline{u : A}\ u \\ \vdots \\ M : B\end{array}}{\lambda u.\, M : A \supset B}\ \supset I^u$$

The hypothesis label u acts as a variable, and any use of the hypothesis labeled u in the proof of B corresponds to an occurrence of u in M.

Notice how a constructive proof of B *true* from the additional assumption A *true* to establish $A \supset B$ *true* also describes the transformation of a proof of A *true* to a proof of B *true*. But the proof term $\lambda u.\, M$ explicitly represents this transformation syntactically as a function, instead of leaving this construction implicit by inspection of whatever the proof does.

As a concrete example, consider the simple proof of $A \supset A$ *true*:

$$\dfrac{\overline{A\ \mathit{true}}\ u}{A \supset A\ \mathit{true}}\ \supset I^u$$

If we annotate the deduction with proof terms, we obtain

$$\dfrac{\overline{u : A}\ u}{(\lambda u.\, u) : A \supset A}\ \supset I^u$$

So our proof corresponds to the identity function id at type A which simply returns its argument.

Constructively, a proof of $A \supset B$ *true* is a function transforming a proof of A *true* to a proof of B *true*. Using $A \supset B$ *true* by its elimination rule $\supset E$, thus, corresponds to providing the proof of A *true* that $A \supset B$ *true* is waiting for to obtain a proof of B *true*. The rule for implication elimination corresponds to function application. Following the convention in functional programming, we write $M\ N$ for the application of the function M to argument N, rather than the more verbose $M(N)$.

$$\dfrac{M : A \supset B \quad N : A}{M\ N : B}\ \supset E$$

What is the meaning of $A \supset B$ as a type? From the discussion above it should be clear that it can be interpreted as a function type $A \to B$. The introduction and elimination rules for implication can also be viewed as typing rules for functional abstraction $\lambda u.\, M$ and application $M\ N$. Forming a function $\lambda u.\, M$ corresponds to a function that accepts input parameter u of type A and produces M of type B (as in $\supset I$). Using a function $M : A \to B$ corresponds to applying it to an argument N of type A to obtain an output $M\ N$ of type B.

Note that we obtain the usual introduction and elimination rules for implication if we erase the proof terms. This will continue to be true for all rules in the remainder of this section and is immediate evidence for the soundness of the proof term calculus, that is, if $M : A$ then A *true*.

Disjunction. Constructively, we think of a proof of $A \vee B$ *true* as either a proof of A *true* or B *true*. Disjunction therefore corresponds to a *disjoint sum* type $A + B$, whose values are either of type A or type B. In order to make sure we can tell which one, such a value is *tagged* with either **inl** (if it is of type A) or **inr** (if it is of type B). We say that **inl** and **inr** *injects* a value of type A or B, respectively, into the sum type $A + B$.

$$\frac{M : A}{\mathbf{inl}\, M : A \vee B}\ \vee I_1 \qquad \frac{N : B}{\mathbf{inr}\, N : A \vee B}\ \vee I_2$$

When using a disjunction $A \vee B$ *true* in a proof, we need to be prepared to handle A *true* as well as B *true*, because we don't know whether $\vee I_1$ or $\vee I_2$ was used to prove it. The elimination rule corresponds to a case construct which discriminates between a left and right injection into a sum type.

$$\frac{\begin{array}{ccc} & \overline{u : A}^{\,u} & \overline{w : B}^{\,w} \\ & \vdots & \vdots \\ M : A \vee B & N : C & P : C \end{array}}{\mathbf{case}(M, u.\, N, w.\, P) : C}\ \vee E^{u,w}$$

Recall that the hypothesis labeled u is available only in the proof of the second premise and the hypothesis labeled w only in the proof of the third premise. This means that the scope of the variable u is N, while the scope of the variable w is P. By writing $u.\, N$ and $w.\, P$ we indicate both that u and w are bound with their corresponding scope N and P.

