Lecture 3

Review the logic of $\Rightarrow$ defined using computable functions

$A \Rightarrow B$ is the type of computable functions whose inputs of type $A$ result in a value of type $B$ in a finitely many steps of computation.

There are such functions in the type $A \Rightarrow (B \Rightarrow A)$ and in the types of the first four of Hilbert’s 1922 axioms.

False is the empty type, say Void. Most programming languages don’t have a Void type. Ours does and we know $False \Rightarrow A$ for any type $A$. As an axiom, this is called $ex\;falso\;quodlibet$. The realizer in our type theory is: $\lambda(x.any(f(x)))$ This gives us the realizer for Hilbert Axiom 5:

$$\begin{align*}
A \Rightarrow (A \Rightarrow False) \Rightarrow B \\
\lambda(x.\lambda(f.any(f(x))))
\end{align*}$$

Computability

Unrealizable types (propositions with no evidence) Some of these are mysteries of mathematics expressed in type theory

We know intuitively how to compute using all of the functions we have used to express evidence, e.g.:

$$\begin{align*}
\lambda(x.x) \quad \text{the identity function} \\
\lambda(f.\lambda(x.\lambda(y.f(y))(x))) \quad \text{for Hilbert 3 (H3)} \\
\lambda(g.\lambda(h.\lambda(x.g(h(x)))))) \quad \text{for Hilbert 4 (H4)} \\
\lambda(x.\lambda(f.any(f(x)))) \quad \text{for Hilbert 5 (H5)}
\end{align*}$$

It will be clear that all of the evidence terms for simple propositional formulas are computable.

The general rule format looks like this: $\lambda(x.b(x))$ realizes some $A \Rightarrow B$ where $x$ is evidence for $A$, $b(x)$ is evidence for $B$. When a value $a$ of type $A$ is supplied for $x$, we know that $b(a)$ computes to evidence for $B$.

The computation rule has the form: $ap(\lambda(x.b(x)) ; a)$ and computes to the value $b(a)$

Here is concrete evidence for $A \Rightarrow (B \Rightarrow A)$. Take for $A$ the concrete proposition ($False \Rightarrow False$). This is a concrete type, not a polymorphic one. It has specific concrete evidence: $\lambda(x.x)$. 

So for \((\text{False} \Rightarrow \text{False}) \Rightarrow (B \Rightarrow (\text{False} \Rightarrow \text{False}))\), we can supply \(\lambda(x.x)\) as evidence for \((\text{False} \Rightarrow \text{False})\). So the realizer for Hilbert 1 (H1), \(\lambda(x.\lambda(y.x))\) can be applied to \(\lambda(x.\lambda(y.x))\) to give \(ap(\lambda(x.\lambda(y.x)); \lambda(x.x))\) which computes to \(\lambda(y.\lambda(x.x))\) as evidence for \(B \Rightarrow (\text{False} \Rightarrow \text{False})\).

We can define another specific proposition if we take \(\text{True}\) to be a constant type with an element. A programming language type that would suffice is Unit. Let’s say it has element \(*\). For \((\text{True} \Rightarrow (B \Rightarrow \text{True}))\) we can compute the realizer \(ap(\lambda(x.\lambda(y.x)); *)\) to obtain \(\lambda(y.x)\).

We will come to understand the precise meaning of other propositions treated as types, e.g. the Goldbach conjectures, say

**Goldbach-1 (G1):** For all even numbers greater than or equal to 6, there are two odd prime numbers \(p_1, p_2\) such that they add up to this even number.

\[
(\forall n : \text{Even}(n \geq 6 \Rightarrow \exists p_1, p_2 : \text{OddPrime}.n = p_1 + p_2))
\]

**Goldbach-2 (G2):** For all odd numbers \(n\) greater than or equal to \(q\), there are three odd primes \(p_1, p_2, p_3\) that add up to \(n\).

\[
(\forall n : \text{Odd}.(n \geq 9 \Rightarrow \exists p_1, p_2, p_3 : \text{OddPrime}.n = p_1 + p_2 + p_3))
\]

We saw that \(G1 \Rightarrow G2\), but that is a numerical truth. A propositional truth is \(G2 \Rightarrow (G1 \Rightarrow G2)\). It is interesting that we do not have any concrete evidence, say \(g1\) for \(G1\) or \(g2\) for \(G2\). Yet we even know things like \(G1 \Rightarrow G2\) and \(G2 \Rightarrow (G1 \Rightarrow G2)\).

