Notes from: Type Theory and Formal Proof, Chapter 2

Notes by Donald Pinckney

May 29, 2019

Note that all of the definition, theorem, etc. numbers correspond to the numbers in the textbook.

1 Lambda-Terms

The goal of the first chapter is to view and understand the simplest and most abstract features of *functions*. Let $V = \{x, y, z, \cdots\}$ be an infinite set of *variables*. We define the set Λ of all λ-terms as follows.

Definition 1.3.2 (The set Λ of all λ-terms)

1. (Variable) If $u \in V$, then $u \in \Lambda$.
2. (Application) If $M \in V$ and $N \in V$, then $(MN) \in \Lambda$.
3. (Abstraction) If $u \in V$ and $M \in \Lambda$, then $(\lambda u.M) \in \Lambda$.

Equivalently we can define Λ with an abstract syntax: $\Lambda = V \mid (\Lambda\Lambda) \mid (\lambda V.\Lambda)$.

Examples $((\lambda x.(xz))y), (y(\lambda x.(xz)))$, etc.

Notation 1.3.4

1. *Letters x, y, z, etc. are used for variables in V.*
2. *Letters L, M, N, P, Q, R are used for elements of Λ.*
3. Syntactic *identity of terms is denoted by $\equiv$, i.e. $(xy) \equiv (xy) \not\equiv (xz)$.*
4. *We also drop parentheses as we wish, using left-associativity for application, and application having higher precedence than abstraction. We may also write multiple variables under λ to mean multiple right-associative abstractions. For example, $\lambda xy.xyx \equiv \lambda x.(\lambda y.((xy)x))$.*

With *terms* (elements of Λ) defined, we now define *subterms* of a term by means of a recursive definition Sub.

Definition 1.3.5 (*Multiset* of subterms)

1. $\text{Sub}(x) = \{x\}$, for all $x \in V$.
2. $\text{Sub}((MN)) = \text{Sub}(M) \cup \text{Sub}(N) \cup \{(MN)\}$.
3. $\text{Sub}((\lambda x.M)) = \text{Sub}(M) \cup \{(\lambda x.M)\}$.

Note that we compute these as *multisets*, i.e. counting duplicates. From this definition it is easy to see that subterms are reflexive and transitive. A *proper* subterm is a subterm excluding reflexivity, i.e. L is a proper subterm of M if L is a subterm of M and $L \not\equiv M$.

2 Free and bound variables

Each occurrence of a variable is either *free*, *bound*, or *binding*. The *binding* occurrences are precisely the occurrences immediately after a λ. Besides that, variables are initially *free*, until they are *bound* by a matching λ binding variable. More precisely we define FV recursively to be the set of free variables in a λ-term:

Definition 1.4.1 (FV, set of free variables in a term)

1. $FV(x) = \{x\}$.
2. $FV(MN) = FV(M) \cup FV(N)$.
3. $FV(\lambda x.M) = FV(M) \backslash \{x\}$.

Examples

$$FV(\lambda x.xy) = \{y\}$$

$FV(x(\lambda x.xy)) = \{x, y\}$ (Note that the first occurence of x is free, the second occurrence of x is bound).

Definition 1.4.3 (Closed term; combinator; Λ^0) By closed λ-term, combinator, or Λ^0 is meant terms with no free variables.

3 Alpha conversion

Since λ-terms are meant to represent the abstract behavior of functions, the specific names of variables used should not matter. For example, we would like to consider $\lambda x.xx$ to be the same as $\lambda y.yy$, even though syntactically they are distinct. To this end we define an equivalence relation $=_\alpha$ on Λ as follows.

Definition 1.5.2 (α-conversion or α-equivalence, $=_\alpha$) Let $M^{x \to y}$ denote the result of replacing every *free* occurrence of x in M with y. Then we define $=_\alpha$ as follows:

1. (Renaming) $\lambda x.M =_\alpha \lambda y.M^{x \to y}$, if $y \notin FV(M)$ and y is not a binding variable in M.
2. (Compatibility) If $M =_\alpha N$, then $ML =_\alpha NL$, $LM =_\alpha LN$, and, for arbitrary z, $\lambda z.M =_\alpha \lambda z.N$.
3. (Reflexivity) $M =_\alpha M$.
4. (Symmetry) If $M =_\alpha N$, then $N =_\alpha M$.
5. (Transitivity) If $L =_\alpha M$ and $M =_\alpha N$, then $L =_\alpha N$.

