Dependent Types and the Curry-Howard Correspondence

Overview: Two Main Points

- The main point of these lectures will be the so-called Curry-Howard correspondence. The rough idea is that we can identify programs with proofs, and types with propositions.
- The second main point is that it is possible to create systems with "dependent types" where the types are allowed to depend on the return values of functions.

Overview: Two Main Points

- Together, this allows the creation of programming languages where assertions can be proven about the program, in particular it allows for the development of proof assistants.
- We will aim to delve into the surrounding mathematics, making sure that all of this can be developed in a precise, rigorous and sound way.
- Specifically one of the main questions will be: when a proof assistant claims that a proof proves a proposition, why can we trust it?

Dependent Types (Examples)

- We are all familiar with types: In programming languages they are used to signal what an object represents e.g an integer, or string, and so on.
- For example imagine that we have a function that determines if an integer is prime or not. Such a function would have the following type:

```
def isPrime : Nat → Bool
```

- The function itself is what we call a *term*.

Dependent Types

- The innovation in dependent types, is that the type can depend on the values of a *term*.
- For example we can define the *type* of prime numbers, as the set of natural numbers for which `isPrime` returns `true`:

```
def Prime := { n : Nat // isPrime n = true }
```

- While `Nat` is a type, `Prime` is a *dependent type*: it's a type that depends on the value of a term (in this case `isPrime`).

Dependent Types: Applications

- Now, any function with type signature

```
def listPrimes : Nat → List Prime
```

is guaranteed to return a list of natural numbers for which isPrime is always true.

- This is of course not guaranteed to be correct: your implementation of isPrime could be faulty and not always return a prime, you'd need a more refined type to also guarantee the correctness of isPrime.

Dependent Types: Applications

- Here is another example: In most programming languages one can define trees.
- In a dependently typed programming language we can define types that correspond to *sorted trees*, thus any function that returns a sorted tree type also has to prove that the tree is sorted in order for the program to compile.

Curry-Howard Correspondence

- The idea of dependent types really comes into its own when we combine it with the Curry-Howard correspondence, which allows us to see types as Propositions and programs that inhabit the types as proofs of said propositions.
- In particular this allows for alternative (constructive) foundations of mathematics, centered around the concept of types.
- In these foundations we can meaningfully compare proofs of propositions; in classical foundations all proofs have to be identified (by necessity).

Constructive vs Classical

Constructive	Classical
Proofs are programs	Proofs are truth certificates
Different algorithms = different proofs	All proofs identified
A proposition is a type (possibly many inhabitants)	A proposition is a boolean (0 or 1 inhabitants)

Constructive Logic

Statement	Constructive Proof Must Provide
$A \vee B$	Which disjunct holds + proof of it
$\exists x.P(x)$	A witness x + proof of P(x)
$A \rightarrow B$	A function transforming proofs of A into proofs of B

Proofs are algorithms, and all algorithms are not the same!

Classical Logic

Statement	Classical Proof Must Provide
$A \vee B$	Truth certificate that $A \vee B$ holds (no indication which)
$\exists x.P(x)$	Truth certificate that some x exists (no witness)
$A \to B$	Truth certificate that A implies B (no function)

Extensional vs Intensional

- Classical logic has an extensional view of computation and functions. A function is just defined by what input is mapped to what output.
- The sentiment here is "it doesn't matter how X is computed, it only matters that it *can* be computed".

Extensional vs Intensional

- In constructive logic we have an intensional view of computation: we can peek "inside" a function and see how it computes. A function is fundamentally identified with an algorithm.
- The sentiment here is "we care about how X is computed, for example, some ways are more efficient than others".

Extensional vs Intensional

- We are all familiar with the extensional point of view, but much less so with the intensional way, which in effect requires us to see functions differently, as *algorithms*.
- This point of view of functions as *computations* rather than plain *maps* is captured by λ-calculus which we will develop next.
- The λ-calculus is a way to express computation, and it will be the basis for everything that we do next (in particular Lean is heavily dependent on λ-calculus).

Models of Computation (Motivation)

- There is no exact definition of what is a computation (in the same way that there is no exact definition of what is randomness).
- However it is broadly understood that a computation is something that takes an input, performs a number of agreed upon operations on the said input and then gives us an output.

Models of Computation: Example

- For example addition of two integers is an example of a computation.
- We have two inputs, a black-box to which the input is connected, and the output is the sum of the two numbers passed in the input.
- There is thus an encoding of an input, a sequence of steps (an algorithm) that runs a predetermined set of operations on this input, and a decoding of the output.

