Don't Substitute Into Abstractions (Functional Perl)

Ben Lippmeier
Vertigo Technology and
University of New South Wales
benl@ouroborus.net

Abstract

We present a new view on an old solution to the problem of name capture in the lambda calculus. During reduction, by not substituting into abstractions we can retain the names present in the initial expression, and avoid the need to generate new, fresh names.

1. Introduction

Let's reduce the following expression:

$$(\lambda x.\ \lambda y.\ add\ \ x\ \ y)\ \ (succ\ \ y)\ \ five$$

The free variables are *add*, *succ*, *five*, and the right-most occurrence of *y*. Although this is a simple expression, when we reduce it we need to manage the fact that *y* is used as both the name of a binder (on the left) as well as a free variable (on the right). We cannot naively substitute our function's first argument into its body, as the binding occurrence of *y* will *capture* the previously free occurrence of *y*, like so:

$$\xrightarrow{\beta} (\lambda \underline{y}.\ add\ \ (succ\ \ \underline{y})\ \ \underline{y})\ \ five \qquad \text{(wrong)}$$

The standard solution is to rename binders so they do not clash with the names of free variables, then perform the substitution as before. The renaming process is called *alpha conversion*. Doing this for the initial expression yields:

$$\begin{aligned}
&\xrightarrow{\alpha} (\lambda x.\ \lambda purple.\ add\ \ x\ \ purple)\ \ (succ\ y)\ \ five\\
&\xrightarrow{\beta} (\lambda purple.\ add\ \ (succ\ y)\ \ purple)\ \ five\\
&\xrightarrow{\beta} add\ \ (succ\ \ y)\ \ five
\end{aligned}$$

Alpha conversion allows us to arrive at the correct answer, but we are left with the awkward question of where the new names (like *purple*) come from. In standard presentations, the new names are required to be *fresh*, meaning unused so far in the given reduction. However, the process of generating fresh names is usually delegated to the meta-level, rather than being part of the system defined. This approach is acceptable for pen-and-paper mathematics, but those of us building compilers and mechanized proofs of language properties are left in a bind.[1]

Other approaches to variable binding include using de Bruijn indices or levels [3], the locally named [4] and nameless [8] approaches, as well as Higher Order Abstract Syntax (HOAS) [10], nominal techniques [11], using a pointer based graphical representation of the program [14], and simply axiomatizing alpha equivalence [7].

However, no one approach seems to be suitable for all applications: the de Bruijn representations trade intuitive named binders for unintuitive numbers; locally named and nameless approaches still need to generate fresh names; HOAS again punts the problem to the meta-level; nominal techniques require substantial tool support; pointer based approaches are well suited to concrete implementations but not the abstract definition of language semantics, and axiomatic techniques work for proof but not for implementation. In practice, implementations of interpreters and compilers often use de Bruijn indices, or techniques (or hacks) to generate fresh names based on global counters [2].

The contribution of this paper is a new view on an old approach to name capture, which is at once simple, easy to explain, and limited (meaning targeted) in application. The tragedy of name capture arises when we substitute an expression into an abstraction, and the expression being substituted has a free variable with the same name as the binder. Our solution is to just not do that.

In summary we make the following contributions:

- We present a solution to the name capture problem that allows us to reduce lambda expressions without the need to generate fresh names, or rename existing ones.
- The solution allows us to reduce open terms without needing to separate variables into multiple classes, as in locally named and nameless approaches.
- Our calculus and its semantics are fully mechanized in Coq, and we provide a simple interpreter.

Our calculus is named λ_{dsim} (lambda don't substitute into me). It is related to prior work on explicit substitutions, in particular the $\lambda\sigma_w$ theory of Curien, Hardin and Levy [5]. The main differences are that we present a standard call-by-value reduction semantics suitable for an interpreter implementation, and do not need to allow the explicit substitution to appear everywhere in the term being reduced.

