Renamingless Capture-Avoiding Substitution, Intrinsically Scoped

Casper Bach Poulsen
Delft University of Technology, Netherlands

Abstract

We describe a simple and direct technique for capture avoiding substitution of untyped λ terms that avoids the need to rename bound variables during substitution. We demonstrate how this substitution technique yields correct normalization of open λ terms to weak head normal form. We also give an intrinsically scoped syntax for untyped λ terms. Using this syntax we show that the substitution technique is, indeed, capture avoiding.

2012 ACM Subject Classification Software and its engineering → Semantics; Theory of computation → Logic and verification

Keywords and phrases Capture-avoiding substitution, Untyped lambda calculus, Agda, Dependent types

Digital Object Identifier 10.4230/OASIcs.CVIT.2016.23

1 Introduction

A key stumbling block when learning or implementing an interpreter for the λ-calculus, is *capture avoiding substitution*. The issue is illustrated by the following term:

$$(\lambda f.\, \lambda y.\, (f\ 1) + y)\ (\lambda z.\ \underbrace{y}_{\text{free variable}}\ )\ 2 \tag{1}$$

This term is *not* normalizable to a number value because y is a *free variable*; i.e., it is not bound by an enclosing λ term. However, using a naïve, non capture avoiding substitution strategy to normalize the term would cause f to be substituted to yield a (wrong) intermediate reduct $(\lambda y.\, ((\lambda z.\, y)\ 1) + y)\ 2$ where the red y is *captured*; that is, it is no longer a free variable.

Following, e.g., Curry and Feys [7], Plotkin [13], or Barendregt [5], the common technique to avoid such name capture is to rename variables during substitution. For example, by renaming the λ bound variable y to r, we can correctly reduce term (1) to $(\lambda r.\, ((\lambda z.\, y)\ 1) + r)\ 2$. However, this renaming based substitution strategy is problematic for two reasons. The first reason is that renaming variables give rise to intermediate reducts whose names differ from the surface program. For applications where intermediate reducts are user facing (e.g., in error messages, or in systems based on rewriting) this gives rise to confusion. The second problem is that implementing substitution functions that do such renaming is fiddly. For these reasons, and since the need for renaming is only relevant for terms that contain free variables, many educational texts and research papers only define substitution for *closed* terms; i.e., terms that do not contain free variables. However, for some applications it is useful to reduce λ terms with free variables; for example, when implementing dependent type checkers [12], or specifying models using the mCRL2 specification language [10].

There exist alternative techniques that can be used to define (lazy) capture avoiding substitution, such as *closures* [11], *de Bruijn indices* [8], *explicit substitutions* [1], and *locally nameless* [6]. However, traditional naive substitution is sometimes preferred because intermediate reducts are easy to inspect and compare.

In recent years, Eelco Visser and I were teaching the λ-calculus to undergraduate students by having them implement definitional interpreters and substitution functions. To this end,

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:9
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

we used a technique for substitution in λ terms that is capture avoiding but does not require renaming of bound variables during substitution. The idea is to delimit and distinguish those terms in abstract syntax trees (ASTs) that have already been computed to normal forms, and to never substitute inside those. For example, using $\lfloor$ and $\rfloor$ for this delimiter, an intermediate reduct of the term labeled (1) above is $(\lambda y.\, (\lfloor(\lambda z.\, y)\rfloor\ 1) + y)\ 2$. Here the delimited highlighted term is closed under substitution, such that the substitution of y for 2 is not propagated past the delimiter; i.e., using $\rightsquigarrow$ to denote reduction:

$$
\begin{array}{ll}
 & (\lambda f.\, \lambda y.\, (f\ 1) + y)\ (\lambda z.\, y)\ 2 \\
\rightsquigarrow & (\lambda y.\, (\lfloor(\lambda z.\, y)\rfloor\ 1) + y)\ 2 \\
\rightsquigarrow & (\lfloor(\lambda z.\, y)\rfloor\ 1) + 2 \\
\rightsquigarrow & ((\lambda z.\, y) 1) + 2 \\
\rightsquigarrow & y + 2
\end{array}
$$

These reductions are equivalent to using a renaming based substitution function. However, our renamingless substitution strategy does not rename variables, and (as we demonstrate in § 2.2) is about as simple to define and implement as a substitution for closed terms.

The idea of distinguishing values from plain terms is not new (for example, *reduction semantics* [9] commonly make this distinction), but I am not aware of this idea being applied to implement capture avoiding substitution functions outside of our course.[1] This paper presents the technique, and shows that the resulting substitution functions are capture avoiding. We make the following technical contributions:

- We present a technique (§ 2) for renamingless capture avoiding substitution, that is about as simple to understand and implement as substitution for closed terms.
- We present an intrinsically scoped capture avoiding normalizer (§ 3) for the untyped λ-calculus, which explains how and why our technique is capture avoiding.

