

Curry-Howard Correspondence

Constructive Logic (15-317)

Instructor: Giselle Reis

Lecture 04

1 Introduction

There are possibly thousands of logics. Classical, intuitionistic, modal, temporal, linear, hybrid, paraconsistent, deontic, epistemic... just to name a few. Every day a logician wakes up and may decide they will invent a new one, give it a meaning, proof system and use. With all of these options, why are we studying constructive logic? Well, because we are computer scientists! This answer should start making sense in this lecture.

The Curry-Howard correspondence is not a thing that was suddenly discovered, formalized and given a name. It is actually the organization of several observations made through many years by different people. Little by little people were realizing that those observations were actually the same, and then they decided to make it a thing. As it is known today, the Curry-Howard correspondence establishes a relation between formulas and proofs of those formulas in propositional intuitionistic logic, and functions of a given type in a functional programming language. More concretely, if we interpret atomic propositions as basic types and logical connectives as type constructors (in SML: $*$, \rightarrow , $|$), then the proof of a formula corresponds to a program of the associated type.

First of all, let us see how formulas can be viewed as types.

The smallest piece of a formula is an atomic proposition, and the smallest piece of a type is an atomic type, so it is only natural that we relate one to the other. Since we usually work with proofs using formula variables A, B, C , we will consider only type variables as well.

Conjunction translates as the product type: $A \wedge B$ assumes proofs of both A and B , analogously, $'a * 'b$ assumes two terms (or programs, or functions, however you want to call them), one of type $'a$ and the other of type $'b$.

Implication is straightforwardly translated to the function type: a proof of $A \supset B$ goes from some given assumption A to B , the same way that a function $'a \rightarrow 'b$ takes an argument of type $'a$ and computes a value of type $'b$.

Logic world	Programming world (SML)
formula variables (A, B, C , etc)	type variables (<code>' a</code> , <code>' b</code> , <code>' c</code> , etc.)
conjunction (\wedge)	product type (<code>*</code>)
implication (\supset)	arrow type (<code>-></code>)
disjunction (\vee)	union type (<code> </code>)
true (\top)	unit type (<code>unit</code>)
false (\perp)	empty type ¹

Table 1: Formulas as types

Disjunction represents a choice, which is incorporated by the union type: an option type is either `SOME` or `NONE`, a list is either `[]` (empty) or `x : L` (cons of at least one element), etc.

True is simply a fact that carries no information, so it is interpreted as the unit type.

Finally, false corresponds to the empty type (or bottom type), a type that has no values (and usually not available in programming languages). When it exists, the empty type is used to signal functions that do not terminate correctly (raising exceptions or not terminating).

These relations are summarized in Table 1.

2 Proof terms

Now that we know that formulas can be types, we will redesign natural deduction to annotate formulas with terms (i.e., expressions) of the corresponding type. In particular we will use the new judgment:

$$M : A$$

to denote that the term M has type A . Therefore, the right side of the colon contains the logical connectives and formulas we are familiar with. The rules on these formulas will be the same ones as we know. The left side of the colon contains a term (in typed λ -calculus) of type A . We will now construct these terms.

A good thing to keep in mind is the duality between introduction and elimination rules. Reading them top-down, introduction rules construct formulas from smaller pieces and elimination rules extract the pieces from the formulas. Analogously, introduction rules will construct terms and elimination rules will deconstruct them.

¹Not available in SML. Available in Scala as `Nothing` and in Rust as `never`.

2.1 Conjunction

Conjunction is represented as the product type. So what term in programming would have type 'a * 'b? A pair! So the introduction rule for conjunction is also a rule that takes two terms M and N , of types A and B , and puts them together in a pair structure.

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

There are two elimination rules for conjunction, each conclude one side of the formula. If the term M is a pair of type $A \wedge B$, what is the operation performed on M to get the first element of the pair? What about to get the second? The pieces of a pair can be obtained by `fst` and `snd` operations, therefore, the conclusions of the elimination rules are exactly the application of these operations to term M .

$$\frac{M : A \wedge B}{\text{fst } M : A} \wedge E_1 \quad \frac{M : A \wedge B}{\text{snd } M : B} \wedge E_2$$

2.2 Implication

Implication is the function type, so which term in programming has the function type? Well, a function! Since we do not care about function names, we will represent them anonymously using λ abstractions². The introduction rule for implication takes an A and constructs B , which will be represented by a term, or function, that takes an argument u of type A and passes it to a term M of type B , i.e., $\lambda u.M$.

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

Implication elimination is quite intuitive. If you have a term of type $A \supset B$ (i.e., a function that takes something of type A as an argument and returns something of type B), and you have a term of type A , what do you do? Apply the function! So $\supset E$ is simply function application.

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

²You can think of a $\lambda x.$ abstraction as a `fn x =>` in SML.

