Cheet Sheet and Explanation (Sort Of)

Kristin McClellan

September 15, 2020

- λ <- that is the Greek letter lambda, pronounced "lamb" like the goat followed by duh
- Λ <- that is the Greek letter Lambda only capital this time
- λ-calculus: a way of modelling computation (how "powerful" does a programming language have to be before it can compute anything)
- The whole point of this paper is to show why working with λ-calculus sucks.
  - The whole point of λ-calculus is searching for stuff and replacing it without breaking things. It turns out, it's really hard to not break things.
  - Replacing things changes the original thing and it's really hard to adjust to the changes.
- λ-term: the building blocks of λ-calculus. They are elements of λ-calculus just like how 1, 2, and 3 are elements of the numbers.
- All λ-terms are written in capital letters whereas the things that make them up (variables) are written in lowercase letters.
- λ-terms may be made up of other λ-terms. For example, the λ-term M may contain the λ-term N.
- The very top of page two says, "A λ-term may be any one of the following: a variable (written as the lowercase x), two λ-terms together (which we call application), or a variable binding a λ-term (which we call abstraction)."
- Whenever you see the word set, you can think of it as a list or collect e.g. saying λ is the set of all λ-terms" is the same as λ is the collection of all λ-terms"
- Induction is a way to look at each part of a λ-term one at a time.
  - Structural induction does this by going through all the cases laid out by the equation at the top of page 2
  - Measure induction does this by figuring out how complex the λ-term. It gives each term a positive whole number value (0, 1, 2, and so on) with higher numbers meaning more complex. Then it starts with the easy (low complexity) ones that it knows how to solve and works its way up to the hard ones. By the time it gets to the hard ones, it has so much experience from the easy ones that it knows how to solve the hard ones.
    - Rank is how it assigns each term a number.
    - Rank induction and measure induction are the same thing only with different names.

- Capture-avoiding substitution is how you do algebra on λ-terms. You know how doing the same thing to both sides of an equation doesn't change the result? Capture-avoiding substitution is the same thing but for λ-terms. It's like changing

$y = x + 1$ to $n = m + 1$; no change to the result but some letters might change.

- Free and fresh variables are how we figure out how to change the term without changing the result. See how in $y = x + 1$ to \$n=m+1, we can change the x and y? Those are the free variables. We replace them with fresh variables (variables that haven't been used before) m and n.
- We say a variables is free if changing it to something else won't screw anything up. We can change the variables x and y because they're just names but changing the 1 to 2 is going to screw everything up. Then x and y are free and 1 is bound (not free). That's what 4.1 Free and Fresh Variables is saying.
- 4.2 Capture-Avoiding Substitution is telling you how exactly you go about changing things without screwing everything up. It's also a pain in the ass to get right which is what the first paragraph is saying.
  - $M[x := N]$ says "look through M and replace any x you find with N (making sure you don't screw anything up)."
  - Case 1a says "if you find an x that is easy to replace, just do it"
  - Case 1b says "if you don't find an x, don't do anything"
  - Case 2 says "if you see two terms next to each other (remember that we call this application), start looking through both of them individually and go from there"
  - Case 3a says "if you find an x that you can't change, don't mess with it. If you do, you'll ruin everything. So don't."
  - Case 3b says "if you're looking for an x but find a bound y, don't touch the y but keep looking for x."
  - Case 3c is the one that makes this a pain. It says "if you find a free y, turn every y you find into something else (z in this case), then start looking for x again."
- 5 Equivalence of Terms shows how we figure out if any two λ-terms are equal.
  - Just like how $y = x + 1$ is the same as $n = m + 1$, we say $\lambda x.x$ is the same as $\lambda y.y$.
  - Essentially, the terms are equal if you can swap out the variable names of the first term and end up with the second.
  - If that doesn't make sense to you, a more intuitive version might be "if you squint hard enough so that you can't read the letters, are the shapes of the terms the same?" That's what the first sentence is saying.
  - Everything else in the section is laying out the groundwork of how exactly that is done. It basically says "any two terms are equal if every individual part of them is equal."
- Section 6 is illustrating why actually using λ-calculus sucks. It starts off with an equation that essentially reads "if you search for any replaceable x but don't find any, you end up with the same thing as when you started." For example, let's say we wanted to paint some rooms of our house. We want to paint the blue rooms of our house purple. We don't have any blue

rooms so we end up with the same house. In this analogy, the thing we're searcing *through* (the house) is M. The thing we're trying to *find* (blue rooms) is x. The thing we're going to replace it with (purple rooms) is N.

  - After that, we try to prove it with structural induction (structural induction is kind of explained up above). This is all the nitty gritty but each case boils down to one of the following three things: we didn't find x so we don't need to replace anything so it's easy (case 1a, 3a), we found x but replacing it would screw everything up so we don't do that and that makes it easy (case 1b, 2, 3b), or we found x, we can replace it, we need to replace it, but replacing it is really hard so we get sad (case 3c).
  - It's hard to replace everything in case 3c because we change the λ-term before we are actually done looking at it. Essentially, we'd have to look at it again to figure it out but the technique we're using (structual induction) can't handle that.
  - Then we try again, this time with rank induction (kind of explained up above). We start by saying (and proving) a theorem that says "changing the names of variables doesn't change the rank of a λ-term (how complex it is) so this attempt won't break in the same way the last one did." Then we actually do the proof. It's easy now that we have look at how hard/complex the terms are.
  - The paragraph after that quickly goes through an alternate way of doing things.

- 7 Syntactic Alternatives presents ways to answer the question "how can we make that suck less?"
  - A way to do that is with de Bruijn (duh-broin, rhymes with the-groin) indices (plural of index, in-deh-seas)
  - It says "stop screwing around with variable names and just give everything a number"
  - It solves the "search and replace x" problem described above by saying "if you see the number you're trying to find, just do it. Stop messing around with names. You found it. Just do it already."
  - Everything past that is the nitty gritty.