[1] or perhaps without one

2. Don't Substitute into Abstractions

Starting with the initial expression:

$$(\lambda x.\lambda y.\ add\ \ x\ \ ((\lambda z.\ z)\ \ y))\ \ (succ\ \ y)\ \ five \tag{1.1}$$

We want to apply the left-most function to its first argument, substituting $(succ\ y)$ for x in its body. We use the syntax $[x = succ\ y]$ (with square parenthesis) for the meta-level operation of performing this substitution.

$$\xrightarrow{\beta}\ ([x = succ\ y]\ \ (\lambda y.\ add\ \ x\ \ ((\lambda z.\ z)\ y)))\ \ five \tag{1.2}$$

At this point we have a problem, because carrying *succ y* under the λy binder would result in the capture of y. Instead, we reify the meta-level substitution into the syntax, writing $\{x = succ\ y\}$ for a concrete substitution (with curly parenthesis), which is attached to the outside of the abstraction. Here, $\triangleright$ is pronounced "attached to".

$$\xrightarrow{\beta}\ (\{x = succ\ y\} \triangleright (\lambda y.\ add\ \ x\ \ ((\lambda z.\ z)\ y)))\ \ five \tag{1.3}$$

When we apply the function to its *next* argument, we add the associated binding $y = five$ to the one we already have, and carry the result into the body, using meta-level substitution. Applying an abstraction eliminates its binder, so it is no longer a problem.

$$\xrightarrow{\beta}\ [x = succ\ y,\ y = five]\ \ (add\ \ x\ \ ((\lambda z.\ z)\ y)) \tag{1.3}$$

Again, when we reach an abstraction we reify the meta-level substitution into the syntax, attach the resulting bindings to the outside of the abstraction, then wait until we can apply that abstraction to its argument:

$$\xrightarrow{sub}\ add\ (succ\ y)\ ((\{x = succ\ y,\ y = five\} \triangleright (\lambda z.\ z))\ \ five) \tag{1.4}$$

Applying the final abstraction to its argument completes the job:

$$\xrightarrow{\beta}\ add\ (succ\ y)\ ([x = succ\ y,\ y = five,\ z = five]\ \ z) \tag{1.5}$$

$$\xrightarrow{sub}\ add\ (succ\ y)\ \ five \tag{1.6}$$

Note that in step (1.3), when we add the new binding $y = five$ to the substitution, the binding goes on the *right*. A substitution is an ordered list of bindings, where the ones on the right have priority. Also note that the bindings $x = succ\ y$ and $y = five$ are in the same order as the binders λx and λy in the initial expression (1.1).

2.1 Name Shadowing

Suppose we want to reduce a different expression where an inner binder shadows an outer one:

$$(\lambda y.\ \lambda x.\ \lambda x.\ add\ \ one\ \ x\ \ y)\ \ x\ \ two\ \ three$$

Now we have two binders named x, as well as a free occurrence of x on the right. Performing the reduction yields:

$$(\lambda y.\ \lambda x.\ \lambda x.\ add\ \ one\ \ x\ \ y)\ \ x\ \ two\ \ three \tag{2.1}$$

$$\xrightarrow{\beta}\ (\{y = x\} \triangleright (\lambda x.\ \lambda x.\ add\ \ one\ \ x\ \ y))\ \ two\ \ three \tag{2.2}$$

$$\xrightarrow{\beta}\ (\{y = x,\ x = two\} \triangleright (\lambda x.\ add\ \ one\ \ x\ \ y))\ \ three \tag{2.3}$$

$$\xrightarrow{\beta}\ ([y = x,\ x = two,\ x = three]\ \ (add\ \ one\ \ x\ \ y)) \tag{2.4}$$

$$\xrightarrow{sub}\ add\ \ one\ \ three\ \ x \tag{2.5}$$

In the result the variable x is free, as it was in the initial expression. Our substitution $[y = x,\ x = two,\ x = three]$ is in fact a *right biased simultaneous priority substitution*. It is a right-biased priority substitution because when we apply it to a particular variable, say x, we replace that variable with the expression in the right-most matching binding. It is a simultaneous substitution because we once we replace a variable with an expression, we do not additionally apply the substitution to that expression. Each binding in the substitution operates independently.