Falsehood. There is no introduction rule for falsehood ($\bot$). We can therefore view it as the empty type **0**. The corresponding elimination rule allows a term of $\bot$ to stand for an expression of any type when wrapped with **abort**. However, there is no computation rule for it, which means during computation of a valid program we will never try to evaluate a term of the form **abort** M.

$$\frac{M : \bot}{\mathbf{abort}\, M : C}\ \bot E$$

3 Some Examples

Consider $(A \wedge (A \supset B)) \supset B$ *true*. Let's first write a natural deduction.

$$\dfrac{\dfrac{\dfrac{\overline{A \wedge (A \supset B)\ \mathit{true}}^{\,x}}{A \supset B\ \mathit{true}}\ \wedge E_2 \quad \dfrac{\overline{A \wedge (A \supset B)\ \mathit{true}}^{\,x}}{A\ \mathit{true}}\ \wedge E_1}{B\ \mathit{true}}\ \supset E}{(A \wedge (A \supset B)) \supset B\ \mathit{true}}\ \supset I^x$$

While we construct this in a combination of bottom-up and top-down steps, annotating it with proof terms is best to do top-down. We start with the hypotheses.

$$\dfrac{\dfrac{\dfrac{\overline{x : A \wedge (A \supset B)}\,x}{A \supset B}\wedge E_2 \quad \dfrac{\overline{x : A \wedge (A \supset B)}\,x}{A}\wedge E_1}{B}\supset E}{(A \wedge (A \supset B)) \supset B}\supset I^x$$

Then the conjunction eliminations, since we now have proof terms for their premises (x, actually, in both cases).

$$\dfrac{\dfrac{\dfrac{\overline{x : A \wedge (A \supset B)}\,x}{\mathbf{snd}\,x : A \supset B}\wedge E_2 \quad \dfrac{\overline{x : A \wedge (A \supset B)}\,x}{\mathbf{fst}\,x : A}\wedge E_1}{B}\supset E}{(A \wedge (A \supset B)) \supset B}\supset I^x$$

Next we can annotate the implication elimination

$$\dfrac{\dfrac{\dfrac{\overline{x : A \wedge (A \supset B)}\,x}{\mathbf{snd}\,x : A \supset B}\wedge E_2 \quad \dfrac{\overline{x : A \wedge (A \supset B)}\,x}{\mathbf{fst}\,x : A}\wedge E_1}{(\mathbf{snd}\,x)\,(\mathbf{fst}\,x) : B}\supset E}{(A \wedge (A \supset B)) \supset B}\supset I^x$$

and finally introduce the λ-abstraction at the root:

$$\dfrac{\dfrac{\dfrac{\overline{x : A \wedge (A \supset B)}\,x}{\mathbf{snd}\,x : A \supset B}\wedge E_2 \quad \dfrac{\overline{x : A \wedge (A \supset B)}\,x}{\mathbf{fst}\,x : A}\wedge E_1}{(\mathbf{snd}\,x)\,(\mathbf{fst}\,x) : B}\supset E}{\lambda x.\,(\mathbf{snd}\,x)\,(\mathbf{fst}\,x) : (A \wedge (A \supset B)) \supset B}\supset I^x$$

We can also go the other way, write the program first and then expand it into a natural deduction. For example, consider the task of writing a program of type $(A \times B) \rightarrow (B \times A)$. This would often be written as $(\alpha \times \beta) \rightarrow (\beta \times \alpha)$ or more concretely as `'a * 'b -> 'b * 'a`. Such a function is easy to construct: it should swap the elements of a pair.

$$\lambda x.\,\langle \mathbf{snd}\,x, \mathbf{fst}\,x\rangle : (A \wedge B) \supset (B \wedge A)$$

Now we can "unwind" this into a natural deduction. For this direction, we work entirely bottom-up.

$$\dfrac{\vdots}{\lambda x.\,\langle \mathbf{snd}\,x, \mathbf{fst}\,x\rangle : (A \wedge B) \supset (B \wedge A)}$$

The term starts with a λ-abstraction, which means the natural deduction starts with $\supset I$.

$$\dfrac{\dfrac{\overline{x : A \wedge B}\ x}{\vdots}}{\lambda x.\, \langle \mathbf{snd}\, x, \mathbf{fst}\, x\rangle : (A \wedge B) \supset (B \wedge A)}\ \supset I^x$$

