Lecture 9: Models of Computation

In the beginning of the class, we have learned a lot about logics and derivations. Some of the the students might have wondered, what is this all about. Why do we need to worry about logic in a discrete math class? In this lecture, I hope to show you the answer: Logic and computation are very closely related, so close in fact, that one could almost say that they are the same thing. Logic is the foundation for everything that has to do with computation. My goal for this lecture is to show you a little bit more about what the two have in common: You can actually think of proofs as programs!

Recall from lecture 2, the three rules defining conjunction:

$$\frac{A\ \mathsf{true} \quad B\ \mathsf{true}}{A \wedge B\ \mathsf{true}}\,\wedge\mathrm{I} \qquad \frac{A \wedge B\ \mathsf{true}}{A\ \mathsf{true}}\,\wedge\mathrm{E}_1 \qquad \frac{A \wedge B\ \mathsf{true}}{B\ \mathsf{true}}\,\wedge\mathrm{E}_2$$

Let's look at the introduction rule $\wedge$I first. It says, that if we have two proofs, one for A true and another for B true, then we can construct (using the rule) a proof that $A \wedge B$ true, as well.

Let's make A and B concrete, that is A says that 3 is a natural number, and B says that 5 is a natural number. Using the rule above, we can deduce that 3 and 5 are both natural numbers or more explicitly, that

$$3 \text{ is a natural number} \wedge 5 \text{ is a natural number}$$

Derivations are extremely unwieldy constructs. They are big, take a long time to write, and seem to have nothing to do with programming, unless ... If we only had a more concise way to talk about these derivations, even if it is at the expense of precision, we could use them to program as we will see. In short, programs are shortened and compactified proofs.

If we just pick n itself to be a program that corresponds to the proof that n is a natural number, then it all that needs to be done, if we ought to write a program that both are numbers, is a pair, such as: $(3, 5)$.

Let's address the elimination rules: $\wedge\mathrm{E}_1$ and $\wedge\mathrm{E}_2$. Those rules allow us to take a conjunction apart. Let's denote the proof for $A \wedge B$ true withe e. In the one case, we want to write out the program that correspond to the proof of A, in the other the program that corresponds to the proof of B. This can be easily done, we either need to take the first or second argument, for which we write "fst e" or "snd e".

This means, that if we write fst $(3, 5)$ we actually refer to the proof that fst $(3, 5)$ stands for a derivation that 3 is a natural number.

Intuitively, it should be clear that "fst $(3, 5)$" should be considered equivalent to 3 itself, fst $(3, 5)$" is a program that can still be evaluated, or simplified, and the result of taking such a computation step is 3. This gives rise to the definition of a *redex*. Whenever "fst" is followed by a pair, then we call it a redex, and we can always take a computation step and replace the the program by its first argument.

These steps of computation are extremely important, because we will have to count them to talk about the complexity of a function.

It is easy to see that it is always possible to reduce a redex:

$$\dfrac{\dfrac{\begin{array}{c}e\\A\ \mathsf{true}\end{array}\quad B\ \mathsf{true}}{A \wedge B\ \mathsf{true}}\wedge\mathrm{I}}{A\ \mathsf{true}}\wedge\mathrm{E}_1$$

can always reduce to

$$\begin{array}{c}e\\A\ \mathsf{true}\end{array}$$

because this proof is simply contained as a subproof in the proof above. A similar observation holds for $\wedge\mathrm{E}_2$ of course.

Therefore, conjunction gives us a very natural way of pairing and computing first and second projection. And this is very close related to the way we package up arguments in lists that are then passed into a method, let's say in programming language like Java or C#.

This brings us two the next big question. Is there some corresponding similarity between methods and functions and proofs? Yes, there is, all we have to do is to look at our understanding of implication and interpret the rules as programs. Recall the two rules from the second lecture.

$$\dfrac{\begin{array}{c}\overline{}\,u\\A\ \mathsf{true}\\\vdots\\B\ \mathsf{true}\end{array}}{A \Rightarrow B\ \mathsf{true}}\Rightarrow\mathrm{I}^u \qquad \dfrac{A \Rightarrow B\ \mathsf{true}\quad A\ \mathsf{true}}{B\ \mathsf{true}}\Rightarrow\mathrm{E}$$

Let's take $\Rightarrow$ E first. We can think of this rule as passing an argument to a function. Let's say that f represents the the proof of $A \Rightarrow B$ and e the proof of A, then we could read it as "send argument e to method f", written as a juxtaposition: $f\ e$ (in words f space e).