The paper is a literate Agda paper whose sources can be found here: `https://github.com/casperbp/intrinsically-capture-avoiding`

2 Renamingless Capture Avoiding Substitution

We present our technique by implementing normalizers for the untyped λ-calculus in Agda. We do not assume familiarity with Agda, but assume some familiarity with typed functional programming and *generalized algebraic data types* (GADTs). We explain Agda specific syntax in footnotes.

We first implement a normalizer for *closed* terms (in § 2.1) using a standard renamingless substitution function. We then generalize this normalizer to work on *open* terms (i.e., terms with free variables; in § 2.2) using our simple renamingless substitution strategy instead.

2.1 Normalizing Closed λ Terms

Our normalizers assume that terms use a notion of name for which it is decidable whether two names are the same. We use a parameterized Agda module to declare these assumptions:[2]

[1] This judgement is made solely by the author of the present paper. Eelco and I never discussed the novelty of the technique.

[2] Set is the type of types. So *Name* : Set declares a type parameter. The $\_{\equiv}?\_$ parameter is a *dependently typed function*: it takes two names x and y as input, and returns a proof that x and y are (un)equal

```
module Normalizer (Name : Set)
                  (_≡?_ : (x y : Name) → Dec (x ≡ y)) where
```

Using these parameters we declare a data type representing untyped λ terms:

```
data Term : Set where
  lam : Name → Term → Term
  var : Name        → Term
  app : Term → Term  → Term
```

The following standard substitution function assumes that the term being substituted for (i.e., the first parameter of the function) is closed:[3]

```
[_/_]_ : Term → Name → Term → Term
[ s / y ] (lam x t) = case (x ≡? y) of λ where
  (yes _) → lam x t
  (no  _) → lam x ([ s / y ] t)
[ s / y ] (var x) = case (x ≡? y) of λ where
  (yes _) → s
  (no _) → var x
[ s / y ] (app t₁ t₂) = app ([ s / y ] t₁) ([ s / y ] t₂)
```

Line 4 above propagates s under a λ that binds an x, which is only capture avoiding if we assume that s does not contain the variable x—or, simply, that s is *closed*.

Using the substitution function above, we can define a normalizer for closed terms which normalizes λ terms to *weak head normal form*, does not evaluate under λs, and errs in case it encounters a free variable. The normal forms (or *values*) computed by our normalizer are functions whose bodies may contain further normalizable terms; i.e.:

```
data Val : Set where
  lam : Name → Term → Val
```

The normalization function below uses the Maybe type to indicate that it either returns a value wrapped in a just or errs by yielding nothing if it encounters a free variable:

```
{-# NON_TERMINATING #-}
normalize : Term → Maybe Val
normalize (lam x t)   = just (lam x t)
normalize (var x)     = nothing
normalize (app t₁ t₂) = case (normalize t₁) of λ where
  (just (lam x t)) → case (normalize t₂) of λ where
    (just v) → normalize ([ V2T v / x ] t)
    _        → nothing
  _ → nothing
```

(Dec $(x \equiv y)$). The underscores in _≡?_ indicates that the function is written as infix syntax; i.e., its first argument is written to the left of ≡? and the second to the right.

[3] case_of_ is a mixfix function whose second argument is a pattern matching function. The λ **where** ... is a pattern matching function, where yes and no are the constructors Dec, each parameterized by a proof that two names are (un)equal. In this section we do not use these proofs; in § 3 we do.

The function uses an auxiliary function V2T : Val → Term which transforms values to terms. The **NON_TERMINATING** pragma disables Agda's termination checker because normalization of untyped λ terms may non-terminate.

Similar definitions as shown above can be found in many programming language educational texts and research papers. The definitions correctly normalize *closed* terms. If we apply normalize to *open* terms instead we may get wrong results. For example, the following term should normalize to the free variable y:

$$(\lambda f.\,(\lambda y.\,(f\ (\lambda one.\,one))))\ (\lambda z.\ \underbrace{y}_{\text{free variable}}\ )\ (\lambda two.\,two) \tag{2}$$

However, the normalize function *incorrectly* normalizes this term to $\lambda two.\,two$ instead. In the next section we present a simple renamingless capture avoiding substitution strategy which correctly normalizes the term above to y.