Computing with \(\lambda(x.b(x))\) and \(ap(\lambda(x.b(x)); a)\) is simple when \(a\) is a value such as \(*\) or a \(g \in \text{Goldbach-1}\). We simply substitute \(a\) for each occurrence of \(x\) in \(b(x)\) that is “bound by \(x\)”. For example, if \(b(x)\) is \(ap(\lambda(x.x); x)\) we only substitute for the occurrence underlined, not the one in \(\lambda(x.x)\).

It is tricky to compute with expression \(a\) that are not values, say expressions that could have free variables \(x\) in them - symbolic values, such as \((x + 1)\) or \(2 \ast x\), etc.

**Unrealizable types/propositions**

There are some types that have no evidence, or also called unrealizable. For example \(\text{True} \Rightarrow \text{False}\) is not realizable. There is no function \(\lambda(x.\text{exp})\) of this type. If there were, then \(ap(\lambda(x.\text{exp}); *)\) would compute to the value \(\text{exp}(*)\) in \(\text{False}\). Note, we should write \(\text{exp}(*/x)\), meaning substitute \(*\) for all free occurrences of \(x\) in \(\text{exp}\).

On the problem set 1, the proposition \(\sim A \Rightarrow A\) is not realizable for all possible types \(A\). We do not know any computable operation that accomplishes this task. You can say this in your own words after you think about it. Problem 1(f) is hard, just say what you can.
Other propositions/Types

A&B corresponds to what type?
A ∨ B corresponds to what type?

A&B ⇒ A
A&B ⇒ B
A&B ⇒ B&A

A ⇒ (B ⇒ A&B)
((A&B) ⇒ C) ⇒ (A ⇒ (B ⇒ C)) Currying arguments
(A ⇒ (B ⇒ C)) ⇒ A&B ⇒ C Uncurrying

Now comes the subtle case. What does A ∨ B mean? In natural language.
A ⇒ A ∨ B
(A ⇒ C) ⇒ (B ⇒ C) ⇒ ((A ∨ B) ⇒ C)
((A ∨ B) ⇒ C) ⇒ (A ⇒ C) ∨ (B ⇒ C) ⇒ (A ⇒ C) ⇒ (B ⇒ C) ⇒ (A ∨ B) ⇒ C

Fact: can’t define any of &, ∨, ⇒ in terms of the others. Why not?
?
Yes  ~ (A & B) ⇒ ~ A ∨ ~ B

~A ⇒ A  ((A ⇒ False) ⇒ False) ⇒ A
~False ⇒ False
((False ⇒ False) ⇒ False) ⇒ False
(True ⇒ False) ⇒ False
λ(x.x)

Types for &

What is the normal evidence for A&B, for instance: True & ~ False, which is True&(False ⇒ False)? Think of Prime(2) & Prime(3) as more standard examples.

To know A&B is to know evidence a for A and evidence b for B in that order. So it is exactly to “know the pair <a, b> as evidence” or to have the data <a, b>. This semantics intuitively justifies these propositions.
A&B ⇒ A
A&B ⇒ B
A ⇒ (B ⇒ A&B)
A&B ⇒ B&A

To write the evidence mathematically, we need notation for getting at the two elements of the ordered pair. Here are standard mathematical notations:
The meaning or sense of $A \lor B$, $A$ or $B$, is more subtle in our type base semantics. To say that we know $A \lor B$ is to have evidence for either $A$ or $B$ and to know which.

This is a natural type constructor for this idea. It is called a *disjoint union*, written $A + B$. The elements of $A + B$ are either $\text{inl}(a)$, “in left of $a$” or $\text{inr}(b)$, “in right of $b$”

We clearly know propositions like this:

- $A \Rightarrow A \lor B$
- $A \Rightarrow B \lor A$
- $A \lor B \Rightarrow B \lor A$
- $A \& B \Rightarrow A \lor B$

A more delicate question is do we know $A \lor \sim A$?

We saw an instance in lecture 1, $\sqrt{2} \lor \sim \sqrt{2}$ is rational $\lor \sim (\sqrt{2} \lor \sim \sqrt{2}$ is rational). We did not know. Also, Goldbach $\lor \sim \text{Goldbach}$ is an example.