2.3 Disjunction

Disjunction is the union type, which is represented in SML by the `|` used in `datatypes`. So if we have a term of type A , how do we construct a term of type $A \vee B$? Let's look at an example. Suppose we have a datatype for users in our system, and the users can be identified either by a name or by a user ID:

```
datatype user = Name of string | UserID of int
```

Now, someone is registering and provided a user name: "aristotle". How do we construct a user from it? We use the type constructor and apply it to the string: `Name("aristotle")`. The operation of applying the type constructor to build a union type is called *injection*. Given a term M of type A , we will use `inl` to construct the union type $A \vee B$ and `inr` if M has type B . The injections will be annotated with the type of the other component in the union (although in programming languages this is completely omitted).

$$\frac{M : A}{\text{inl}^B M : A \vee B} \vee I_1 \quad \frac{M : B}{\text{inr}^A M : A \vee B} \vee I_1$$

The deconstruction of a union type is done by casing on the type constructors (the injections) and extracting the term inside it, so that's exactly what we will do for the disjunction elimination rule. We start with a term M of type $A \vee B$. The other two assumptions in the rule gives us ways to construct some term N or O (possibly different) of type C if we are given terms u and v of types A and B , respectively. When casing on M and deconstructing this union, we have a case for each possible term u or v that produces some term N or O , both of type C . Notice the consistency of the types in the case statement and the scope of the terms u and v .

$$\frac{\begin{array}{c} \overline{u : A} \quad \overline{v : B} \\ \vdots \quad \vdots \\ M : A \vee B \quad N : C \quad O : C \end{array}}{\text{case } M \text{ of } \text{inl } u \Rightarrow N \mid \text{inr } v \Rightarrow O : C} \vee E^{u,v}$$

2.4 True

True is the unit type, which has only one term. We will represent it by $\langle \rangle$ (the fact that this is an empty pair is not a coincidence ;) in the rule $\top I$. There is no deconstruction of unit, the same way that there is no elimination rule for \top .

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

2.5 False

False represents the empty type, which is not supposed to have any inhabitants. So if we constructed a term M of type \perp , we can be sure that M is a nonsense program and we can abort it. An aborted program may have any type C , which will be used to annotate the command.

$$\frac{M : \perp}{\mathbf{abort} \ M : C} \perp E$$

$$\frac{M : A \quad N : B}{\langle M, N \rangle : A \wedge B} \wedge I \quad \frac{M : A \wedge B}{\mathbf{fst} \ M : A} \wedge E_1 \quad \frac{M : A \wedge B}{\mathbf{snd} \ M : B} \wedge E_2$$

$$\frac{\frac{\frac{}{u : A}}{\vdots} \quad M : B}{\lambda u : A. M : A \supset B} \supset I^u \quad \frac{M : A \supset B \quad N : A}{MN : B} \supset E$$

$$\frac{M : A}{\mathbf{inl}^B M : A \vee B} \vee I_1 \quad \frac{M : B}{\mathbf{inr}^A M : A \vee B} \vee I_2$$

$$\frac{\frac{\frac{}{u : A} \quad \frac{}{v : B}}{\vdots} \quad M : A \vee B \quad N : C \quad O : C}{\mathbf{case} \ M \mathbf{of} \ \mathbf{inl} \ u \Rightarrow N \mid \mathbf{inr} \ v \Rightarrow O : C} \vee E$$

$$\frac{M : \perp}{\mathbf{abort} \ M : C} \perp E \quad \frac{}{\langle \rangle : \top} \top I$$

Figure 1: Natural deduction annotated with proof terms.

2.6 Example

All the rules from the previous section are summarized on Figure 1. Let's see them in action now by proving the formula $A \wedge (A \wedge A \supset B) \supset B$. A possible proof for this formula is the following:

$$\begin{array}{c}
 \frac{}{A \wedge (A \wedge A \supset B) \text{ true}}^u \quad \frac{}{A \text{ true}}^w \quad \frac{}{A \text{ true}}^w \\
 \frac{}{(A \wedge A) \supset B \text{ true}}^{\wedge E_2} \quad \frac{}{A \wedge A \text{ true}}^{\wedge I} \\
 \frac{}{B \text{ true}}^{\supset I^w} \quad \frac{}{A \wedge (A \wedge A \supset B) \text{ true}}^u \\
 \frac{}{A \supset B \text{ true}}^{\supset I^w} \quad \frac{}{A \text{ true}}^{\wedge E_1} \\
 \frac{}{B \text{ true}}^{\supset E} \quad \frac{}{A \wedge (A \wedge A \supset B) \supset B \text{ true}}^{\supset I^u}
 \end{array}$$

By using the rules we have just learned, we can transform it in a proof annotated with terms (in red). The types of λ variables are omitted due to space constraints.