Note that α-equivalence does *NOT* allow you to rename arbitrary variables, it only allows for renaming at binding occurrences (and the appropriately bound occurrences). That is, $xy \neq_\alpha zy$, but $\lambda x.xy =_\alpha \lambda z.zy$.

Examples

$$(\lambda x.x(\lambda z.xy))z =_\alpha (\lambda x.x(\lambda z.xy))z$$
$$(\lambda x.x(\lambda z.xy))z =_\alpha (\lambda u.u(\lambda z.uy))z$$
$$(\lambda x.x(\lambda z.xy))z =_\alpha (\lambda z.z(\lambda x.zy))z$$
$$(\lambda x.x(\lambda z.xy))z \neq_\alpha (\lambda y.y(\lambda z.yy))z$$
$$(\lambda x.x(\lambda z.xy))z \neq_\alpha (\lambda z.z(\lambda z.zy))z$$
$$(\lambda x.x(\lambda z.xy))z \neq_\alpha (\lambda u.u(\lambda z.uy))v$$
$$\lambda xy.xzy =_\alpha \lambda vy.vzy$$
$$\lambda xy.xzy =_\alpha \lambda vu.vzu$$
$$\lambda xy.xzy \neq_\alpha \lambda yy.yzy$$
$$\lambda xy.xzy \neq_\alpha \lambda zy.zzy$$

4 Substitution

In order to build up to defining β-reduction, we now give a formal definition of *substitution*. By $M[x := N]$ is meant replacing the free variable x with N in M. Formally:

Definition 1.6.1 (Substitution)

1. $x[x := N] \equiv N$.
2. $y[x := N] \equiv y$ if $x \not\equiv y$.
3. $(PQ)[x := N] \equiv (P[x := N])(Q[x := N])$.
4. $(\lambda y.P)[x := N] \equiv \lambda z.(P^{y \to z}[x := N])$, if $\lambda z.P^{y \to z}$ is an α-variant of $\lambda y.P$ such that $z \notin FV(N)$.

Note that something of the form $P[x := N]$ is not literally a lambda-term, but is a meta-notation for some actual lambda-term.

Most of Definition 1.6.1 is straightforward, but case 4 is somewhat subtle. The issue with substituting under an abstraction is that a free variable in N may accidentally become bound. For example, consider $(\lambda y.yx)[x := xy]$. The y in xy is free so we expect it to remain free after substitution. But if we blindly substitute we get $\lambda y.y(xy)$ and the y incorrectly becomes bound. Rather, we follow case 4 and first rename the abstraction to get $\lambda z.((zx)[x := xy])$, and then finally $\lambda z.z(xy)$. However, in many cases the abstraction binding variable is not a free variable in N, in which case we do not need to rename the abstraction.

5 Lambda-terms module α-equivalence

Previously we defined the equivalence relation $=_\alpha$ on Λ using the intuition that α-equivalent terms represent essentially the same function (or at least the same behavior), just written down with different variable names. The following lemma shows that indeed α-equivalence respects lambda-term construction as well as substitution.

Lemma 1.7.1 *Let $M_1 =_\alpha N_1$ and $M_2 =_\alpha N_2$. Then:*

1. $M_1 N_1 =_\alpha M_2 N_2$.
2. $\lambda x.M_1 =_\alpha \lambda x.M_2$.
3. $M_1[x := N_1] =_\alpha M_2[x := N_2]$.

Whichever term-construction operations or substitution operations we perform will always respect α-equivalence classes. Thus we can mod out by α-equivalence:

Notation 1.7.3 *With an abuse of notation we say that $M \equiv N$ if they are in fact α-equivalent.*

Remark *An alternate approach to α-equivalence may be using de Bruijn indices.*

Since we can choose binding variables as we wish, we may as well choose a convention that is convenient.

