On capture-avoiding substitution in the λ-calculus

James Caldwell and Aaron McClellan
Department of Computer Science
University of Wyoming
Laramie, WY

September 15, 2020

1 Introduction

Capture-avoiding substitution is the fundamental operation in the λ-calculus. It is used to define both α-equivalence (which is the standard notion of what it means for two λ-terms to be identified) and β-reduction (which captures the notion of computation in the system).

But also, capture-avoiding substitution is notoriously difficult to reason about. Many (if not most) presentations of λ-calculus sweep the details under the rug[1] by following the conventions of the classic presentation by Barendregt [1]. There, the difficulties are hidden by the adoption of two conventions:

1. 2.1.12. Convention. Terms that are α-congruent are identified. So now we write $\lambda x.x = \lambda y.y$, et cetera.

2. 2.1.13. Variable Convention. If $M_1, ..., M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

It is difficult to find detailed proofs about the properties of capture-avoiding substitution which do not adopt these conventions, and proofs are often left to the reader.

The difficulty arising from capture-avoiding substitution is twofold. First, structural induction on λ-terms requires potentially complex proofs on several cases of the term. Second, name collisions can mutate the λ-term in question, making it difficult to apply the induction hypothesis. Here, we will provide detailed proofs of a number of properties of capture-avoiding substitution; the reader will no-doubt, find the proofs technically detailed and tedious. Finally, we provide a possible solution to some of the complexities introduced by capture-avoiding substitution.

[1] Both Curry and Feys [2] and Hindley and Seldin [4] include rigorous proofs but use less intuitive notation or hide the rigor in an appendix at the end of the book.

2 Syntax

The syntax for (pure) λ-terms is given as follows:

$$\begin{array}{l}\Lambda ::= x|MN|\lambda x.M \\ \quad \text{where } \ M, N \in \Lambda \ \text{ and } \ x \in \mathbf{V}\end{array}$$

Λ is the set (syntactic class) of all λ-terms and $\mathbf{V}$ is the set of variable names.

3 Induction Principles

3.1 Structural Induction

Structural induction follows the syntactic structure of a term.

Theorem 1 (Structural Induction)**.**

$$\begin{array}{l}((\forall x.\mathcal{P}(\S))\wedge \\ \quad (\forall M, N.\mathcal{P}(\mathcal{M}) \wedge \mathcal{P}(\mathcal{N}) \Rightarrow \mathcal{P}(\mathcal{M}\mathcal{N}))\wedge \\ \quad (\forall x, M.\mathcal{P}(\mathcal{M}) \Rightarrow \mathcal{P}(\lambda\S.\mathcal{M})) \\ (\Rightarrow (\forall M.\mathcal{P}(\mathcal{M}))))\end{array}$$

This theorem justifies the following proof technique for proving a property $\mathcal{P}$ holds for all λ-terms.

Proof Technique 1 (Term Induction)**.** *To show a property* $\mathcal{P} : * \to$ **Bool** *holds for every* λ*-term, show the following:*

$$\begin{array}{ll}i.) & \forall x.\mathcal{P}(\S) \\ ii.) & \forall M, N.\mathcal{P}(\mathcal{M}) \wedge \mathcal{P}(\mathcal{N}) \Rightarrow \mathcal{P}(\mathcal{M}\mathcal{N}) \\ iii.) & \forall x, M.\mathcal{P}(\mathcal{M}) \Rightarrow \mathcal{P}(\lambda\S.\mathcal{M})\end{array}$$

This technique sometimes is difficult to use because, in case (iii.), the variable x is arbitrary; it may or may not be a free variable of M.

3.2 Measure Induction

In general, a useful induction principle for arbitrary types A (A could be collection of syntactic representations of, say λ-terms) is to use so-called *complete induction* on the natural numbers ($\mathbb{N}$) by first applying a measure function of type $A \to \mathbb{N}$ mapping elements of A to a natural number.

Theorem 2 (Measure Induction)**.** *For every* $f : A \to \mathbb{N}$*, the following holds for all properties* $\mathcal{P} : A \to \mathbb{B}$

$$\begin{array}{l}\forall M : A, \ (\forall N : A.\ f(N) \leq f(M) \Rightarrow \mathcal{P}(N)) \Rightarrow \mathcal{P}(M) \\ \quad \forall M : \mathcal{A}.\mathcal{P}(M)\end{array}$$

We can define a rank (or measure) function which maps syntax to a natural number. We can then use the analogue of *complete induction on the natural numbers*, sometimes also known as *rank induction* or *measure induction*.