The fact that the proof term is a pair tells us the next rule should be $\wedge I$.

$$\dfrac{\dfrac{\dfrac{\overline{x : A \wedge B}\ x}{\vdots}}{\mathbf{snd}\, x : B} \qquad \dfrac{\dfrac{\overline{x : A \wedge B}\ x}{\vdots}}{\mathbf{fst}\, x : A}}{\langle \mathbf{snd}\, x, \mathbf{fst}\, x\rangle : B \wedge A}\ \wedge I}{\lambda x.\, \langle \mathbf{snd}\, x, \mathbf{fst}\, x\rangle : (A \wedge B) \supset (B \wedge A)}\ \supset I^x$$

Now we can fill in the remaining gaps, reading the rules $\wedge E_2$ and $\wedge E_1$ off the proof terms $\mathbf{snd}\, x$ and $\mathbf{fst}\, x$, respectively.

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{x : A \wedge B}\ x}{\mathbf{snd}\, x : B}\ \wedge E_2 \qquad \dfrac{\overline{x : A \wedge B}\ x}{\mathbf{fst}\, x : A}\ \wedge E_1 x}{\langle \mathbf{snd}\, x, \mathbf{fst}\, x\rangle : B \wedge A}\ \wedge I}{\lambda x.\, \langle \mathbf{snd}\, x, \mathbf{fst}\, x\rangle : (A \wedge B) \supset (B \wedge A)}\ \supset I^x$$

Programs in constructive propositional logic are somewhat uninteresting in that they do not manipulate basic data types such as natural numbers, integers, lists, trees, etc. We introduce such data types later in this course, following the same method we have used in the development of logic.

Summary. To close this section we recall the *guiding principles* behind the correspondence

1. Every proposition corresponds to a type and vice versa.
2. Introduction rules correspond to value constructors.
3. Elimination rules correspond to value destructors.
4. For every deduction of A *true* there is a proof term M and deduction of $M : A$. We can effectively construct this top-down.
5. For every deduction of $M : A$ there is a deduction of A *true*. We can effective construct this bottom-up.

We can also observe that a given term can correspond to more than one proof. For example, $\lambda x.\, x$ could be a proof of $A \supset A$, but it is equally suitable as a proof of $(A \supset B) \supset (A \supset B)$. From the theory of programming languages Milner [1978] we know that there is

always a *most general type* for a (well-typed) term, which corresponds to the *most general proposition* proved by a given proof term. Concretely, this means we can obtain all other natural deductions from the most general one by instantiating the schematic variables (like A, B, etc.) with other propositions without affecting the structure of the deduction.

We can also disambiguate the terms by adding type information. For example $\lambda x{:}A \supset A.\, x$ then must be proof of $(A \supset A) \supset (A \supset A)$. We will develop this idea further in a future lecture (perhaps already the next one).

4 Reduction

In the preceding section, we have introduced the assignment of proof terms to natural deductions. If proofs are programs then we need to explain how proofs are to be executed, and which results may be returned by a computation.

We explain the operational interpretation of proofs in two steps. In the first step we introduce a judgment of *reduction* written $M \Longrightarrow_R M'$ and read "M *reduces to* M'". In the second step, a computation then proceeds by a sequence of reductions $M \Longrightarrow_R M_1 \Longrightarrow_R M_2 \Longrightarrow_R \ldots$, according to a fixed strategy, until we reach a value which is the result of the computation. In this section we cover reduction; we may return to reduction strategies in a later lecture.

As in the development of propositional logic, we discuss each of the connectives separately, taking care to make sure the explanations are independent. This means we can consider various sublanguages and we can later extend our logic or programming language without invalidating the results from this section. Furthermore, it greatly simplifies the analysis of properties of the reduction rules.

As mentioned before, we think of the proof terms corresponding to the introduction rules as the *constructors* and the proof terms corresponding to the elimination rules as the *destructors*. The key guiding principle is:

> *Reduction arises when a destructor is applied to a constructor.*

It turns out this is exactly the same as:

> *A proof reduction arises when we apply an elimination rule to the result of an introduction rules.*

Therefore, local proof reductions (our evidence for local soundness of the elimination rules) correspond precisely to term reductions.