2.2 Nested Substitutions

The final piece of our system is the mechanism for applying a meta-level substitution to an expression that already has an attached concrete substitution. Consider the following:

$$(\{x = succ\ \ y\} \triangleright (\lambda y.\ add\ \ x\ \ y\ \ z))\ \ five$$

Suppose we want to apply a further substitution to this expression, say to replace y by *one* and z by *three*. We represent this as follows:

$$[y = one,\ z = three]\ ((\{x = succ\ \ y\} \triangleright (\lambda y.\ add\ \ x\ \ y\ \ z))\ \ five)$$

To combine the outer substitution with the inner one, we first apply the outer substitution to each binding of the inner one, then append the outer substitution to the *left* of that result.

$$(\{y = one,\ z = three,\ x = succ\ one\} \triangleright (\lambda y.\ add\ \ x\ \ y\ \ z))\ \ five$$

Completing the reduction yields:

$$[y = one,\ z = three,\ x = succ\ one,\ y = five]\ \ (add\ \ x\ \ y\ \ z)$$
$$\xrightarrow{sub}\ add\ \ (succ\ \ one)\ \ five\ \ three$$

In the last step, the binding $y = five$ at the front (right) of the list shadows the $y = one$ binding at the back (left). We choose to preserve shadowed bindings in the list to simplify the semantics and meta-theory. In an interpreter implementation it would be more helpful to remove shadowed bindings during evaluation, and reclaim the associated space.

Finally, note that the interaction between meta-level substitution and beta-reduction makes this evaluation method work out nicely. Reifying a meta-level substitution into the term allows us to suspend its execution and avoid ever needing to handle the case that can result in name capture. Once the meta-level operation is suspended, performing the next beta-reduction both eliminates the problematic binder and re-starts the substitution process.

3. Formal System

The grammar and meta-functions for a simply typed version of λ_{dsim} are given in Figure 1. We use a for atomic type names, and x for term variables. The abstraction form $\theta \triangleright \lambda x : \tau.e$ includes a concrete substitution θ. We write term application with an explicit @ operator for clarity. Note that the only place where an explicit substitution can appear in a term is attached to an abstraction. We use explicit substitutions to deal with the complexities of variable binding, and only the construct that binds a variable needs to be modified relative to the standard lambda calculus.

The concrete substitutions θ are right biased lists of term bindings, where $\bullet$ is an empty substitution. Similarly, type environments are right biased lists of type signatures, where we overload $\bullet$ to mean an empty environment. We write the result of appending two substitutions as $\theta_1 \circ \theta_2$, and similarly for environments.

In the formal presentation we write *subst* θ e for the application of an explicit substitution θ to a term e. In the previous section we wrote $\{...\}$ for concrete substitutions and $[...]$ e for a meta-level substitution applied to a term. We did this for expositional purposes, but we see now that both the meta-level and concrete substitutions are just lists of term bindings — in the abstract syntax there are no parenthesis. In the definition of substitution *mapExp* is a meta level operation that applies its functional argument to the expressions in a list of bindings.

We define two right-biased lookup functions, $lookup_S$ for substitutions and $lookup_E$ for type environments. We use *None* and *Some* as constructors of a meta-level option type, which we use to express whether or not a particular binding appears in a substitution or type environment.

3.1 Type Checking

Figure 2 gives the rules for type checking. The judgment $\Gamma \vdash e :: \tau$ reads "Under type environment Γ, expression e has type τ". The judgment $\Gamma \vdash \theta \dashv \Delta$ reads: "Under type environment Γ, substitution θ has type Δ", where Δ is a list of type signatures, one for each element of the substitution.

The typing rules themselves are similar to the ones for simply typed lambda calculus. In rule TySub we lift the single-expression typing judgment to an entire substitution, to yield a list of types of the term bindings. In rule TyAbs we take the types of all the bindings in the concrete substitution and add this to the environment. The types from the substitution are added to the *left* of the $x : \tau_1$ signature for the formal parameter. The bound variable x shadows any similarly named variables in the substitution.

