Renamingless Capture-Avoiding Substitution, Intrinsically

Casper Bach Poulsen

Delft University of Technology, Netherlands
c.b.poulsen@tudelft.nl

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 describe an intrinsic typing discipline for untyped λ terms which we use to verify that our substitution technique is, indeed, capture avoiding.

1 Introduction

As argued by Accattoli [2], "the λ-calculus is as old as computer science"; still, it remains a relevant subject of study thanks to its far-reaching applications in both theory and practice. For the same reason, the λ-calculus remains an important part of computer science curricula around the world.

A key stumbling block of learning and implementing 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 the underbraced y is a *free variable*; i.e., a variable that is not bound by an enclosing λ term. However, if we use 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, meaning it is no longer a free variable.

Following, e.g., Curry and Feys [8], Plotkin [12], or Barendregt [5], the common technique to avoid such name capture is to rename variables during substitution. Using renaming, normalization of the term in Equation (1) yields the correct intermediate reduct $(\lambda r.\, ((\lambda z.\, y)\ 1) + r)\ 2$.

However, defining substitution functions that do such renaming is fiddly. For this reason, 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. But for some applications, such as implementing dependent type checking [11], it can be desirable to work on λ terms with free variables.

There exists alternative techniques that can be used to define (lazy) capture avoiding substitution, such as *closures and environments* [10], *de Bruijn indices* [6], *explicit substitutions* [1], and *locally nameless* [7]. However, traditional naive substitution functions are often preferable for teaching the λ-calculus, because they provide an intuitive framework for understanding the concept that the aforementioned alternative techniques encode.

In recent years, Eelco Visser and I have been teaching the λ-calculus at the undergraduate students by having them implement definitional interpreters and substitution functions for λ terms using a technique for capture avoiding substitution that does not require fiddly renaming of bound variables. The idea is to distinguish those terms in abstract syntax trees (ASTs) that have already been computed to normal forms, and to never substitute inside of those terms. The benefit of this approach is that it makes it as simple to understand and implement substitutions for *open* terms (i.e., terms that may contain free variables) as it is 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 proves that the resulting substitution functions are capture avoiding. We make the following technical contributions:

- We present a technique (§ 2) for capture avoiding substitution that does not require renaming of bound variables, and that is about as simple to understand and implement as substitution for closed terms.
- We present an intrinsically capture avoiding normalizer (§ 3) for the untyped λ-calculus, which provides proof that the technique is, indeed, capture avoiding.

2 Renamingless Capture Avoiding Substitution

We present our technique for renamingless capture avoiding substitution 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 include footnotes to explain Agda specific syntax. Familiarity with dependent types is not needed to read this section, but is needed for § 3.

We first implement a normalizer for *closed* terms (in § 2.1) using a standard renamingless substitution function, and then use our technique to generalize this to a normalizer for *open* terms (in § 2.2) that uses an equally simple and renamingless substitution function.

[1] This judgement is made solely by the author of the present paper. I never had the chance to discuss the novelty of the technique with Eelco.

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]

```
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)
```

The reason this substitution function assumes that the term being substituted for is closed is that we could otherwise get variable capture when propagating a substitution under a λ binder, as illustrated in the introduction.

Using the substitution function above, we can now define a normalizer which also assumes that the input term is closed, and yields an error in case it encounters a free variable. It normalizes λ terms to *weak head normal form*, and does not evaluate under λs. Thus, the normal forms computed by our normalizer are simply functions whose bodies may contain further normalizable terms:

[2] In Agda, Set is the type of types. So *Name* : Set declares a type parameter. The _≡?_ parameter has a *dependent type*: it takes two names as input, where the return type depends on these names. The type Dec (x ≡ y) represents a proof that x and y are (un)equal. The use of underscores on the left hand side of a function declarations declares the function as mixfix syntax. For example, _≡?_ is an infix function whose first argument is written to the left of ≡? and whose second argument is to the right.

[3] case_of_ is a mixfix function whose second argument is a pattern matching function. The λ **where** ... is Agda syntax for a pattern matching function, and yes and no are the two constructors of the Dec type, each parameterized by a proof that the two names are (un)equal. The substitution function in this section does not make use of these proofs; however, in § 3 we will.

```
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
```

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 what we have shown in this section can be found in many programming language educational texts and research papers. The definitions let us correctly normalize *closed* terms. If we attempt to use normalize to normalize *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 our technique which performs capture avoiding substitution to correctly normalize the term above, without relying on fiddly renaming during normalization.

2.2 Normalizing Open λ Terms using Renamingless Substitution

The idea is to enrich our notion of term to distinguish those terms that have been computed to normal forms (values). The motivation for distinguishing values, is that values represent terms where all substitutions from their the lexically enclosing context have already been applied. It is futile (and morally wrong) 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 it also has a case for values (val). This case says that we never substitute inside values.[4]

```
⟨_/_⟩_ : {V : Set} → TermV V → Name → TermV V → TermV V
⟨ s / y ⟩ (lam x t) = case (x ≡? y) of λ where
  (yes p) → 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)
⟨ s / y ⟩ (val v) = val v
```

This substitution function still assumes that the term being substituted for (i.e., the first parameter of the function) is closed. However, since values may contain free variables, the substitution function does support substitution involving open terms. Free variables in values are never be captured because we never propagate substitutions into values.