Conjunction. Recall one of the two local reductions for conjunction:

$$\dfrac{\dfrac{\begin{array}{cc}\mathcal{D} & \mathcal{E} \\ A\ \mathit{true} & B\ \mathit{true}\end{array}}{A \wedge B\ \mathit{true}}\wedge I}{A\ \mathit{true}}\wedge E_1 \qquad \Longrightarrow_R \qquad \begin{array}{c}\mathcal{D} \\ A\end{array}$$

Let's annotate these deduction with proof terms

$$\dfrac{\dfrac{\begin{array}{cc}\mathcal{D} & \mathcal{E} \\ M : A & N : B\end{array}}{\langle M, N\rangle : A \wedge B}\,\wedge I}{\mathbf{fst}\,\langle M, N\rangle : A}\,\wedge E_1 \qquad \Longrightarrow_R \qquad M : A$$

We can read off the term reduction (and its symmetric form) as

$$\begin{array}{rcl}\mathbf{fst}\,\langle M, N\rangle & \Longrightarrow_R & M \\ \mathbf{snd}\,\langle M, N\rangle & \Longrightarrow_R & N\end{array}$$

We can see in each rule we have a destructor (**fst** or **snd**) applied to a constructor ($\langle \_, \_ \rangle$). The fact that they can both reduce follows from the fact that the elimination rules are locally sound: each elimination (destructor) applied to the result of an introduction (constructor) can be reduced.

In terms of programming language theory, this foreshadows the result that *well-typed programs cannot go wrong* [Milner, 1978], which in modern language we call *progress* (see, for example, Harper [2016]). Moreoever, the fact that the derivation before and after the reduction prove the same proposition implies *preservation*, that is, if $M : A$ and $M \Longrightarrow_R M'$ then $M' : A$.

Truth. The constructor just forms the unit element, $\langle\,\rangle$. Since there is no destructor, there is no reduction rule.

Implication. The constructor forms a function by λ-abstraction, while the destructor applies the function to an argument. In general, the application of a function to an argument is computed by *substitution*. As a simple example from mathematics, consider the following equivalent definitions

$$f(x) = x^2 + x - 1 \qquad f = \lambda x.\, x^2 + x - 1$$

and the computation

$$f(3) = (\lambda x.\, x^2 + x - 1)(3) = [3/x](x^2 + x - 1) = 3^2 + 3 - 1 = 11$$

In the second step, we substitute 3 for occurrences of x in $x^2 + x - 1$, the *body of the λ-expression*. We write $[3/x](x^2 + x - 1) = 3^2 + 3 - 1$.

In general, the notation for the substitution of N for occurrences of x in M is $[N/x]M$. We therefore write the reduction rule as

$$(\lambda x.\, M)\, N \quad \Longrightarrow_R \quad [N/x]M$$

We have to be somewhat careful so that substitution behaves correctly. In particular, no variable in N should be bound in M in order to avoid conflict. We can always achieve this by renaming bound variables—an operation which clearly does not change the meaning of

a proof term. Again, this computational reduction directly relates to the logical reduction from the local soundness using the substitution notation for the right-hand side:

$$\dfrac{\dfrac{\dfrac{}{x : A}\,x}{\vdots}}{\dfrac{\dfrac{M : B}{\lambda x.\, M : A \supset B}\,\supset I^x \qquad N : A}{(\lambda x.\, M)\, N : B}\,\supset E} \qquad \Longrightarrow_R \qquad [N/x]M : B$$