2.2 Normalizing Open λ Terms using Renamingless Substitution

The idea is to add a term which delimits and distinguishes those terms that have been computed to normal forms (values) already. Values represent terms where all substitutions from their lexically enclosing context have already been applied, so it is futile to propagate substitutions into values. The reason that traditional expositions of the untyped λ calculus rely on renaming of bound variables, is that they propagate substitutions into values. By distinguishing values, we avoid this pitfall, and thus the need for renaming.

The TermV data type below is the same as Term but has a distinguished val constructor for representing values given by a type parameter V : Set:

```
data TermV (V : Set) : Set where
  lam : Name → TermV V       → TermV V
  var : Name                 → TermV V
  app : TermV V → TermV V → TermV V
  val : V                    → TermV V
```

We can define a substitution function for TermV that is case-by-case the same as the substitution function in § 2.1, except that (1) it substitutes *values* (V) into terms, and (2) it has a case for value terms (val):[4] In other words, values are *closed*.

```
⟨_/_⟩_ : {V : Set} → V → Name → TermV V → TermV V
⟨ v / y ⟩ (lam x t) = case (x ≡? y) of λ where
  (yes p) → lam x t
  (no _) → lam x (⟨ v / y ⟩ t)
⟨ v / y ⟩ (var x) = case (x ≡? y) of λ where
  (yes _) → val v
  (no _) → var x
⟨ v / y ⟩ (app t₁ t₂) = app (⟨ v / y ⟩ t₁) (⟨ v / y ⟩ t₂)
⟨ v / y ⟩ (val u) = val u
```

The final case above says that we never substitute inside values. This way, free variables that occur in values are never captured because we never propagate substitutions into values.

[4] The curly braces {...} in the type signature of ⟨_/_⟩_ denotes an *implicit parameter* which does not need to be passed explicitly when we call the function. Agda will automatically infer what the parameter is.

As opposed to the substitution function in § 2.1, the function above accepts values rather than terms as its first argument. However, this suffices to define a normalizer to values in weak head normal form. Since terms may contain free variables, the notion of values for this normalizer is now *either* a function, a free variable, *or* an application whose sub-terms are also in values (i.e., weak head-normal forms):

```
data ValV : Set where
  lam : Name → TermV ValV → ValV
  var : Name               → ValV
  app : ValV → ValV        → ValV

{-# NON_TERMINATING #-}
normalizeV : TermV ValV → ValV
normalizeV (lam x t) = lam x t
normalizeV (var x)   = var x
normalizeV (app t₁ t₂) = case (normalizeV t₁) of λ where
  (lam x t) → normalizeV (⟨ normalizeV t₂ / x ⟩ t)
  v₁        → app v₁ (normalizeV t₂)
normalizeV (val v) = v
```

Unlike the normalizer in § 2.1, normalizeV is a *total* (but possibly non-terminating) function, which takes open untyped λ calculus terms as input and yields their weak head normal form as output. For example, normalizing the term labeled (2) yields the free variable y, as intended. Unlike the substitution functions and normalizers found in most educational texts and research papers in the literature, the normalizer above does not rely on renaming.

The substitutions performed by normalizeV are capture avoiding because we only ever propagate closed terms under λ binders. However, these closed terms may now contain free (but unsubstitutable) variables. The difference between the normalizer in § 2.1 and here is thus that our normalizer above has a more liberal notion of what it means for a term to be closed. The next section makes this intuition formal.

3 Intrinsically Scoped Renamingless Capture Avoiding Substitution

We introduce an intrinsic scoping discipline for the untyped λ calculus, inspired by *intrinsic typing* [3, 4]. This intrinsic scoping discipline explains how our renamingless capture avoiding substitution strategy and the normalizers that use it rely on a loose notion of what it means for a term to be closed; namely, closed terms may contain free (but unsubstitutable) variables.

Our approach to intrinsic scoping is inspired by the Agda standard library[5] and the work of, e.g., Allais et al. [2] and Rouvoet et al. [14].

3.1 Prelude to Intrinsic Typing

We will associate untyped λ terms with their set of free variables. For example, $\lambda x. y$ where $x \neq y$ is encoded as a term associated with the free variable set is $\{y\}$. We will encode a syntax that intrinsically associates a term with its set of free variables by making it impossible to define terms with any other association.