Models of Computation: Conventions

- We see that for a computation, conventions are important: we have to agree on an encoding and decoding of what the input and output of the "black-box" means.
- Of course not all computations can be run on our simple black-box that simply adds two numbers. For example, we obviously cannot use it to exponentiate or even multiply two numbers.

Universal Computation

- One can therefore ask: is there a "most expressive" computation, a computation such that all other computations can be performed inside of it?
- One immediately thinks that a computer would have to be "it".

Universal Computation

- A computer clearly fits into our model of a "computation". The input is computer code stored on a hard-drive.
- The input is then executed, loaded into RAM memory, there is a program counter that fetches an instruction at an address in RAM, executes the instruction, and then increments the program counter and repeats.
- The simplest mathematical model of a computer is a Turing machine. A variant of a Turing machine is captured by a programming language called "brainfuck".

Brainfuck (Turing Machine Model)

Character	Instruction Performed
>	move to cell to the right
<	move to cell to the left
+	increment value at cell modulo 256
-	decrement value of cell modulo 256
.	output byte at current cell
,	input byte and store in current cell
[	if cell byte is zero jump until next]
]	if cell byte is non-zero jump to previous [

What a Computer Really Is

Fundamentally this is very simple:

- we can write to a memory location
- read from a memory location
- jump to a new address based on the memory location
- accept input and write it to a memory location
- print output based on memory

That's all you need for a computer. This is basically what a Turing machine is.

Church-Turing Thesis

- The Church-Turing thesis is that all possible computations can be performed by a computer, in other words that a computer is the most universal "computation" there is.
- This is a philosophical statement and it cannot be proven.

Hello World in Brainfuck

```
++++++++[>++++[>++>+++>+++>+
<<<<-]>+>+>->>+[<]<-]>>.>---
.+++++++..+++.>>.>-.<<-.<-.
+++.------.--------.>>+.>++.
```

Lambda Calculus (Syntax)

- Lambda calculus is another model of computation which can be proven to be as expressive as the Turing machine one. It was developed by Alonzo Church.
- This model recognizes that there is something superfluous about a Turing machine. Why do we need an extra RAM component in which we store the state of the computation?

Lambda Calculus

- Why can't we just get the output by repeated direct transformation of the input?
- Since we aim to only transform the input to an output (without an intermediate notion of RAM memory or state) everything will be expressed in terms of functions.

Lambda Calculus

- This model of computation is the foundation of *functional programming languages* (e.g Lisp, Ocaml, Haskell, Lean, Agda, and so on).
- *Imperative programming languages* (e.g C, C++, Java, Python, Rust, and so on) are modelled after a Turing machine.

Lambda Terms: Variables

- We now describe the set Λ of valid expressions in Lambda calculus. These are called *lambda terms*.
- First there are variables, we can agree that any alpha-numeric symbol such as `a`, `b`, or `a1` is a valid variable. We call this set V.
- If $a \in V$ then $a \in \Lambda$. In other words all variables are valid Lambda terms.

Lambda Terms: Application

- Second there is composition of Lambda terms, if $M \in \Lambda$ and $N \in \Lambda$ then MN is also a valid lambda term, that is $MN \in \Lambda$.
- Note that we have not assigned any *meaning* to MN we are just asserting that this is *valid* term, i.e a valid encoding, if you wish.
- Thus for example if x is a variable then xx is a valid λ-term.

Lambda Terms: Abstraction

- Third, there are so-called λ-abstractions. If u is a variable, i.e $u \in V$ and M is a valid lambda term i.e $M \in \Lambda$ then

$$\lambda u.M$$

  is a valid lambda-term.

- Again we have not specified the meaning of this. We are just defining valid encodings.

Lambda Terms: BNF Grammar

- This set of valid encodings is usually succinctly written as

$$\Lambda = V \mid \Lambda\,\Lambda \mid (\lambda V.\Lambda)$$

- Anything in V is a valid term, if you have two valid terms M and N then MN is also a valid term, and if you have a variable a and a valid term M then $\lambda a.M$ is also a valid term.

Lambda Terms: BNF Grammar

- We have thus defined the set of valid λ-terms but we have not assigned any meaning to them.
- This is similar to saying that anything composed of letters between a and z gives you a "word", but the "word" might have no meaning yet!