Disjunction. There are two local reductions for disjunction. We show one of the two

$$\dfrac{\dfrac{\mathcal{D}}{A\ true}}{\dfrac{A \vee B\ true}{C\ true}\,\vee I_1 \quad \begin{array}{ccc} \dfrac{}{A\ true}\,u & \dfrac{}{B\ true}\,w \\ \mathcal{E} & \mathcal{F} \\ C\ true & C\ true \end{array}}\,\vee E^{u,w} \qquad \Longrightarrow_R \qquad \begin{array}{c} \dfrac{\mathcal{D}}{A\ true}\,u \\ \mathcal{E} \\ C\ true \end{array}$$

and annotate it with proof terms:

$$\dfrac{\dfrac{M : A}{\mathbf{inl}\, M : A \vee B}\,\vee I_1 \quad \begin{array}{ccc} \dfrac{}{u : A}\,u & \dfrac{}{w : B}\,w \\ \vdots & \vdots \\ N : C & P : C \end{array}}{\mathbf{case}(\mathbf{inl}\, M, u.\, N, w.\, P) : C}\,\vee E^{u,w} \qquad \Longrightarrow_R \qquad [M/u]N : C$$

Computationally, the constructors (**inl** and **inr**) inject into a sum type, and the destructor (**case**) distinguishes these two cases. Summarizing the computational reduction rules, we have

$$\begin{array}{lcl} \mathbf{case}(\mathbf{inl}\, M, u.\, N, w.\, P) & \Longrightarrow_R & [M/u]N \\ \mathbf{case}(\mathbf{inr}\, M, u.\, N, w.\, P) & \Longrightarrow_R & [M/w]P \end{array}$$

Falsehood. Since there is no constructor for the empty type there is no reduction rule for falsehood. There is no computation rule and we will not try to evaluate **abort** M.

5 Example Computations

We return to an example of proof reduction from last lecture:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{C \supset C\ true}\,x}{B \supset (C \supset C)\ true}\,\supset I^y}{(C \supset C) \supset (B \supset (C \supset C))\ true}\,\supset I^x \qquad \dfrac{\dfrac{}{C\ true}\,z}{C \supset C\ true}\,\supset I^z}{B \supset (C \supset C)\ true}\,\supset E$$

Annotating it with proof terms (recall this is best done top-down):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : C \supset C}\,x}{\lambda y.\, x : B \supset (C \supset C)}\,\supset I^y}{\lambda x.\, \lambda y.\, x : (C \supset C) \supset (B \supset (C \supset C))}\,\supset I^x \quad \dfrac{\dfrac{}{z : C}\,z}{\lambda z.\, z : C \supset C}\,\supset I^z}{(\lambda x.\, \lambda y.\, x)\,(\lambda z.\, z) : B \supset (C \supset C)}\,\supset E$$

We can reduce this proof term.

$$(\lambda x.\, \lambda y.\, x)\,(\lambda z.\, z) \Longrightarrow_R [(\lambda z.\, z)/x](\lambda y.\, x) = \lambda y.\, \lambda z.\, z$$

We can unwind the result back into a natural deduction and (as our theory predicts) we obtain the same one as in the last lecture, just written with proof terms.

$$\dfrac{\dfrac{\dfrac{}{z : C}\,z}{\lambda x.\, z : C \supset C}\,\supset I^z}{\lambda y.\, \lambda z.\, z : B \supset (C \supset C)}\,\supset I^y$$

As a final example we consider a simple program for the composition of two functions. It takes a pair of two functions, one from A to B and one from B to C and returns their composition which maps A directly to C.

$$\mathsf{comp} \quad : \quad ((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$$

We transform the following implicit definition into our notation step-by-step:

$$\begin{array}{rcl} \mathsf{comp}\,\langle f, g\rangle\,(w) & = & g(f(w)) \\ \mathsf{comp}\,\langle f, g\rangle & = & \lambda w.\, g(f(w)) \\ \mathsf{comp}\,u & = & \lambda w.\, (\mathbf{snd}\,u)\,((\mathbf{fst}\,u)w) \\ \mathsf{comp} & = & \lambda u.\, \lambda w.\, (\mathbf{snd}\,u)\,((\mathbf{fst}\,u)\,w) \end{array}$$