[5] E.g., `https://github.com/agda/agda-stdlib/blob/v1.7.1/src/Relation/Unary.agda`

```
FVPred = List Name → Set

_⇒_ : FVPred → FVPred → FVPred
(P ⇒ Q) xs = P xs → Q xs

∀[_] : FVPred → Set
∀[ P ] = {xs : List Name} → P xs

ϵ[_] : FVPred → Set
ϵ[ P ] = P []
```

```
record _∧_ (P Q : FVPred)
           (xs : List Name) : Set where
  constructor _∧_|_
  field {ys zs} : List Name
        px : P ys
        qx : Q zs
        ϕ    : xs ≡ ys ⊔ zs


_−_ : FVPred → Name → FVPred
(P − x) xs = P (xs \ [ x ])
```

Figure 1 Logical connectives for predicates over free variables

To this end, we will encode terms as *predicates over free variables*; i.e., the FVPred type in Figure 1 which uses a list of names to represent a (multi-)set of free variables. Figure 1 also introduces the logical connectives we will use to write concise type signatures and normalizers, akin to the ones in the previous section, but which Agda can check are safe-by-construction. We recommend that readers read § 3.2 and consult Figure 1 as needed.

The logical connectives in Figure 1 assume the existence of a union-like operation for lists of names _⊔_ : List *Name* → List *Name* → List *Name* with accompanying laws proving that the operation is *commutative* and *monoidal* (i.e., *associative* and the empty list is the *identity element* w.r.t. ⊔). It also assumes a difference-like operation __ : List *Name* → List *Name* → List *Name* with accompanying laws that characterize its difference-like nature (the laws can be found in the source code of the paper).

3.2 Normalizing Closed λ Terms using Intrinsic Typing

Using the operations in Figure 1, we define a data type of λ terms that is intrinsically typed by the set of free variables of the term:

```
data FV : List Name → Set where
  lam : (x : Name) → ∀[ (FV − x) ⇒ FV ]
  var : (x : Name) → ∀[ One x      ⇒ FV ]
  app : ∀[ (FV ∧ FV)               ⇒ FV ]
```

The only inhabitants of the type FV *xs* are terms whose set of free variables is exactly *xs*.

Using the FV type, we can refine the type of the substitution function from § 2.1 to make explicit the assumptions about closedness that were previously implicit. The type and implementation of the function is given below, where each ⋯ represents an elided (but straightforward) Agda proof term which uses the laws about the ⊔ and \ operations to prove to Agda that the intrinsic typing is valid.[6]

```
⟦_/_⟧_ : ϵ[ FV ] → (x : Name) → ∀[ FV ⇒ (FV − x) ]
⟦ s / y ⟧ (lam x t) = case (x ≡? y) of λ where
  (yes ϕ)  → lam x $ t : FV | ⋯
```

[6] The $ operation is an infix operation for function application (akin to the operation by the same name in Haskell).

```
  (no ¬φ) → lam x $ (⟦ s / y ⟧ t) : FV | ···
⟦ s / y ⟧ (var x φ₁) = case (x ≡? y) of λ where
  (yes φ₂)  → s : FV | ···
  (no ¬φ₂) → var x $ ···
```

The type signature of the substitution function above says that the term being substituted for has no free variables (ϵ[FV]), and that the final set of free variables is the set of free variables of the term being substituted in minus the variable x that was substituted ($\forall$[FV $\Rightarrow$ (FV $-$ x)]); i.e., no variables are captured.

The normalizer from § 2.1 can be similarly generalized to show that normalizing a closed term is guaranteed to yield a normal form (value), where a normal form is a (closed) λ value given by the NF type:

```
data Val : Set where
  lam : (x : Name) → ϵ[ FV − x ] → Val
```

The type signature of the generalized normalizer is given below. Its definition is case-by-case similar to the normalizer from § 2.1, and is elided for brevity.

```
{-# NON_TERMINATING #-}
normalize : ϵ[ FV ] → Val
```

Unlike the normalizer from § 2.1, which was partial, normalize is a *total*, possibly non-terminating function, because the type ϵ[FV] intrinsically guarantees that the input term has no free variables.

3.3 Normalizing Open λ Terms using Intrinsic Typing

We now show that our normalizer from § 2.2 provides similar guarantees as the normalizer for closed terms in § 3.2. To this end we enrich the FV type from before by an additional constructor for values given by a type parameter V : Set:[7]

```
data FVV (V : Set) : List Name → Set where
  lam : (x : Name) → ∀[ (FVV V − x) ⇒ FVV V ]
  var : (x : Name) → ∀[ One x         ⇒ FVV V ]
  app : ∀[ (FVV V ∧ FVV V)            ⇒ FVV V ]
  val : ϵ[ const V                    ⇒ FVV V ]
```

Crucially, the val case says that values have *no free variables*. However, below we will use a notion of value that may contain free variables. But since these free variables are delimited by a value, they are *unsubstitutable*.