3.2 Evaluation

Figure 2 also gives the call-by-value evaluation semantics. Rule EsReduce performs the role of β-reduction. The difference for λ_{dsim} being that when we substitute the argument into the body, we also carry down the substitution attached to the outside of the abstraction. This rule converts the concrete substitution into the meta-level version. During reduction, the meta-level substitution is converted back to an concrete one by the definition of *subst* back in Figure 1.

In Figure 3 the judgment (e **value**) indicates that e is a value, the only one of which is an abstraction. The judgment (e **done**) indicates that e has finished reduction. Reduction ends when the term is in weak head normal form.

4. Metatheory

Our Coq formalization contains the soundness theorems of Substitution, Progress and Preservation, which we state as follows:

(Substitution)
If $\Gamma \vdash \theta \dashv \Delta$ and $\Gamma \circ \Delta \vdash e :: \tau$
then $\Gamma \vdash \mathit{subst}\ \theta\ e :: \tau$

(Progress)
If $\Gamma \vdash e :: \tau$
then e **done** or $e \longrightarrow e'$ for some e'

(Preservation)
If $\Gamma \vdash e :: \tau$ and $e \longrightarrow e'$
then $\Gamma \vdash e' :: \tau$

The Coq proof itself is quite pleasing. In particular, there are no lemmas about adjusting de Bruijn indices, no need for tactics specialized to our particular approach to variable binding, and no need to invoke any extra axioms. Our interpreter implementation also implements the semantics as written. We did prove some extra lemmas about lists, though these are completely generic, and could be added to the Coq base libraries.

5. Limitations

The technique of not substituting into abstractions avoids name capture during reduction, but is not a "full spectrum" approach to binding. Specifically, the typing rules of polymorphic calculi introduce additional scoping problems that are unrelated to reduction. Consider the following System-F example:

$$(\Lambda a.\ \lambda x : a.\ \Lambda a.\ \lambda y : a.\ \text{pair}\ x\ y) :: \forall a.(a \to \forall a.\ (a \to a \times a))$$

Here we see that reusing the names of type binders in the inferred type results in name capture. Although no reduction has taken place, we still need to introduce new names for the type to be well scoped, for example:

$$(\Lambda a.\ \lambda x : a.\ \Lambda a.\ \lambda y : a.\ \text{pair}\ x\ y) :: \forall a.(a \to \forall b.\ (b \to a \times b))$$

Language Grammar

a	::=	(type names)	
x	::=	(term variables)	
τ	::=	$a \mid \tau \to \tau$	(types)
e	::=	$x \mid \theta \rhd \lambda x : \tau.\ e \mid e\ @\ e$	(terms)
θ	::=	$\bullet \mid \theta,\ x = e$	(term substitutions)
Γ, Δ	::=	$\bullet \mid \Gamma,\ x : \tau$	(type environments)

Substitution and Lookup

$$\begin{array}{lcl}
\mathit{subst}\ \theta\ x & & \\
\quad \mid \mathit{Some}\ e \leftarrow \mathit{lookup}_S\ \theta\ x & = & e \\
\quad \mid \mathit{otherwise} & = & x \\
\mathit{subst}\ \theta\ (\theta' \rhd \lambda x\ :\ \tau.\ e) & & \\
\quad = \theta \circ (\mathit{mapExp}\ (\mathit{subst}\ \theta)\ \theta') \rhd \lambda x : \tau.\ e & & \\
\mathit{subst}\ \theta\ (e_1\ @\ e_2) & & \\
\quad = (\mathit{subst}\ \theta\ e_1)\ @\ (\mathit{subst}\ \theta\ e_2) & & \\
\mathit{lookup}_S\ x\ \bullet & = & \mathit{None} \\
\mathit{lookup}_S\ x\ (\theta,\ x = e_1) & = & \mathit{Some}\ e_1 \\
\mathit{lookup}_S\ x\ (\theta,\ y = e_1) & = & \mathit{lookup}_S\ x\ \theta \\
\mathit{lookup}_E\ a\ \bullet & = & \mathit{None} \\
\mathit{lookup}_E\ a\ (\Gamma,\ a\ :\ \tau_1) & = & \mathit{Some}\ \tau_1 \\
\mathit{lookup}_E\ a\ (\Gamma,\ b\ :\ \tau_1) & = & \mathit{lookup}_E\ a\ \Gamma
\end{array}$$