The final definition represents a correct proof term, as witnessed by the following deduction that directly follows the proof term.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{u : (A \supset B) \wedge (B \supset C)}\,u}{\mathbf{snd}\,u : B \supset C}\,\wedge E_2 \quad \dfrac{\dfrac{\dfrac{}{u : (A \supset B) \wedge (B \supset C)}\,u}{\mathbf{fst}\,u : A \supset B}\,\wedge E_1 \quad \dfrac{}{w : A}\,w}{(\mathbf{fst}\,u)\,w : B}\,\supset E}{(\mathbf{snd}\,u)\,((\mathbf{fst}\,u)\,w) : C}\,\supset E}{\lambda w.\, (\mathbf{snd}\,u)\,((\mathbf{fst}\,u)\,w) : A \supset C}\,\supset I^w}{(\lambda u.\, \lambda w.\, (\mathbf{snd}\,u)\,((\mathbf{fst}\,u)\,w)) : ((A \supset B) \wedge (B \supset C)) \supset (A \supset C)}\,\supset I^u$$

This proof can be read off directly from the proof term we constructed above, since it directly describes the shape of the proof and the rule to apply. For example $\mathbf{snd}u$ indicates

that $\wedge E_2$ has been used on u. We could also have first conducted the proof of $((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$ *true* in the same way that the above proof works and then annotate the proof with proof terms.

We now verify that the composition of two identity functions reduces again to the identity function. First, we verify the typing of this application.

$$(\lambda u.\, \lambda w.\, (\mathbf{snd}\, u)\, ((\mathbf{fst}\, u)\, w))\, \langle (\lambda x.\, x), (\lambda y.\, y)\rangle \quad : \quad A \supset A$$

For this typing, we use the substitution $[A/A, A/B, A/C]$, which we can apply without changing the proof term or the structure of the natural deduction (just the propositions appearing in it)

Now we show a possible sequence of reduction steps. This is by no means uniquely determined. In each step we have underlined the *redex*, that is, the subterm that is reduced.

$$\begin{array}{ll} & \underline{(\lambda u.\, \lambda w.\, (\mathbf{snd}\, u)\, ((\mathbf{fst}\, u)\, w))\, \langle (\lambda x.\, x), (\lambda y.\, y)\rangle} \\ \Longrightarrow_R & \lambda w.\, \underline{(\mathbf{snd}\, \langle (\lambda x.\, x), (\lambda y.\, y)\rangle)}\, ((\mathbf{fst}\, \langle (\lambda x.\, x), (\lambda y.\, y)\rangle)\, w) \\ \Longrightarrow_R & \lambda w.\, (\lambda y.\, y)\, (\underline{(\mathbf{fst}\, \langle (\lambda x.\, x), (\lambda y.\, y)\rangle)}\, w) \\ \Longrightarrow_R & \lambda w.\, (\lambda y.\, y)\, \underline{((\lambda x.\, x)\, w)} \\ \Longrightarrow_R & \lambda w.\, \underline{(\lambda y.\, y)\, w} \\ \Longrightarrow_R & \lambda w.\, w \end{array}$$

We see that we may need to apply reduction steps to subterms in order to reduce a proof term to a form in which it can no longer be reduced. We postpone a more detailed discussion of this.

6 Summary of the Proofs as Programs Correspondence

The proofs-as-programs correspondence consist of three interconnected relationships.

Propositions as Types.

Connective	**Proposition**	**Type**		Values
conjunction	$A \wedge B$	$A \times B$	products	$\langle \_, \_ \rangle$
implication	$A \supset B$	$A \to B$	functions	$\lambda x.\, \_$
disjunction	$A \vee B$	$A + B$	sums	$\mathbf{inl}\ \_ \quad \mathbf{inr}\ \_$
truth	$\top$	$\mathbf{1}$	unit	$\langle\,\rangle$
falsehood	$\bot$	$\mathbf{0}$	void	(*no value*)

Proofs as Programs. For a proposition A, the synthetic judgment A *true* corresponds to the analytic judgment $M : A$ where M is a proof term. The inference rules for $M : A$ are given in Figure 1. Erasing the proof terms and adding *true* yields rules for the judgment

A *true*. The correspondences between rules and terms can be read off the rules and are summarized here.