Convention 1.7.4 (Barendregt convention) *We chose the binding variables to all be different, and such that each of them is different from any free variables in the term. So instead of writing $(\lambda xy.xz)(\lambda xz.z)$ we write $(\lambda xy.xz)(\lambda uv.v)$ for example. We also extend this convention to not-yet-performed substitutions, so we do not write $(\lambda x.xyz)[y := \lambda x.x]$, nor do we write $(\lambda x.xyz)[y := \lambda y.y]$, and nor do we write $(\lambda x.xyz)[y := \lambda z.z]$, but rather we write $(\lambda x.xyz)[y := \lambda u.u]$, for example.*

6 Beta reduction

Definition 1.8.1 (One-step β-reduction, $\rightarrow_\beta$)

1. $(\lambda x.M)N \rightarrow_\beta M[x := N]$
2. If $M \rightarrow_\beta N$, then $ML \rightarrow_\beta NL$, $LM \rightarrow_\beta LN$ and $\lambda x.M \rightarrow_\beta \lambda x.N$

Note that case 2 says that any subterm of the form in case 1 can be chosen to be reduced. Such a subterm that is chosen to be reduced is called a *redex*, and the outcome is the *contractum*. The intuition of $\rightarrow_\beta$ is that $(\lambda x.M)N$ represents a function being applied to an argument N, and $\rightarrow_\beta$ is the means to compute the result, via substitution of the argument into the body of the function (abstraction).

Examples

1. $(\lambda x.x(xy))N \rightarrow_\beta N(Ny)$
2. In $(\lambda x.(\lambda y.yx)z)v$ there are 2 possible one-step β-reductions, which yield 2 non-α equivalent results:

$$(\lambda x.(\lambda y.yx)z)v \rightarrow_\beta (\lambda y.yv)z$$

$$(\lambda x.(\lambda y.yx)z)v \rightarrow_\beta (\lambda x.zx)v$$

   However, if we apply another one-step β-reduction to each result we obtain zv in both cases.

3. $(\lambda x.xx)(\lambda x.xx) \rightarrow_\beta (\lambda x.xx)(\lambda x.xx)$

Now we define general β-reduction which is simply repeated one-step β reduction.

Definition 1.8.3 (β-reduction, $\twoheadrightarrow_\beta$) We say that $M \twoheadrightarrow_\beta N$ if there is a (possibly 0 length) chain $M \equiv M_0 \rightarrow_\beta M_1 \rightarrow_\beta \cdots \rightarrow_\beta M_{n-1} \rightarrow_\beta M_n \equiv N$. Clearly $\twoheadrightarrow_\beta$ is reflexive and transitive.

We can define an equivalence relation $=_\beta$ as follows.

Definition 1.8.5 (β-conversion, β-equality, $=_\beta$) We say that $M =_\beta N$ if there is a sequence $M \equiv M_0, M_1, \cdots, M_n \equiv N$ such that $M_i \rightarrow_\beta M_{i+1}$ or $M_{i+1} \rightarrow_\beta M_i$. In other words, each pair M_i and M_{i+1} are related by $\rightarrow_\beta$ but not necessarily left-to-right.

Examples $(\lambda y.yv)z =_\beta (\lambda x.zx)v$ by the chain $(\lambda y.yv)z \rightarrow_\beta zv \leftarrow_\beta (\lambda x.zx)v$

We have the following result for $=_\beta$:

Lemma 1.8.6

1. *$=_\beta$ extends $\twoheadrightarrow_\beta$ in both directions, i.e. if $M \twoheadrightarrow_\beta N$ or $N \twoheadrightarrow_\beta M$ then $M =_\beta N$.*
2. *$=_\beta$ is an equivalence relation.*
3. *If $M \twoheadrightarrow_\beta L_1$ and $M \twoheadrightarrow_\beta L_2$ then $L_1 =_\beta L_2$.*
4. *If $L_1 \twoheadrightarrow_\beta N$ and $L_2 \twoheadrightarrow_\beta N$ then $L_1 =_\beta L_2$.*

In this way we can see an equivalence class of $=_\beta$ as representing a single computation / program, but in possible different stages of carrying out the computation.

7 Normal forms and confluence

We now take a closer look at the computational aspects of β-reduction. First we define the notion of the *outcome* of a lambda-term.

Definition 1.9.1 (β-normal form, β-nf, β-normalizing)

1. *M is in β-normal form (in β-nf)* if M does not contain any redex.
2. *M has a β-nf (is β-normalizing)* if there is an N in β-nf such that $M =_\beta N$. Such an N is a *β-normal form of M*.

