MAU11602
Lecture 12
2025-02-18

Laziness

The Curry-Howard correspondence suggests that we consider language features `force` and `delay` with typing rules

$$\frac{e : \tau}{\texttt{delay}\ e : \texttt{unit -> } \tau} \qquad \frac{e : \texttt{unit -> } \tau}{\texttt{force}\ e : \tau}$$

This takes care of the type, and I've implicitly assumed they behave grammatically like functions, but what should their semantics be? In other words, what rules should govern their evaluation?

`force` takes a τ-valued function of a unit argument, which could only be `()`, and gives you a value of type τ. The obvious way to do this is to evaluate the function at `()`.

`delay` is in some sense the inverse. It takes an expression of type τ and gives you a function of `unit` argument which gives a value of type τ. The obvious way to do this is to use the expression to define an anonymous function.

If implemented as above these aren't new language features. We can consider `delay` e as a shorthand for `fn _ =>` e and to consider `force` e as a shorthand for e `()`.

Laziness, continued

Are these useful? Yes. We can use them to delay evaluation of an expression.
In particular, we can delay evaluation until we're sure the value will be needed.
The disadvantage, compared to let expressions, is that the expression gets evaluated every time we `force` it, rather than just once.
We can fix this though. We just need to redefine the rule for `force` so that it evaluates the expression the first time, but remembers the value and skips evaluation whenever its `force`d in the future. In some contexts this can give a large gain in efficiency.
This is referred to as *call by need*, rather than the *call by value* semantics we've been using for expressions so far. It's also often called *lazy* evaluation, contrasted with *eager* evaluation, which tries to reduce all expressions to values immediately.
Why not be lazy all the time?
Most languages are eager by default. Some, but not all, introduce extra features like `delay` and `force` for optional laziness.
It's also possible to make a language lazy by default, which optional eagerness features. Among mainstream general purpose programming languages Haskell is currently the only one which does this.

Curry-Howard again

There are trade-offs in both directions. I've been using eager evaluation so far not because I think it's superior but because it's more widespread.
The choice doesn't affect Curry-Howard directly, since laziness versus eagerness is a reduction question and doesn't affect typing.
Still, if we hadn't already known about laziness, it could have been discovered via the Curry-Howard correspondence by wondering what language constructs correspond to the inference rules for $\top$.
Various useful concepts in type theory and logic have in fact been discovered by asking what the Curry-Howard counterpart of some known notion from the other field is.
There is actually a third field involved, besides type theory and logic, which is category theory, but I won't talk about it here.

Set theoretic interpretation of ZOL

Recall the set theoretic interpretation of zeroeth order logic.
$\wedge$, $\vee$, $\rightarrow$ and $\top$ correspond to $\cup$, $\cap$, $\setminus$ and $\varnothing$, respectively, with the order of operands reversed.
Classical tautologies correspond to set identities, and more precisely to those identities which say a certain set is empty.
We can always write an inclusion of sets as a statement that a relative complement is empty. For example

$$(B \cup A) \setminus B \subseteq A$$

means the same thing as

$$((B \cup A) \setminus B) \setminus A = \varnothing,$$

the set theoretic counterpart of the tautology

$$p \rightarrow (q \rightarrow (p \wedge q)).$$

Set theoretic interpretation of ZOL, continued

This works because we can substitute for p and q any statements which are either true or false, so in particular we can substitute $x \notin A$ for p and $x \notin B$ for q:

$$(x \notin A) \to ((x \notin B) \to ((x \notin A) \wedge (x \notin B))).$$

This is equivalent to each of the following:

$$(x \notin A) \to ((x \notin B) \to (x \notin B \cup A))$$
$$(x \notin A) \to (x \notin (B \cup A) \setminus B)$$
$$x \notin ((B \cup A) \setminus B) \setminus A.$$

The only way this is true for all x is if the set $((B \cup A) \setminus B) \setminus A$ is empty.
Alternatively, we could replace the last of our three lines with

$$x \in (B \cup A) \setminus B \to x \in A,$$

which is the same as $(B \cup A) \setminus B \subseteq A$.

$\bot$

Our set theoretic interpretation for zeroeth order logic has no counterpart for $\bot$, but our classical interpretation and type theoretic interpretations do.
We don't have any special rules of inference for $\bot$ and don't have any typing rules for the empty type but that doesn't mean we can't say anything about them.
For example,

$$(((p \to q) \to q) \to r) \to p \to r$$

is a theorem. In other words, it follows from our rules of inference.

$$(((p \to \bot) \to \bot) \to \bot) \to p \to \bot$$

is then also a theorem.
We don't actually have substitution as a rule of inference but if you take a proof of the first statement and substitute $\bot$ for q and r everywhere they appear then you get a proof of the second statement.
You could do the same with substitution of any expressions for any variables in any theorem, so substitution is what we call a *derived rule of inference*.

$\bot$, continued

It is possible to introduce a special rule of inference for $\bot$, usually called the *principle of explosion*:

$$\frac{\bot}{P}$$

This, for example, allows us to deduce $\bot \to p$, which is certainly a tautology, but which is not a theorem with only our previous rules of inference.
With the previous rules of inference the only tautologies we can prove are the ones, like the one on the previous slide, which remain tautologies when $\bot$ is replaced by a fresh variable.
The type theoretic analogue of $\bot$ is the empty type. The tautology $\bot \to p$ corresponds to a function which, if given an argument of empty type, produces a value of given type.
We can introduce such functions without breaking type safety since there is no expression of empty type to evaluate such a function on, but that's mostly pointless.

An overview of Curry-Howard

Here's a table of the Curry-Howard correspondence, as developed so far:

Logic	(Polymorphic) type theory
statements (propositions)	types
axioms	types of constants
rules of inference	typing rules
proofs	typing derivations
conjunction ($\wedge$)	ordered pairs
disjunction ($\vee$)	(binary) tagged unions
implication ($\rightarrow$)	function definition
truth ($\top$)	the unit type
falsity ($\bot$)	the empty type
theorems	inhabited types
proof checking	type checking
automated theorem proving	type inference

A type is said to be *inhabited* if there is at least one expression of that type.

Negation

I never introduced a symbol for logical negation. The most common choice $\neg$. It has higher precedence than $\wedge$, $\vee$ or $\rightarrow$.
With it we could, for example, simplify

$$(((p \rightarrow \bot) \rightarrow \bot) \rightarrow \bot) \rightarrow p \rightarrow \bot$$

to

$$\neg\neg\neg p \rightarrow \neg p.$$

$\neg$ is redundant, in the sense that $\neg P$ can be expressed by $P \rightarrow \bot$, but it's often used in systems which don't have $\top$ or $\bot$.
It doesn't have any nice counterpart in type theory, and has no counterpart at all in set theory.