Using this type of terms, we define a substitution function with a similar type signature as the substitution function from § 3.2. It also has similar cases which we elide, except for the case for the val constructor:[8]

```
⟨_/_⟩_ : {V : Set} → V → (x : Name) → ∀[ FVV V ⇒ (FVV V − x) ]
```

[7] Here const : $\{A\ B$: Set$\} \to A \to B \to A$ is the *constant function* which ignores its second argument, and always returns its first argument.

[8] The _ in FVV _ represents a term that we ask Agda to automatically infer for us. In this case, Agda infers that it is the implicitly parameter type V : Set.

```
...
⦇ v / y ⦈ (val u) = val u : FVV _ | ···
```

As with the substitution function from § 2.1, we do not propagate substitutions into values. Thus the only difference between the substitution function in § 3.2 and the substitution function above is that the function above has a more liberal notion of what it means for a term to be closed; namely, it is either a plain closed term, or a value.

Using this substitution function we generalize the type of the normalizer from § 2.2 to operate on intrinsically typed terms. Unlike the normalizer in § 3.2, the normalizer below takes *open terms* as input and normalizes these to weak head normal forms:

```
data ValV : Set where
  lam : (x : Name) → ∀[ (FVV ValV ─ x) ⇒ const ValV ]
  var : Name                           → ValV
  app : ValV → ValV                    → ValV
```

The normalizer is given by the normalizeV function:

```
{-# NON_TERMINATING #-}
normalizeV : ∀[ FVV ValV ⇒ const ValV ]
normalizeV (app (t₁ ∧ t₂ | ϕ)) = case (normalizeV t₁) of λ where
  (lam x t) → normalizeV $ (⦇ (normalizeV t₂) / x ⦈ t) : FVV _ | ···
  v₁        → app v₁ (normalizeV t₂)
normalizeV (lam x t) = lam x t
normalizeV (val v)   = v
normalizeV (var x ϕ) = var x
```

The function is case-by-case similar to the function from § 2.2. However, thanks to its intrinsic typing information, we are guaranteed that (1) normalization only ever applies substitutions that are capture avoiding, since the substitution function only propagates closed terms past λ bindings; and (2) normalization may yield values that correspond to open terms. All without any fiddly renaming.

4 Conclusion and Future Directions

We have presented a technique for capture avoiding substitution that does not require renaming of bound variables. The technique results in substitution functions that perform capture avoiding substitution involving open terms, but which is as simple to implement and understand as substitution functions involving only closed terms. This makes this style of substitution attractive for, e.g., teaching and learning the untyped λ calculus. By intrinsically typing substitution functions we have shown that our technique is indeed capture avoiding.

This paper only considers normalization to weak head normal form, using a *call-by-value* normalization strategy. We conjecture that the techniques are equally applicable to *call-by-name normalization* strategies, as well as normalization to stronger normal forms. We leave verification of this conjecture as future work.

References

1 Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *J. Funct. Program.*, 1(4):375–416, 1991. doi:10.1017/S0956796800000186.

2 Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type- and scope-safe universe of syntaxes with binding: their semantics and proofs. *J. Funct. Program.*, 31:e22, 2021. `doi:10.1017/S0956796820000076`.

3 Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 1999. `doi:10.1007/3-540-48168-0\_32`.

4 Lennart Augustsson and Magnus Carlsson. An exercise in dependent types: A well-typed interpreter. In *In Workshop on Dependent Types in Programming, Gothenburg*, 1999.

5 Hendrik Pieter Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985.

6 Arthur Charguéraud. The locally nameless representation. *J. Autom. Reason.*, 49(3):363–408, 2012. `doi:10.1007/s10817-011-9225-2`.

7 Haskell B. Curry and Robert Feys. *Combinatory Logic*. Combinatory Logic. North-Holland Publishing Company, 1958.

8 N.G de Bruijn. Lambda calculus notation with nameless dummies. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. `doi:10.1016/1385-7258(72)90034-0`.

9 Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992. `doi:10.1016/0304-3975(92)90014-7`.

10 Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems*. MIT Press, 2014. URL: `https://mitpress.mit.edu/books/modeling-and-analysis-communicating-systems`.

11 P. J. Landin. The mechanical evaluation of expressions. *Comput. J.*, 6(4):308–320, 1964. `doi:10.1093/comjnl/6.4.308`.

12 Lena Pareto. *The Implementation of ALF—a Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. PhD thesis, University of Gothenburg, 1995.

13 Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975. `doi:10.1016/0304-3975(75)90017-1`.

14 Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for linear, session-typed languages. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 284–298. ACM, 2020. `doi:10.1145/3372885.3373818`.