A useful rank function, mapping λ-terms to natural numbers is the following:

$$\begin{aligned} r(x) &= 0 \\ r(MN) &= r(M) + r(N) + 1 \\ r(\lambda x.M) &= r(M) + 1 \end{aligned}$$

Loosely defined, this rank function measures the complexity of a λ-term.

4 Capture-Avoiding Substitution

To define capture-avoiding substitution, we must first define two other helping definitions.

4.1 Free and Fresh Variables

We say a variable is free if it is not bound by any λ-term. A function $fv : \Lambda \to \mathbf{V}$ that calculates the free variables of a λ-term or expression can be defined as follows:

$$\begin{array}{lcl} fv(x) & = & \{x\} \\ fv(MN) & = & fv(M) \cup fv(N) \\ fv(\lambda x.M) & = & fv(M) - \{x\} \end{array}$$

A variable is "fresh" for a λ-term if it does not occur in the free variables of the term. Succinctly, x is fresh with respect to M when

$$\begin{array}{l} x \notin fv(M) \\ \qquad \text{where } \; M \in \Lambda \; \text{ and } \; x \in \mathbf{V} \end{array}$$

4.2 Capture-Avoiding Substitution

Capture-avoiding substitution is the process of replacing subterms of a λ-term with other λ-terms in such a way that the semantics of the function are unaffected. This process is where many of the difficulties often associated with λ-calculus are derived. There exist many small challenges obscured in the cases of capture-avoiding substitution. Most notoriously, case (3c) mutates the λ-term, making direct proofs dependent on both the original and post-substitution terms. It is in this case where richer proof techniques begin to show their value; in many cases, the discrepancy between terms is abstracted away leaving only equivalent values.

The notation $M[x := N]$ denotes the replacement of x for N in M where $x \in \mathbf{V}$ and $M, N \in \Lambda$.

Definition 1 (Capture-Avoiding Substitution)**.**

$$x[x := N] = N \tag{1a}$$
$$y[x := N] = y \tag{1b}$$
$$(MM')[x := N] = M[x := N]M'[x := N] \tag{2}$$
$$(\lambda x.M)[x := N] = \lambda x.M \tag{3a}$$
$$(\lambda y.M)[x := N] = \lambda y.M[x := N] \qquad x \neq y \wedge y \notin fv(N) \tag{3b}$$
$$(\lambda y.M)[x := N] = \lambda z.M[y := z][x := N] \qquad x \neq y \wedge y \in fv(N) \wedge z \text{ fresh} \tag{3c}$$

5 Equivalence of Terms

We say two λ-terms are α-*equivalent* (read "alpha equivalent") if they have the same shape modulo the names of bound variables. For example:

$$\lambda x.x =_\alpha \lambda y.y$$
$$\lambda x.\lambda y.xy =_\alpha \lambda y.\lambda x.yx$$
$$\lambda x.\lambda y.x(\lambda x.x) =_\alpha \lambda y.\lambda x.y(\lambda z.z)$$

The following definition specifies when a pair of terms are α-equivalent:

$$x =_\alpha y \quad \stackrel{\text{def}}{=} \quad \begin{cases} true & \text{if } x \text{ and } y \text{ are identically the same variable.} \\ false & \text{otherwise} \end{cases}$$
$$MN =_\alpha M'N' \quad \stackrel{\text{def}}{=} \quad M =_\alpha M' \wedge N =_\alpha N'$$
$$(\lambda x.M) =_\alpha (\lambda y.N) \quad \stackrel{\text{def}}{=} \quad M[x := z] =_\alpha N[y := z] \text{ where } z \text{ is fresh with respect to } M \text{ and } N.$$
$$M =_\alpha N \quad \stackrel{\text{def}}{=} \quad false \text{ otherwise}$$

This equivalence captures an intuition mathematicians and computer scientists frequently exercise: the name of a variable has no bearing on the outcome. It also forms the foundation for other concepts in λ-calculus, among which is α-conversion. If a λ-term M undergoes capture-avoiding substitution and subsequent reduction to M' but $M =_\alpha M'$, it is said to have gone α-conversion.

Although there exist other classes of equivalence in the λ-calculus (namely β and η-equivalence), this examination requires only α-equivalence and thus will refrain from providing other definitions.