Lambda Terms: Examples

These are all valid λ-terms:

$$\lambda x.xy$$

$$(\lambda x.xy)(\lambda z.x)$$

$$xx(\lambda x.z)$$

Beta Reduction (Semantics)

- In fact the meaning of the lambda term will have to be a matter of our interpretation, we will have to decide that such and such λ-terms means something to us. The assignment of meaning will be up to us!
- What will not be up to us, however, is how we perform computations once we are given a λ-term.

Beta Reduction (Semantics)

- We will now specify how computation is performed on a set of valid λ-terms. This is thus the second side of the picture, the rules of performing computations on a valid λ-term.

The β-Reduction Rule

- Essentially computation is performed by a process called β-reduction. The only rule is that when we see a term of the form

$$(\lambda x.M)N$$

then this evaluates to M with every occurrence of x inside M replaced by N.

β-Reduction Examples

$$(\lambda x.xy)u \longrightarrow uy$$

$$(\lambda x.xx)(\lambda x.xx) \longrightarrow (\lambda x.xx)(\lambda x.xx)$$

The second example evaluates to itself — this is an infinite loop, this computation never terminates.

Encodings: Booleans

- We can first declare that we encode *true* and *false* as

$$\mathsf{TRUE} = \lambda x.\lambda y.x$$

and

$$\mathsf{FALSE} = \lambda x.\lambda y.y$$

Encodings: IF-THEN-ELSE

- We can encode IF and ELSE using

$$\text{IF } b \text{ THEN } t \text{ ELSE } e = \lambda b.\lambda t.\lambda e.b\, t\, e$$

- This is a function that takes three arguments `b`, `t` and `e`. If `b` happens to be `TRUE` it returns `t` otherwise `e`.

Verifying IF-THEN-ELSE

- Does this work? Let's check that

$$\text{IF TRUE THEN } t \text{ ELSE } e$$

returns `t`. By definition:

$$\text{IF TRUE THEN } t \text{ ELSE } e = (\lambda b.\lambda t'.\lambda e'.b\, t'\, e')\ \text{TRUE}\ t\, e$$

Verifying IF-THEN-ELSE

This is then

$$(\lambda t'.\lambda e'.\mathsf{TRUE}\ t'\ e')t\,e = (\lambda e'.\mathsf{TRUE}\ t\,e')e = \mathsf{TRUE}\ t\,e$$

And by definition of `TRUE` this is

$$(\lambda x.\lambda y.x)t\,e = (\lambda y.t)e = t$$

The check for FALSE is the same.

Church Numerals

- Can we encode integers? Yes, define them as follows:

$$0 = \lambda f.\lambda x.x$$

$$1 = \lambda f.\lambda x.f\, x$$

$$2 = \lambda f.\lambda x.f\, (f\, x)$$

$$3 = \lambda f.\lambda x.f\, (f\, (f\, x))$$

ISEVEN Function

- Can we encode a function that given an integer as input will return true or false depending on whether the integer is even?

$$\begin{aligned} \text{TRUE} &= \lambda x.\lambda y.x \\ \text{FALSE} &= \lambda x.\lambda y.y \\ \text{NOT} &= \lambda b.b\ \text{FALSE}\ \text{TRUE} \end{aligned}$$

ISEVEN Function

Our parity detecting function is:

$$\text{ISEVEN} = \lambda n.n \text{ NOT TRUE}$$

Or in fully expanded form:

$$\text{ISEVEN} = \lambda n.n\,(\lambda b.b\,(\lambda x.\lambda y.y)\,(\lambda x.\lambda y.x))\,(\lambda x.\lambda y.x)$$

Verifying NOT

- If we look at `NOT` we should have `NOT FALSE = TRUE` and `NOT TRUE = FALSE`. Let's check:

$$\begin{aligned} \text{NOT FALSE} &= (\lambda b.b\ \text{FALSE TRUE})\ \text{FALSE} \\ &= \text{FALSE FALSE TRUE} \end{aligned}$$

Verifying NOT

And this equals:

$$(\lambda x.\lambda y.y)\ \text{FALSE}\ \text{TRUE} = \text{TRUE}$$

It works!

Why ISEVEN Works

- Recall that an integer n is a function that takes a function f and an input x, and then applies the function f n times to the input x.
- We therefore get:

$$\text{ISEVEN } n = n \text{ NOT TRUE}$$

Why ISEVEN Works

- Now `n NOT` means that we will apply the `NOT` function to TRUE n times.
- This corresponds to bit flipping. If n is even we will flip back to true, and if n is odd we end with false.