The following lemma is trivial:

Lemma 1.9.2 *If M is in β-nf, then $M \twoheadrightarrow_\beta N$ implies $M \equiv N$, since there are no possible reductions to perform.*

Examples

1. $(\lambda x.(\lambda y.yx)z)v$ has a β-nf of zv since $(\lambda x.(\lambda y.yx)z)v \twoheadrightarrow_\beta zv$ and zv is in β-nf.
2. Let $\Omega = (\lambda x.xx)(\lambda x.xx)$ has no β-nf.
3. Let $\Delta = \lambda x.xxx$. Then, $\Delta\Delta \to_\beta \Delta\Delta\Delta \to_\beta \Delta\Delta\Delta\Delta \to_\beta \cdots$. Since there are no other options for redexes to apply in this chain, $\Delta\Delta$ has no β-nf.
4. Let Ω be defined as in 2. Then $(\lambda u.v)\Omega$ has a β-nf of v, but you must be careful when choosing the redex, since if you always choose Ω as the redex you will never reach β-nf.

Example 4. above shows that the choice of what path to take in β-reduction might matter. We define a reduction path now.

Definition 1.9.5 (Reduction path) A *finite reduction path* from M is a finite sequence $M \equiv N_0 \to_\beta N_1 \to_\beta N_2 \to_\beta \cdots \to_\beta N_n$, or equivalently if $M \twoheadrightarrow_\beta N_n$. An *infinite reduction path* from M is an infinite sequence sequence $M \equiv N_0 \to_\beta N_1 \to_\beta N_2 \to_\beta \cdots$.

We can now define some subsets of Λ that "behave nicely" with β-reduction:

Definition 1.9.6 (Weak normalization and strong normalization)

1. M is *weakly normalizing* if there is an N in β-nf such that $M \twoheadrightarrow_\beta N$.
2. M is *strongly normalizing* if there are no infinite reduction paths starting from M.

Strongly normalizing implies weakly normalizing.

Examples

1. $(\lambda u.v)\Omega$ is weakly normalizing, but not strongly normalizing.
2. $(\lambda x.(\lambda y.yx)z)v$ is strongly normalizing.
3. Ω and $\Delta\Delta$ are not weakly normalizing.

Theorem 1.9.8 (Church-Rosser; CR; Confluence) *Suppose that for a term M, we have that $M \twoheadrightarrow_\beta N_1$ and $M \twoheadrightarrow_\beta N_2$. Then there exists a term N_3 such that $N_1 \twoheadrightarrow_\beta N_3$ and $N_2 \twoheadrightarrow_\beta N_3$.*

Please see other sources for the proof (it is not included in the book either). The intuitive consequence of the Church-Rosser Theorem is that the outcome of a computation (if it exists) is independent of the order in which intermediate computations are performed. This is what we expect in computations. For example, in $(1 + 2) + (3 + 4)$ we can compute either $3 + (3 + 4)$ or $(1 + 2) + 7$ first, but it does not matter as we will get the same answer at the end.

Corollary 1.9.9 *Suppose that $M =_\beta N$. Then there is an L such that $M \twoheadrightarrow_\beta L$ and $N \twoheadrightarrow_\beta L$.*

Proof. See book for proof. □

From Corollary 1.9.9 we obtain the following result:

Lemma 1.9.10

1. *If M has N as a β-nf, then $M \twoheadrightarrow_\beta N$.*
2. *A term M has at most one β-nf.*

Proof.

1. We have that $M =_\beta N$ with N in β-nf. By Corollary 1.9.9 there is an L such that $M \twoheadrightarrow_\beta L$ and $N \twoheadrightarrow_\beta L$. Since N is in β-nf by Lemma 1.9.2 $N \equiv L$. This means that $M \twoheadrightarrow_\beta L \equiv N$, so $M \twoheadrightarrow_\beta N$.
2. Assume that M has 2 β-nfs, N_1 and N_2. By part 1. we have that $M \twoheadrightarrow_\beta N_1$ and $M \twoheadrightarrow_\beta N_2$. By the Church-Rosser Theorem there is an L such that $N_1 \twoheadrightarrow_\beta L$ and $N_2 \twoheadrightarrow_\beta L$. But since both N_1 and N_2 are in β-nf we have that $N_1 \equiv L$ and $N_2 \equiv L$, so $N_1 \equiv N_2$.