$$\begin{array}{c}
 \frac{}{u : A \wedge (A \wedge A \supset B)}^{\wedge E_2} \quad \frac{}{w : A}^w \quad \frac{}{w : A}^w \\
 \frac{}{\text{snd } u : A \wedge A \supset B}^{\wedge I} \quad \frac{}{\langle w, w \rangle A \wedge A}^{\supset E} \\
 \frac{}{\text{snd } u \langle w, w \rangle : B}^{\supset I^w} \quad \frac{}{u : A \wedge (A \wedge A \supset B)}^{\wedge E_1} \\
 \frac{}{\lambda w. \text{snd } u \langle w, w \rangle : A \supset B}^{\supset E} \quad \frac{}{\text{fst } u : A}^{\supset E} \\
 \frac{}{(\lambda w. \text{snd } u \langle w, w \rangle) \text{fst } u : B}^{\supset I^u} \\
 \frac{}{\lambda u. ((\lambda w. \text{snd } u \langle w, w \rangle) \text{fst } u) : A \wedge (A \wedge A \supset B) \supset B}^{\supset I^u}
 \end{array}$$

The term $\lambda u. (\lambda w. \text{snd } u \langle w, w \rangle) \text{fst } u$ on the conclusion is called the *proof term* of this proof. It can be thought of as an abbreviation of how the proof was constructed. Notice that if someone gives you only this term, you are able to reconstruct exactly this proof. There are other proofs for this same formula, but using this proof term you can only reconstruct this one! That is what *proofs as terms* and *formulas as types* is all about :) A term, or a program of a certain type corresponds to a proof of the formula, and checking a proof then reduces to type checking a program.

If you are familiar with λ calculus or with proofs, you might have noticed some redundancy either on that term or in the proof. On the term level, there is a *redex*, i.e., a part that can be simplified, in this case, by function application. On the proof level, there is an implication introduction constructing $A \supset B$ followed immediately by its elimination (remember from local soundness that this can be reduced). These are exactly the same thing: the redex is caused by the redundancy in the proof and vice versa. If we transform the proof, or the term, we get the new annotated proof below:

$$\begin{array}{c}
\frac{}{u : A \wedge (A \wedge A \supset B)} \wedge E_2 \quad \frac{\frac{}{u : A \wedge (A \wedge A \supset B)} \wedge E_1 \quad \frac{}{u : A \wedge (A \wedge A \supset B)} \wedge E_1}{fst\ u : A} \wedge I \\
\frac{\frac{}{u : A \wedge (A \wedge A \supset B)} \wedge E_2 \quad \frac{\frac{}{u : A \wedge (A \wedge A \supset B)} \wedge E_1 \quad \frac{}{u : A \wedge (A \wedge A \supset B)} \wedge E_1}{fst\ u : A} \wedge I}{\langle fst\ u, fst\ u \rangle : A \wedge A} \supset E \\
\frac{\frac{}{u : A \wedge (A \wedge A \supset B)} \wedge E_2 \quad \frac{\frac{}{u : A \wedge (A \wedge A \supset B)} \wedge E_1 \quad \frac{}{u : A \wedge (A \wedge A \supset B)} \wedge E_1}{fst\ u : A} \wedge I}{snd\ u\ \langle fst\ u, fst\ u \rangle : B} \supset I^u \\
\frac{}{\lambda u. snd\ u\ \langle fst\ u, fst\ u \rangle : A \wedge (A \wedge A \supset B) \supset B} \supset I^u
\end{array}$$

Now we have no more redexes and no more ways to simplify the proof. How cool is that? Turns out that normalizing proofs and (typed) λ -terms (or programs) is the same thing. That's Curry-Howard.

Now that we can write proofs more concisely as proof terms, we can rewrite the proofs of local soundness and completeness of the connectives as reductions and expansions of terms.

3 What about a real world function?

The Curry-Howard isomorphism states that one can map programs of type T to proofs of a proposition T . In particular, a program of type $A \supset A$ is mapped to a function $\lambda x.x$, which is just the identity function. At this point one might wonder if all functions of type $A \supset A$ reduce to the identity function. Well, certainly not! Take list sorting for instance. The type of a list sorting function would be $(\text{int list}) \rightarrow (\text{int list})$ and this is definitely not the identity function (unless you live in a parallel world where all lists are sorted). What happened with the isomorphism there?

There are two things going on here. First of all, the isomorphism will work if *no* assumption is made about the types. As soon as you decide that your A is int list , you give some structure to the type and use that structure in your program. Curry-Howard does not work this way. In fact, it can be shown that, if no assumptions are made with respect to A (i.e., if it is only a generic type), any function with type $A \supset A$ can be reduced to the identity function. Neat, no?

The second thing is that the isomorphism, as presented here, works only for a very small notion of programs. These programs are composed of variables, tuples, two constructors, a `case` for pattern matching, function application, and anonymous functions. As such, there is no way to write recursive functions (since functions are anonymous, they cannot be referred back to). That means that all programs written in this language are terminating. Indeed, the type of non-terminating programs is \perp , and if we could derive \perp in our logic, the logic itself would be inconsistent.

In any case, the Curry-Howard correspondence is important because it organizes perspectives. And this was only the first step. After that, people realized that many other

logics can account for many other constructions in programming languages, and that is an active research field. This kind of mapping of formalisms is very useful to look at things from a different perspective and to translate results from one side to another (e.g., typed λ -calculus is normalizing, therefore, so are proofs in our logic). Another result of this kind is the Church-Turing thesis, for example, which relates recursive functions and Turing machines.