Why ISEVEN Works

Let's check formally:

$$\begin{aligned} n \text{ NOT TRUE} &= (\lambda f.\lambda x.f^n x) \text{ NOT TRUE} \\ &= (\lambda x.\text{NOT}^n x) \text{ TRUE} \\ &= \text{NOT}^n \text{ TRUE} \end{aligned}$$

We know `NOT TRUE = FALSE` so this is `NOT^(n-1) FALSE`. You can finish with a proof by induction.

Other Operations

There are other important operations; checking that these work is left as an exercise:

$$\mathsf{SUCC} = \lambda n.\lambda f.\lambda x.f\,(n\,f\,x)$$

$$\mathsf{PLUS} = \lambda m.\lambda n.\lambda f.\lambda x.m\,f\,(n\,f\,x)$$

$$\mathsf{MULT} = \lambda m.\lambda n.\lambda f.m\,(n\,f)$$

Brainfuck Multiplication Comparison

Compare the elegant lambda calculus definition:

$$\text{MULT} = \lambda m.\lambda n.\lambda f.m\,(n\,f)$$

with the equivalent in brainfuck, which needs multiple slides...

Brainfuck Multiplication: Input

```
,           Read a into cell 0
>,          Read b into cell 1
<           Back to cell 0

            Memory: [a] [b] [0] [0]
```

Brainfuck Multiplication: Main Loop

```
[               While a != 0
  -               a--
  >               Move to b
  [->+>+<<]       Copy b → result and temp
                  Memory: [a-1] [0] [result+b] [b]
  >>              Move to temp
  [-<<+>>]        Move temp → b (restore b)
                  Memory: [a-1] [b] [result+b] [0]
  <<              Back to b
  <               Back to a
]
                  Memory: [0] [b] [a×b] [0]
```

Brainfuck Multiplication: Output

```
>>              Move to result
.               Output
```

- Compact form:

```
,>,<[->[->+>+<<]>>[-<<+>>]<<<]>>.
```

Evaluation Order and Church-Rosser

- A few theoretical questions raise themselves rather quickly.
- Nothing prescribes in what order we should perform β-reduction. We could start by reducing the innermost function first, or reduce the outermost functions first.
- Could it be that we receive different results?

Evaluation Order: Example

Consider:

$$(\lambda x.y)\,((\lambda z.z\,z)\,(\lambda z.z\,z))$$

- If you reduce the argument first (the inner redex) you will be stuck in an infinite loop, since $(\lambda z.z\,z)\,(\lambda z.z\,z)$ reduces to itself.
- If you reduce the outermost function first, you simply get y (since $\lambda x.y$ ignores its argument).

Evaluation Order Matters

- So the order of evaluation matters!
- Now it's one thing to get stuck in an infinite loop, but could we actually get different "final" values?
- If that were the case we would have to throw the entire concept out. What is in fact a "final" value?

Church-Rosser Theorem

- The Church-Rosser theorem will give us some answers to these questions.
- We can always reduce to a common value. This will be clarified in the next lecture.

Lazy vs Strict Evaluation

- The order of evaluation is important from a practical point of view.
- Evaluation from the outside (outermost first) leads to *lazy* programming languages (Haskell) and allows working with infinite structures.
- Evaluation from the inside (innermost first) leads to *strict* evaluation (OCaml) and has more predictable performance.

Why Types? (Eliminating Divergence)

- The discussion of evaluation order introduces the idea of types into the picture.
- We would like to further constrain our λ-calculus by introducing *types* to eliminate non-terminating evaluations.

Why Types? (Eliminating Divergence)

- Our first attempt will be in fact too constraining (we will throw away too many possible programs).
- But it will be an introduction to the basic idea of this course: we will see a basic instance of the Curry-Howard isomorphism.

Preview: Simply Typed Lambda Calculus

- In the next lecture, we will introduce the simply typed lambda calculus (STLC).
- The key idea is to assign types to terms in such a way that only "well-behaved" programs are allowed.

Preview: Simply Typed Lambda Calculus

- A remarkable fact: every well-typed term in STLC terminates! This is called the "strong normalization" property.
- The price we pay: some useful programs (like general recursion) become impossible to express.
- This tradeoff between expressiveness and safety will be a recurring theme.

The Curry-Howard Correspondence (Preview)

Type Theory	Logic
Type	Proposition
Term of type A	Proof of proposition A
Function type $A \to B$	Implication $A \to B$
Pair type $A \times B$	Conjunction $A \wedge B$
Sum type $A + B$	Disjunction $A \vee B$

This correspondence is the foundation of proof assistants like Lean, Coq, and Agda.