6 A Simple Theorem

All preliminary information has been presented; an elementary theorem can now be proposed and proved. The purpose of this exercise is to demonstrate the most infamous pitfall of capture-avoiding substitution as well as show the general tediousness of proofs in λ-calculus. Now, for the theorem in question:

Theorem 3 (No-Substitution Theorem)**.** *For given λ-terms M, N and a variable x*

$$x \notin fv(M) \implies M[x := N] = M$$

This appears to be a rather innocent-looking theorem. It is simple to see that no meaningful substitution would be performed, thus leaving M unchanged. Surely a direct term induction proof would suffice? Alas, simple term induction will fall victim to the final case of capture-avoiding substitution.

Proof. To begin, assume $x \notin fv(M)$. Then we induct over the terms of M; we must prove theorem 3 for each case of capture-avoiding substitution.

Case 1a: By definition, $fv(M) = x$ and the statement is vacuously true.

Case 1b: $y[x := N]$ is trivially equal to y.

Case 2: Following definitions, we know

$$M[x := N] = (AB)[x := N] = A[x := N]B[x := N].$$

Then, by the induction hypothesis of case 2 of proof technique 1,

$$A[x := N]B[x := N] = AB = M.$$

Case 3a: $(\lambda x.A)[x := N]$ is trivially equal to M.

Case 3b: By definition, we know $(\lambda y.A)[x := N] = \lambda y.A[x := N]$. Then, by the induction hypothesis of case 3 of proof technique 1, $\lambda y.A[x := N] = \lambda y.A = M$.

Case 3c: By definition, $M[x := N] = \lambda z.A[y := z][x := N]$. Let $A[y := z]$ reduce to some λ-term A'. Then $M[x := N] = \lambda z.A'[x := N]$.

This proof is intentionally left incomplete.

□

At this points, one must note several things. First, the proof is simple yet tedious due to the necessity of proofs corresponding to each case of capture-avoiding substitution. Second and most importantly, the proof is incomplete. The transformation of $\lambda z.A$ into $\lambda z.A'$ has rendered the induction hypothesis inapplicable. Referring back to case 3 of 1, the hypothesis requires the abstracted λ-terms in the antecedent and consequent to be equal. Alas, A is not strictly equal to A', hence the hypothesis cannot be used and the proof must be left incomplete without an alternative proof technique. This is the phenomenon of mutating λ-terms previous mentioned.

At this point, a question can be posed: "If such a simple theorem cannot be proven with term induction, how is any theorem of λ-calculus proven?" The answer usually lies with measure induction. Measure induction is a more widespread and, in this context, useful technique. To fully appreciate the richer induction hypothesis, one should prove theorem 3. The proof presented here is similar to the one presented in [2]. For brevity, cases identical to the previous proof are omitted leaving only cases 3b and 3c.

First, we prove a useful lemma (variable substitution doesn't change rank).

Lemma 1. *Given the aforementioned rank function,* $r(M) = r(M[x := u])$.

Proof. We prove by case analysis

Case 1a/1b: All terms in question are variables and thus have rank 0.

Case 2: By definition, $M[x := u] = (AB)[x := u] = A[x := u]B[x := u] = A'B'$ for some A', B'. Since $A =_\alpha A'$ and $B =_\alpha B'$, $r(A) = r(A')$ and $r(B) = r(B')$. Therefore $r(AB) = r(A'B') = r(M)$.

Case 3a: The rank is equal because the terms are identical.

Case 3b: We induct over the rank of M i.e. measure induct over M. Let the base case be when the rank is zero. Since this occurs when the term is a variable, the base case is already proven. Now, by definition, $M[x := u] = \lambda y.A[x := u]$. By application of the induction hypothesis, $r(\lambda y.A[x := u]) = r(\lambda y.A)$ and thus $r(M[x := u]) = r(M)$.

Case 3c: We induct over the rank of M i.e. measure induct over M. Let the base case be when the rank is zero. Since this occurs when the term is a variable, the base case is already proven. Now, by definition, $M[x := u] = \lambda z.A[y := z][x := u]$. By two applications of the induction hypothesis, $r(M[x := u]) = r(M)$.

□

With this lemma, we can move on to the proof of 3.

Proof. To begin, assume $x \notin fv(M)$. Then we induct over the rank of M using the aforementioned rank function.

Case 3b: We know $M[x := N] = \lambda y.A[x := N]$. Since $r(A[x := N]) < \lambda y.r(A[x := N])$, we can apply the induction hypothesis. Hence $A[x := N] = A$, therefore $M[x := N] = M$.

Case 3c: We know $M[x := N] = \lambda z.A[y := z][x := N]$. We apply the induction hypothesis twice, therefore $M[x := N] = M$.