Proposition	Rules	Terms	Type
$A \wedge B$	$\wedge I$	$\langle \_, \_ \rangle$	$A \times B$
	$\wedge E_1, \wedge E_2$	**fst** $\_$, **snd** $\_$	
$A \supset B$	$\supset I^x$	$\lambda x. \_$	$A \to B$
	$\supset E$	$\_\ \_$	
$A \vee B$	$\vee I_1, \vee I_2$	**inl** $\_$, **inr** $\_$	$A \vee B$
	$\vee E^{u,w}$	**case**$(\_, u. \_, w. \_)$	
$\top$	$\top I$	$\langle\,\rangle$	$\mathbf{1}$
	(no $\top E$)	(no destructor)	
$\bot$	(no $\bot I$)	(no constructor)	$\mathbf{0}$
	$\bot E$	**abort** $\_$	

Proof Reduction as Computation. The local reductions that serves as witnesses for the local soundness of the connections are interpreted as basic rule of computation for the terms. The basic judgment is $M \Longrightarrow_R M'$, where we presuppose $M : A$ and find that $M' : A$.

$A \times B$	**fst** $\langle M, N \rangle$	$\Longrightarrow_R$	M
	snd $\langle M, N \rangle$	$\Longrightarrow_R$	N
$A \to B$	$(\lambda u. M)\, N$	$\Longrightarrow_R$	$[N/u]M$
$A \vee B$	**case**(**inl** $M, u. N, w. P$)	$\Longrightarrow_R$	$[M/u]N$
	case(**inr** $M, u. N, w. P$)	$\Longrightarrow_R$	$[M/w]P$
$\mathbf{1}$	no reduction for $\langle\,\rangle$		
$\mathbf{0}$	no reduction for **abort** M		

7 Revisiting Local Expansion

We saw in the last lecture, that eliminations (destructors) applied to the result of introductions (constructor) give rise to computation in the form of a reduction.

What about local expansion as a witness for completeness? It turns out that the local expansions are less relevant to computation. What they tell us, for example, is that if we need to return a pair from a function, we can always construct it as $\langle M, N \rangle$ for some M and N. Another example would be that whenever we need to return a function, we can always construct it as $\lambda u. .\ M$ for some M.

We can derive what the local expansion must be by annotating the deductions witnessing local expansions on proofs from this lecture with proof terms. We leave this as an exercise to the reader. The left-hand side of each expansion has the form $M : A$, where M is an arbitrary term and A is a logical connective or constant applied to arbitrary propositions. On the right hand side we have to apply a destructor to M and then reconstruct a term of the original type. The resulting rules are summarized below.

Constructors	Destructors
$\dfrac{M : A \quad N : B}{\langle M, N\rangle : A \wedge B}\ \wedge I$	$\dfrac{M : A \wedge B}{\mathbf{fst}\, M : A}\ \wedge E_1$ $\dfrac{M : A \wedge B}{\mathbf{snd}\, M : B}\ \wedge E_2$
$\dfrac{}{\langle\rangle : \top}\ \top I$	no destructor for $\top$
$\dfrac{\begin{array}{c}\overline{u : A}^{\,u} \\ \vdots \\ M : B\end{array}}{\lambda u.\, M : A \supset B}\ \supset I^u$	$\dfrac{M : A \supset B \quad N : A}{M\, N : B}\ \supset E$
$\dfrac{M : A}{\mathbf{inl}\, M : A \vee B}\ \vee I_1$ $\dfrac{N : B}{\mathbf{inr}\, N : A \vee B}\ \vee I_2$	$\dfrac{M : A \vee B \quad \begin{array}{c}\overline{u : A}^{\,u} \\ \vdots \\ N : C\end{array} \quad \begin{array}{c}\overline{w : B}^{\,w} \\ \vdots \\ P : C\end{array}}{\mathbf{case}(M, u.\, N, w.\, P) : C}\ \vee E^{u,w}$
no constructor for $\bot$	$\dfrac{M : \bot}{\mathbf{abort}\, M : C}\ \bot E$

Figure 1: Proof term assignment for natural deduction

$$
\begin{array}{lll}
M : A \wedge B & \Longrightarrow_E & \langle \mathbf{fst}\, M, \mathbf{snd}\, M \rangle \\
M : A \supset B & \Longrightarrow_E & \lambda u.\, M\, u \quad \text{for } u \text{ not free in } M \\
M : \top & \Longrightarrow_E & \langle\,\rangle \\
M : A \vee B & \Longrightarrow_E & \mathbf{case}(M, u.\, \mathbf{inl}\, u, w.\, \mathbf{inr}\, w) \\
M : \bot & \Longrightarrow_E & \mathbf{abort}\, M
\end{array}
$$