Figure 1. λ_{dsim} Grammar and Metafunctions

$$\boxed{\Gamma \vdash e :: \tau}$$

$$\frac{\mathit{lookup}_E\ x\ \Gamma = \mathit{Some}\ \tau}{\Gamma \vdash x :: \tau}\ \text{(TyVar)}$$

$$\frac{\Gamma \vdash \theta \dashv \Delta \qquad \Gamma \circ \Delta,\ x : \tau_1 \vdash e :: \tau_2}{\Gamma \vdash \theta \rhd \lambda x : \tau_1.\ e :: \tau_2}\ \text{(TyAbs)}$$

$$\frac{\Gamma \vdash e_1 :: \tau_{11} \to \tau_{12} \qquad \Gamma \vdash e_2 :: \tau_{11}}{\Gamma \vdash e_1\ @\ e_2 :: \tau_{12}}\ \text{(TyApp)}$$

$$\boxed{\Gamma \vdash \theta \dashv \Delta}$$

$$\frac{\{\Gamma \vdash e_i :: \tau_i\}^i}{\Gamma \vdash \{x_i = e_i\}^i \dashv \{x_i : \tau_i\}^i}\ \text{(TySub)}$$

$$\boxed{e \longrightarrow e'}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\ @\ e_2 \longrightarrow e_1'\ @\ e_2}\ \text{(EsAppLeft)}$$

$$\frac{e_1\ \textbf{value} \qquad e_2 \longrightarrow e_2'}{e_1\ @\ e_2 \longrightarrow e_1\ @\ e_2'}\ \text{(EsAppRight)}$$

$$\frac{e_2\ \textbf{done}}{(\theta \rhd \lambda x : \tau.\ e_1)\ @\ e_2 \longrightarrow \mathit{subst}\ (\theta,\ x = e_2)\ e_1}\ \text{(EsReduce)}$$

Figure 2. Type Checking and Evaluation

$$\boxed{e\ \textbf{value}}$$

$$\theta \rhd \lambda x : \tau.\ e\ \textbf{value}\ \text{(ValueAbs)}$$

$$\boxed{e\ \textbf{done}}$$

$$x\ \textbf{done}\ \text{(DoneVar)} \qquad \frac{e\ \textbf{value}}{e\ \textbf{done}}\ \text{(DoneValue)}$$

$$\frac{e_1\ \textbf{done} \quad \neg(e_1\ \textbf{value})}{e_1\ @\ e_2\ \textbf{done}}\ \text{(DoneApp)}$$

Figure 3. Value and Done

In the standard presentations of System-F a side condition is added to the typing rule for type abstraction (Λ) to ensure that the bound name is not already used in the type environment [12]. However, in general this is property is not preserved during reduction, at least without performing intermediate alpha conversions.

6. Related Work

Any λ_{dsim} expression can be converted to one in the standard lambda calculus, provided we are willing to generate fresh names. For example, returning to the example from §2.2 we have:

$$(\{x = succ\ \ y\} \rhd (\lambda y.\ add\ \ x\ \ y\ \ z))\ \ five$$

To eliminate the concrete substitution from the front of the abstraction we can alpha-convert the abstraction and substitute the bindings into the body:

$$(\lambda purple.\ add\ \ (succ\ y)\ \ purple\ \ z)\ \ five$$

Alpha conversion is a decidedly non-local and computationally inefficient process. To alpha-convert an expression we must descend into every part of it, performing a brute-force search for all occurrences of the variable that is being renamed. Delaying the meta-level substitution by reifying it into the term being reduced avoids continuously exploring the entire term during reduction. This process also helps retain sharing of results longer than in the standard calculus. For example, with the following expression:

$$[y = one,\ z = three]\ ((\{x = succ\ \ y\} \rhd (\lambda y.\ add\ \ x\ \ x\ \ y))\ \ five)$$

Here we have two occurrences of x in the body, so it is better to substitute into the right of the $x = succ\ y$ binding first, *and then* carry that result into the inner abstraction, rather than the other way around.