The last rule that we need to discuss is that of $\Rightarrow$ I. For students that is perhaps the most difficult rule in the bunch, because it introduces a fresh hypothesis A (only visible in the premiss) named u. The rule says, in plain English, that if under the assumption u that A holds, we can infer that B holds, then the implication $A \Rightarrow B$ also holds. Perhaps you can already see the connection, the program representing the proof of B, let's call it e' for the lack of a better name, is actually the body of a function/method. The assumption u plays the role of the parameter that may occur free in the body of the program. For example, a method that adds the value of a quarter (25) to some other amount u is written in Java as

```
int addQuarterToAmount (int u) {
   return (u+25);
}
```

is such a function. As for $\Rightarrow$ E, we need some kind of a program constructor that corresponds to $\Rightarrow$ I. If e' witnesses the derivation of B under the assumption A (which we called u), then $\lambda u.\, e'$ is a the witness for the whole derivation. Using λ, we write `addQuarterToAmount` as

$$\lambda u.\, u + 25$$

Here is one more example:

$$\lambda u.\, \lambda v.\, u$$

is a witness of the proof that a $A \Rightarrow B \Rightarrow A$. The λ symbols is pronounced lambda, it a Greek letter, and the only Greek letter that we will be using in this class. Because of the choice of letter, this calculus is also called the λ-calculus.

What is one step computation in this setting? It is just executing the application of a function to an argument. Consider the program

```
addQuarterToAmount (10)
```

What you do is replace all occurrences of `u` in the definition of the method by 10 and continue. The next step will be to compute `return(10+25);`. This replacement operation is also called substitution, because we substitute the number `10` for `u` anywhere in the body of the method.

Analogously, when we find a *redex* $(\lambda u.\, u+25)(10)$, we need to replace 10 for all occurrences u, for which we also write $[10/u](u + 25) = 10 + 25$. We always think of substitutions to take place instantaneously. This is justified because it can be implemented this way. And mathematical operations such as $10 + 25$ count also as one step. Let's check that this actually makes sense.

$$\dfrac{\dfrac{\dfrac{}{A\ \mathsf{true}}\,u}{\begin{array}{c}e'\\ B\ \mathsf{true}\end{array}}}{A \Rightarrow B\ \mathsf{true}}\Rightarrow \mathrm{I}^u \qquad \begin{array}{c}e\\ A\ \mathsf{true}\end{array}}{B\ \mathsf{true}}\Rightarrow \mathrm{E}$$

can always reduce to

$$\begin{array}{c}[e/u]e'\\ B\ \mathsf{true}.\end{array}$$

In summary, we have defined three useful computation (or better reduction) steps, that define the operational behavior of programs.

$$\begin{array}{rcl} \mathrm{fst}(e_1, e_2) & \rightarrow & e_1 \\ \mathrm{snd}(e_1, e_2) & \rightarrow & e_2 \\ (\lambda u.\, e')\ e & \rightarrow & [e/u]e' \end{array}$$

If we are interested about the running time of an algorithm or a program, we simply need to count these steps . But enough of the theory already, let's look at some examples.

Example 47 (Delegates in C#) Consider the following code in the C# language. If you don't know C#, just ignore this example.

```
1  public delegate int SimpleDelegate(int x);
2
3  class TestDelegate
4  {
5    public static int MyFunc(int x)
6    {
7      Console.WriteLine("I was called by delegate ...");
8      return x;
10   }
11
12   public static void Main()
13   {
14     // Instantiation
15     SimpleDelegate simpleDelegate = new SimpleDelegate(MyFunc);
16
17     // Invocation
18     Console.WriteLine(simpleDelegate(5));
19   }
20 }
```

Line 1. declares a new argument "type" for methods that may be used to pass other methods around. In our more logical formalism we woud say that we define the type `SimpleDelegate` to stand for nat $\Rightarrow$ nat. Next, in lines 5-10, we define a method that takes x as an argument and returns x, and this method is called `MyFunc`. In our notation, `MyFunc` $= \lambda x.\, x$ ignoring the content of line 7. In the `Main` method, line 15, we turn then `MyFunc` into an official method, which we then call `simpleDelegate`, which for us is just a different name for `MyFunc`. In line 18, we then apply `simpleDelegate` to the number 5, which, is similar to calling `simpleDelegate`$(5) = (\lambda x.\, x)(5) = 5$. In summary, λ-terms are just delegates!

The next examples illustrates that functions are just values, that can be passed to other functions as arguments. This is an important observation. Modern programming languages take advantages of this fact more and more frequently, for example $C\#$ and $F\#$. In the beginning this fact usually feels a bit funny, but it is normal. It will pass over time.