□

At long last, theorem 3 is proven. Note how rank induction's layer of abstraction over the structure of the term frees it from the chains of α-conversion, the obstruction that halted term induction. Additionally, it brings to light some repetitiveness in the definition of capture-avoiding substitution. The final two proof cases in both lemma 1 and the rank induction proof of 3 share nearly identical logic. It begs the question: could capture-avoiding substitution cases 3b and 3c be collapsed into a single case? Indeed they can as shown in [2][2]. This decision is merely an aesthetic choice and differs by author; for example, Hindley and Seldin present capture-avoiding substitution in seven cases [4] compared to six and five in this article and [2] respectively.

[2]The book does acknowledge the additional case but does so as a definition of the replacement variable rather than a case.

7 Syntactic Alternatives

So far, tackling the problem of α-conversion in theorems involving capture-avoiding substitution has required richer proof techniques. While acceptable, solving this issue at the proof level is somewhat unsatisfactory. Since the problem persists, each proof will follow the same basic formula: state the intent to induct on the rank of a term, define the base case, et cetera. Repeated boilerplate verbiage in proofs quickly becomes tedious; more fundamental solutions to the α-conversion problem are necessary to aid in this ailment. There are many solutions, one of which will be presented.

7.1 de Bruijn Indices

In 1972, mathematician Nicolaas Govert de Bruijn presented a paper in which he introduced a simple method for solving α-conversion [3]. Since the problem arises from name conflicts, the new tool, called de Bruijn indices, would simply avoid arbitrary naming of bound variables. Instead, a variable is systemically named using the number of abstractions it is bound under; if a variable is not bound, it is assigned a number greater than the number of abstractions above its call site.

7.1.1 Formal Definition

A λ-term written using de Bruijn indices take the form

$$\begin{aligned}\Lambda ::= & \ n | MN | \lambda M \\ & \text{where} \ \ M, N \in \Lambda \ \ \text{and} \ \ n \in \mathbb{N}\end{aligned}$$

Λ is the set (syntactic class) of all λ-terms. Conversion of a traditional λ-term M to de Bruijn indices follows the following algorithm (starting from the innermost scope):

1. If a variable is bound, is is assigned index n where n is the number of abstractions it is bound under. Intuitively, count how many abstractions must be traversed before the call site is reached.

2. If a variable is free, it is assigned index $i + k + 1$ where i is the maximum number of nested abstractions in M and k is the number of free variables that have been assigned an index[3].

All string-based variable names are eliminated in the assigning process. For example, the traditional λ-term $\lambda x.\lambda y.yz((\lambda u.uux)(\lambda z.kz))$ is converted into the de Bruijn term $\lambda \ \lambda \ 1 \ 5((\lambda \ 1 \ 1 \ 3)(\lambda \ 4 \ 1))$. This process also eliminates the concept of α-equality; α-equality is now equivalent to syntactic equality, thus removing the problem of mutating λ-terms seen earlier. Unfortunately, while a solution to the α-conversion problem is valuable, most humans avoid de Bruijn indices because most find it easier to read and understand the λ-terms written in the traditional syntax[4].

[3]This is not the only way to assign indices to free variables. The only requirement is that the assigning function must be able to generate globally unique indices for all free variables.

[4]However, de Bruijn indices are often used in automated systems like compiler where identification information is less valuable.

8 Conclusion

λ-calculus is a powerful and compact tool to model computation that forms the foundation for modelling computation. It is upon this base that capture-avoiding substitution is built. Capture-Avoiding Substitution is the basis for many things in λ-calculus, which is why it is unfortunate that it introduces so many complexities into an otherwise simple system. Due to these complexities, proofs involving capture-avoiding substitution oftentimes end up being fraught with subtle errors.

Many authors avoid rigor in an effort to circumvent the logical tar-pits presented. However, such workarounds leave proofs feeling hollow and readers feeling unsatisfied. To fill the void, we presented definitions, proof techniques, a proof, and detailed annotations explaining the difficulties as they arise, all while using none of the standard conventions that many authors accept. Lastly, we presented a possible solution to the complexities of α-conversion that arise from capture-avoiding substitution.

References

[1] Henk P. Barendregt. The lambda calculus: its syntax and semantics. In *Studies in Logic*, volume 103. Amsterdam:North-Holland, 1981.

[2] H.B. Curry, R. Feys, and W. Craig. *Combinatory Logic, Vol. 1.* Amsterdam:North-Holland, 1968.

[3] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. North-Holland, 1972.

[4] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press, New York, NY, USA, 2 edition, 2008.