6.1 Explicit Substitutions

Improving the computational efficiency of reduction is the main purpose of work on explicit substitutions, beginning with the $\lambda\sigma$ calculus [1], where σ refers to the explicit substitution. However the named version of $\lambda\sigma$ still suffers from the usual problems of name capture — the presentation in [1] uses de Bruijn indices. The fact that $\lambda\sigma$ allows an explicit substitution to appear at any point in the term, and is phrased as a rewrite system with no particular evaluation order also leads to non-confluence [9].

The non-confluence of $\lambda\sigma$ arises due to an interaction between the rewrite rule for beta-reduction, and the ones that carry the explicit substitution σ under binders. The $\lambda\sigma_w$ calculus [5] is then a restricted (weak) form of $\lambda\sigma$ that regains confluence by not carrying substitutions under binders. As a happy side effect, this restriction also ensures that the calculus does not suffer from name capture.

The λ_c calculus of closed reductions [6] is a related rewrite system that uses named variables but allows only closed terms to be carried under binders, thus avoiding name capture. The λ_s calculus of delayed substitutions [13] is another system with named variables which also includes a rule to convert a concrete substitution into a meta substitution, as per our own λ_{dsim}. However, λ_s also propagates substitutions under binders and thus relies on alpha conversion to avoid name capture.

By themselves, rewrite systems such as $\lambda\sigma_w$ are not directly implementable in interpreters as they have no implied order of evaluation. In $\lambda\sigma_w$ the grammar of substitutions also includes an explicit constructor that represents the result of appending two substitutions, and a rewrite rule that invokes associativity of append, rather than using standard data structures to represent substitutions.

The system in this paper, λ_{dsim}, closes the loop. We take $\lambda\sigma_w$, impose a specific evaluation order (call-by-value), and lift most of the substitution machinery back to the meta-level. Representing substitutions by lists means that we can reuse existing list libraries, in both the implementation and proof, and variable lookup is just list lookup. For an interpreter implementation we could also use other standard container types to represent substitutions, like finite maps of named variables to terms. With a finite map representation the space leak due to name shadowing mentioned in §2.2 disappears, as inserting a binding with an existing name into a map replaces the old one.

Exactly which calculus is *best* for a particular application depends on the application. However, if your goal is to reduce some open lambda expressions, and you want a cheap and cheerful implementation that does not fuss around with alpha conversion, then we claim the answer is λ–don't–substitute–into–me.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lvy. Explicit substitutions. Technical report, digital Systems Research Center, 1990.

[2] Lennart Augustsson, Mikael Rittri, and Dan Synek. On generating unique names. *Journal of Functional Programming*, 4(1), 1994.

[3] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34, 1972.

[4] Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3), 2012.

[5] Pierre-Louis Curien, Therese Hardin, and Jean-Jacques Levy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43, 1996.

[6] Maribel Fernández and Ian Mackie. Closed reductions in the lambda-calculus. In *Computer Science Logic*, 1999.

[7] Andrew D. Gordon and Tom Melham. Five axioms of alpha-conversion. In *Theorem Proving in Higher Order Logics*, 1996.

[8] James McKinna and Robert Pollack. Pure type systems formalized. In *Typed Lambda Calculi and Applications*, 1993.

[9] Paul-André Melliès. Typed lambda-calculi with explicit substitutions may not terminate. 1995.

[10] Frank Pfenning and Conal Elliott. Higher order abstract syntax. In *Programming Language Design and Implementation*, 1988.

[11] Andrew M. Pitts. Nominal logic: A first order theory of names and binding. In *Theoretical Aspects of Computer Software*, 2001.

[12] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque Sur La Programmation*. Springer-Verlag, 1974.

[13] José Espírito Santo. Delayed substitutions. In *Term Rewriting and Applications*, pages 169–183, 2007.

[14] Olin Shivers and Mitchell Wand. Bottom-up beta-substitution: Uplinks and lambda-DAGs, 2004. Basic Research in Computer Science, University of Aarhus.