□

Informally Lemma 1.9.10 as the following consequences:

1. If a term has an outcome (i.e. a β-nf), then we can reach this outcome by only performing *forward* computation (i.e. β-reduction).
2. An outcome of a computation, if it exists, is unique.

8 Fixed Point Theorem

Theorem 1.10.1 (Fixed Point Theorem) *Suppose $L \in \Lambda$. Define $M := (\lambda x.L(xx))(\lambda x.L(xx))$. Then $LM =_\beta M$.*

Proof. $M \equiv (\lambda x.L(xx))(\lambda x.L(xx)) \to_\beta L((\lambda x.L(xx))(\lambda x.L(xx))) \equiv LM$. Thus, $LM =_\beta M$. □

Note that these fixed points do not necessarily represent numbers or other concrete values. For example, we can define a successor function on natural numbers in λ-calculus, and such a successor function has a fixed point, but this fixed point does not represent any natural number.

From the above proof it follows that there is a so called *fixed point combinator*, i.e. a closed term which "constructs" a fixed point for an arbitrary given input term. It is given by:

$$Y \equiv \lambda y.(\lambda x.y(xx))(\lambda x.y(xx))$$

This gives that for any term L, YL is a fixed point of L.

A very nice consequence of the existence of fixed points is the ability to solve recursive equations. Suppose we have the following recursive equation:

$$M =_\beta \boxed{\cdots\cdots M \cdots\cdots}$$

in which M might appear multiples times on the right side. Our goal is to solve for M such that the given equation holds. Let L be the right hand side expression, but with abstracted by λz and with M replaced by λz. That is, $L \equiv \lambda z.\boxed{\cdots\cdots z \cdots\cdots}$. Clearly, $LM \to_\beta \boxed{\cdots\cdots M \cdots\cdots}$. Thus, it suffices to find an M such that $M =_\beta LM$, as then $M =_\beta \boxed{\cdots\cdots M \cdots\cdots}$. Such an M explicitly exists by Theorem 1.10.1. Let's see this with some examples.

Examples

1. Does there exists a term M such that $Mx =_\beta xMx$? We can rephrase this as asking if there exists an M such that $M =_\beta \lambda x.xMx$. Define $L := \lambda z.(\lambda x.xzx)$. Then $LM \to_\beta \lambda x.xMx$, so if we find M such that $M =_\beta LM$ then we are done. We can do this via the fixed point combinator, giving YL to be the desired result.

2. We can code natural numbers in untyped lambda calculus, as well as *add, mult, pred, iszero*, and we can also code booleans and a conditional *if-then-else*. (See exercises in book for these). Suppose we already have these, and we wish to define a function *fact* which computes the factorial. In other words, we want to get some $fact \equiv \lambda n.(\cdots)$. Rather than work with the lambda terms explicitly, we can say that *fact* should obey this recursive equation:

$$\text{fact}\, x =_\beta \text{if}(\text{iszero}\, x)\, \text{then}\, 1\, \text{else}\, \text{mult}\, x\, (\text{fact}(\text{pred}\, x))$$

   Then, we can obtain the entire term fact by solving this equation using the above method.

9 Conclusions

1. Things that are pretty great about λ-calculus:
   - We have formalized the behavior of functions.
   - It is a clean and simply formalization (I guess?).
   - Substitution seems to be fundamental to function evaluation. It is a bit tricky to get right, but we have treated it rigorously in λ-calculus.
   - Conversion is a cool extension of reduction.
   - We have a formalization of "outcomes" of computations by β-nf.
   - Confluence (Church-Rosser Theorem) holds, which is something we would expect for computations.
   - β-nf is unique (if it exists)
   - Many recursive equations can be solved via fixed points.
   - λ-calculus is Turing-complete.
2. Things that aren't so great:
   - Weird things such as self-application (xx or MM) are allowed, though they are quite counter-intuitive.
   - Not all terms have a β-nf, we get undesired infinite computation.
   - Every λ-term has a fixed point, which is quite different from ordinary functions in mathematics.