Example 48 (Program that corresponds to an earlier proof) Do you recall the proof of $(A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$? We can write this easily as program as follows:

$$\lambda u.\, \lambda v.\, \lambda w.\, u\ w\ (v\ w)$$

Example 49 (Reduction sequence) Let's consider the term from above applied to two arguments.

$$(\lambda u.\, \lambda v.\, \lambda w.\, u\ w\ (v\ w))\ (\lambda x.\, \lambda y.\, x + y)\ (\lambda z.\, z + 25)\ 42$$

. When we apply the reduction rules from above in a particular order, we obtain.

$$
\begin{aligned}
&(\lambda u.\, \lambda v.\, \lambda w.\, u\ w\ (v\ w))\ (\lambda x.\, \lambda y.\, x + y)\ (\lambda z.\, z + 25)\ 42 \\
&= (\lambda v.\, \lambda w.\, (\lambda x.\, \lambda y.\, x + y)\ w\ (v\ w))\ (\lambda z.\, z + 25)\ 42 \\
&= (\lambda w.\, (\lambda x.\, \lambda y.\, x + y)\ w\ ((\lambda z.\, z + 25)\ w))\ 42 \\
&= (\lambda x.\, \lambda y.\, x + y)\ 42\ ((\lambda z.\, z + 25)\ 42) \\
&= (\lambda x.\, \lambda y.\, x + y)\ 42\ (42 + 25) \\
&= (\lambda x.\, \lambda y.\, x + y)\ 42\ 67 \\
&= (\lambda y.\, 42 + y)\ 67 \\
&= 42 + 67 \\
&= 109
\end{aligned}
$$

No matter, in which order we apply reduction steps, we will always end up with the same result. This theorem can be proven, of course by induction, but it would go to far to cover this in class. Turning our interest to complexity, we can count that this computation took 9 steps. In certain situations we need to make a choice of where to apply a reduction rule. More precisely, we had only 1 way to reduce the first step, two ways to reduce the second, three ways, to reduce the third, etc. Can you find which ones?

The programs that we have introduced in this lecture are of mathematical nature. At the beginning of the lecture, we took the freedom and abstracted "3 is a natural number" to "nat". We can further abstract, drop any distinction between all types, and arrive this way a calculus that seems to be able to express any function, that we can write out on paper and compute with. This conjecture is also known as the Church Turing thesis. It has not and probably never will be formally proven.

Lastly, we direct our interest back to the induction principles that also seem to provide a form of computation. Recall the induction principle for natural numbers. Recall that if we have proven the base case

$$P(0)$$

and the step case

$$\forall n \in \mathbb{N}.P(n) \to P(n+1)$$

then we effectively can compute $P(n)$ for any n, by starting with the base case and iterating the step case n times:

$$P(0), P(1), P(2) \ldots P(n)$$

Let e_0 be a proof of $P(0)$ and e_1 the proof that for $\forall n \in \mathbb{N}.P(n) \to P(n+1)$. As above, let's forget about the little n in the index, which allows us to simplify the use universal quantification to implication. This is completely fine, because the "type systems" of modern general purpose programming languages are not

expressive enough to capture the indices any way. If $P(n)$ stands for $n!$, then a good approximation of $P(n)$ is nat, since n is a natural number. Consequently $e_0 = 1$ represents nat and $e_1 = (\lambda n.\, \lambda x.\, n \times x)$ represents nat $\Rightarrow$ nat $\Rightarrow$ nat.

$$\begin{aligned}
!5 &= e_1\ 5\ (e_1\ 4; (e_1\ 3\ (e_1\ 2\ (e_1\ 1\ e_0))))\\
&= e_1\ 5\ (e_1\ 4; (e_1\ 3\ (e_1\ 2\ (e_1\ 1\ 1))))\\
&= e_1\ 5\ (e_1\ 4; (e_1\ 3\ (e_1\ 2\ 1)))\\
&= e_1\ 5\ (e_1\ 4; (e_1\ 3\ 2))\\
&= e_1\ 5\ (e_1\ 4; 6)\\
&= e_1\ 5\ 24\\
&= 120
\end{aligned}$$

Altogether, this took about 15 steps. It seems that the two program e_0 and e_1 can define the result of $!n$ for any n. All we have to do is iterate the e_1 n times. These kind of programs should look familiar as they define basically exactly the recursive functions that we have encountered earlier in this class. They scale to any other induction principle, even those that range over more complex data structures, such as lists and trees. As an example, we look the definition of the factorial numbers.

$$n! = \begin{cases} e_0 & \text{if } n = 0\\ e_1\ n\ (n-1)! & \text{otherwise} \end{cases}$$