8 Programs as Proofs

The strong correspondence between proofs and programs means that we can also use it in the opposite direction. In particular, we can (to some extent) use a functional language as a proof checker! We use SML, in which sum types are not native but are available in the form of `datatype` declaration. We show here the SML code we live-coded in lecture, with a few additional annotations.

A key issue is that we have to check, by hand, that the purported proof terms don't use recursion, exceptions, or effects. You can find the source file at pcheck.sml.

```
1  datatype ('a,'b) sum = inl of 'a | inr of 'b
2  datatype zero = void of zero
3  type 'a not = 'a -> zero
4
5  (* case_ : ('a, 'b) sum * ('a -> 'c) * ('b -> 'c) -> 'c *)
6  fun case_ (M, F, G) = case M
7                         of inl x => F x
8                          | inr y => G y
9
10 fun fst (x,y) = x
11 fun snd (x,y) = y
12
13 (* A /\ B => B /\ A *)
14 val ex1 : 'a * 'b -> 'b * 'a =
15  fn x => (snd x, fst x)
16
17 (* A /\ (A => B) => B *)
18 val ex2 : 'a * ('a -> 'b) -> 'b =
19  fn x => (snd x) (fst x)
20
21 (* A \/ B => B \/ A *)
22 val ex3 : ('a, 'b) sum -> ('b, 'a) sum =
23  fn x => case_ (x, fn y => inr y, fn z => inl z)
24
25 (* T => F *)
26 (* incorrect proof using recursion *)
27 (*
28 val rec ex4 : unit -> zero = fn () => ex4 ()
29  *)
30
31 val ex4 : ('a * 'a not) not =
32     fn x => (snd x) (fst x)
```

```
33
34 (* incorrect proof to see error message *)
35 (*
36 val ex5 : ('a * 'a not) not =
37     fn x => (fst x) (snd x)
38 *)
```

9 Revisiting Equivalence[1]

Picking up some optional material from last lecture, we assign proof terms and examine local reductions and expansions on such terms. The local reduction should give us a rule of computation; the local expansion an extensional equality principle.

$$\dfrac{\dfrac{}{x : A\ true}\,x \quad \dfrac{}{y : B\ true}\,y \\ \vdots \qquad\qquad \vdots \\ N : B\ true \qquad M : A\ true}{(\!|x.\, N, y.\, M|\!) : A \equiv B\ true}\equiv I^{x,y}$$

$$\dfrac{M : A \equiv B\ true \quad N : A\ true}{\frown\ M\ N : B\ true}\equiv E_1 \qquad \dfrac{M : A \equiv B\ true \quad N : B\ true}{\frowny\ M\ N : A\ true}\equiv E_2$$

We can now annotate the local reductions and expansion with proof terms and read off:

$$\frown\ (\!|x.\, N, y.\, M|\!)\ P \Longrightarrow_R [P/x]N$$
$$\frowny\ (\!|x.\, N, y.\, M|\!)\ P \Longrightarrow_R [P/y]M$$
$$M : A \equiv B \Longrightarrow_E (\!|x.\frown\ M\ x, y.\frowny\ N\ y|\!)$$

Introducing new syntax for new connectives and programs can be tedious and difficult to use. Therefore, in practice, we probably wouldn't define logical equivalence as a new primitive, but use *notational definition* (as we did for negation):

$$A \equiv B \quad \triangleq \quad (A \supset B) \wedge (B \supset A)$$

whose meaning as a type is simply a pair of functions between the types A and B.

References

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

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

[1] Not covere in lecture

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.

Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North-Holland, 1980.

Per Martin-Löf. Analytic and synthetic judgements in type theory. In P. Parrini, editor, *Kant and Contemporary Epistemology*, pages 87–99. Kluwer Academic Publishers, 1994.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.