Using these definitions, we define a normalizer to values in weak head normal form, which is now either a function, a free variable, or an application whose sub-terms are also in weak head-normal form:

```
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 (⟨ val (normalizeV t₂) / x ⟩ t)
  v₁        → app v₁ (normalizeV t₂)
normalizeV (val v) = v
```

Unlike the normalizer in § 2.1, normalizeV is a *total*, possibly non-terminating function, which takes open untyped λ calculus terms as input and yields their

[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.

weak head normal form as output. 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, but does do capture avoiding substitution. For example, normalizing the term in Equation (2) yields the free variable y, as intended.

More generally, any substitution performed by normalizeV is going to be capture avoiding because it only ever performs substitution of *closed* terms under λ binders. The difference between the normalizer in § 2.1 and here is 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 Renamingless Capture Avoiding Substitution using Intrinsic Typing

We show that both of the substitution functions from § 2 are capture avoiding because they only ever substitute closed terms under λ binders. We do so by strengthening the type of our substitution functions and normalizers to reflect the invariants that they are subject to, using *intrinsic typing* [3, 4]. Our approach to intrinsic typing is inspired by the Agda standard library[5] and the work of Rouvoet et al. [13].

3.1 Prelude to Intrinsic Typing

We will be intrinsically typing untyped λ terms by their set of free variables. For example, $\lambda x.\, y$ where $x \neq y$ will be typed as a term whose free variable set is $\{y\}$. On other words, terms will be given by *predicates over free variables*. Figure 1 introduces some logical connectives for such predicates. We will make use of these connectives to write concise type signatures and normalizers, akin to the ones in the previous section, but which Agda can check are safe-by-construction. For example, in § 3.2 we use the connectives to assert and verify that the closed substitution function from § 2.1 only substitutes for closed terms, that substitution eliminates a free variable, and that normalization of closed terms only applies closed substitutions to compute closed λ terms as a result. In § 3.3 we apply the same approach to the substitution function and normalizer from § 2.2.

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).

[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 []

One : Name → List Name → Set
One x xs = xs ≡ [ x ]
```

```
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 ])
```

Fig. 1. Logical connectives for predicates over free variables

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 | ⋯
  (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

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

variables is the set of free variables of the term being substituted in minus the variable x that was substituted ($\forall[\ \mathsf{FV} \Rightarrow (\mathsf{FV} - x)\ ]$). Agda automatically checks for us that the substitution function inhabits this type, thereby showing that the substitution function is capture avoiding by construction.

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 $\epsilon[\ \mathsf{FV}\ ]$ 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 ]
```

The val case says that values have *no free variables*.

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}
       → ϵ[ FVV V ] → (x : Name) → ∀[ FVV V ⇒ (FVV V − x) ]
⟪ s / y ⟫ (val v) = val v : FVV _ | ⋯
```

[7] Here const : $\{A\ B : \mathsf{Set}\} \to A \to B \to A$ 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.

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 given by the following type:

```
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 $ (⦇ val (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.

Bibliography

[1] Abadi, M., Cardelli, L., Curien, P., Lévy, J.: Explicit substitutions. J. Funct. Program. **1**(4), 375–416 (1991), https://doi.org/10.1017/S0956796800000186

[2] Accattoli, B.: A fresh look at the lambda-calculus (invited talk). In: Geuvers, H. (ed.) 4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany, LIPIcs, vol. 131, pp. 1:1–1:20, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019), https://doi.org/10.4230/LIPIcs.FSCD.2019.1

[3] Altenkirch, T., Reus, B.: Monadic presentations of lambda terms using generalized inductive types. In: Flum, J., Rodríguez-Artalejo, M. (eds.) Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings, Lecture Notes in Computer Science, vol. 1683, pp. 453–468, Springer (1999), https://doi.org/10.1007/3-540-48168-0_32

[4] Augustsson, L., Carlsson, M.: An exercise in dependent types: A well-typed interpreter. In: In Workshop on Dependent Types in Programming, Gothenburg (1999)

[5] Barendregt, H.P.: The lambda calculus - its syntax and semantics, Studies in logic and the foundations of mathematics, vol. 103. North-Holland (1985)

[6] de Bruijn, N.: Lambda calculus notation with nameless dummies. Indagationes Mathematicae (Proceedings) **75**(5), 381392 (1972), ISSN 1385-7258, https://doi.org/10.1016/1385-7258(72)90034-0

[7] Charguéraud, A.: The locally nameless representation. J. Autom. Reason. **49**(3), 363–408 (2012), https://doi.org/10.1007/s10817-011-9225-2

[8] Curry, H.B., Feys, R.: Combinatory Logic. Combinatory Logic, North-Holland Publishing Company (1958)

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

[10] Landin, P.J.: The mechanical evaluation of expressions. Comput. J. **6**(4), 308–320 (1964), https://doi.org/10.1093/comjnl/6.4.308

[11] Pareto, L.: The Implementation of ALF—a Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution. Ph.D. thesis, University of Gothenburg (1995)

[12] Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. **1**(2), 125–159 (1975), https://doi.org/10.1016/0304-3975(75)90017-1

[13] Rouvoet, A., Bach Poulsen, C., Krebbers, R., Visser, E.: Intrinsically-typed definitional interpreters for linear, session-typed languages. In: Blanchette, J., Hritcu, C. (eds.) Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA,

USA, January 20-21, 2020, pp. 284–298, ACM (2020), https://doi.org/10.1145/3372885.3373818