Renamingless Capture-Avoiding Substitution for Definitional Interpreters

Casper Bach Poulsen
Delft University of Technology, Netherlands

Abstract

Substitution is a common and popular approach to implementing name binding in definitional interpreters. A common pitfall of implementing substitution functions is *variable capture*. The traditional approach to avoiding variable capture is to rename variables. However, traditional renaming makes for an inefficient interpretation strategy. Furthermore, for applications where partially-interpreted terms are user facing it can be confusing if names in uninterpreted parts of the program have been changed. In this paper we explore two techniques for implementing capture avoiding substitution in definitional interpreters in a way that avoids renaming.

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

Keywords and phrases Capture-avoiding substitution, Untyped lambda calculus, Definitional interpreter

Digital Object Identifier 10.4230/OASIcs.CVIT.2016.23

1 Introduction

Following Reynolds [21], a definitional interpreter is an important and frequently used method of defining a programming language, by giving an interpreter for the language that is written in a second, hopefully better understood language. The method is widely used both for programming language research [3, 4, 13, 18, 22] and teaching [15, 19, 23]. A commonly used approach to defining name binding in such interpreters is *substitution*. A key stumbling block when implementing substitution is how to deal with *name capture*. The issue is illustrated by the following untyped λ term:

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

This term does *not* evaluate 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 an interpreter state corresponding to the following (wrong) intermediate term $(\lambda y.\, ((\lambda z.\, y)\ 1) + y)\ 2$ where the red y is *captured*; that is, it is no longer a free variable.

Following, e.g., Curry and Feys [12], Plotkin [20], or Barendregt [5], the common technique to avoid such name capture is to *rename* variables either before or during substitution (a process known as α-*conversion* [11]). For example, by renaming the λ bound variable y to r, we can correctly reduce term (1) to $(\lambda r.\, ((\lambda z.\, y)\ 1) + r)\ 2$.

While a renaming based substitution strategy provides a well behaved and versatile approach to avoiding name capture, it has some trade-offs. For example, since renaming typically works by fully traversing terms, definitional interpreters that use renaming based substitution are typically relatively slow. Another trade-off is that renaming gives rise to intermediate terms whose names differ from the names in source programs. For applications where intermediate terms are user facing (e.g., in error messages, or in systems based on rewriting) this can be confusing. For this reason, definitional interpreters often use alternative techniques for (lazy) capture avoiding substitution, such as *closures* [16], *de*

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:11
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Bruijn indices [14], *explicit substitutions* [1], or *locally nameless* [9]. However, traditional named variable substitution is sometimes preferred because intermediate terms are easy to inspect and compare. This paper considers and explores named substitution strategies that do not rely on renaming variables. We explore two possible solutions to this problem, neither of which seem to be widely known or at least not widely used.

The first technique we explore is a technique that Eelco Visser and I were using to teach students about static scoping, by having students implement definitional interpreters. To this end, we used a simple renamingless substitution strategy which (for applications that do not perform evaluation under binders) is capture avoiding. The idea is to delimit and never substitute into those terms in abstract syntax trees (ASTs) where all substitutions that were supposed to be applied to the term, have been applied; e.g., terms that have been computed to normal form. 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 step-wise evaluation:

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

The result term computed by these reduction steps is equivalent to using a renaming based substitution function. However, the renamingless substitution strategy we used does not rename variables (and so preserves the names of bound variables in programs), is simple to implement, and is more efficient than interpreters that rename variables at run time.

I never had the chance to discuss the novelty of the technique with Eelco. However, the technique we used in the course does not seem widely known or used. In this paper we explain and explore the technique and its limitations. The main known limitation of using the technique for defining interpreters is that it assumes an interpretation strategy that does not do evaluation under binders. For the toy language interpreters we used for teaching this was not a problem; for more serious languages and applications it may be.

The second technique for capture-avoiding named substitution that we explore is an existing technique which we were made aware of by a reviewer of a previous version of this paper. The technique is due to Berkling and Fehr [7] and has similar benefits as the technique we used in our course: it does not rename variables and is also more efficient than interpreters that rename variables at run time. Furthermore, the technique does not make assumptions on behalf of interpretation strategy, and it supports evaluation under binders. On the other hand, Berkling and Fehr's substitution technique is more involved to implement and is a little less efficient than the renamingless substitution strategy that we used in our course.

The renamingless techniques we consider in this paper are not new (at least the second technique is not; we do not expect that the first one is either, though we have not found it in the literature). But we believe they deserve to be more widely known. Our contributions are:

- We describe (§ 2) a simple, renamingless substitution technique for languages with open terms where evaluation does not happen under binders. The meta-theory of this technique is left for future work, but we discuss and illustrate known limitations in terms of examples.
- We describe (§ 3) an existing and more general technique [7] which has similar benefits

and does not suffer from the same limitations. However, its implementation is a little more involved to implement than the simple renamingless substitution strategy, and it is a little less efficient.

This paper is a literate Haskell document, available at `https://github.com/casperbp/renamingless-capture-avoiding`, and is structured as follows. § 2 describes a simple renamingless capture avoiding substitution strategy and its known limitations and § 3 describes Berkling-Fehr substitution which has similar benefits and fewer limitations but is less simple to implement. § 4 discusses related work and § 5 concludes.

2 Renamingless Capture-Avoiding Substitution

We present a simple technique for capture avoiding substitution, which avoids the need to rename bound variables. To demonstrate that the technique is about as simple to implement as substitution for closed terms (i.e., terms with no free variables, for which variable capture is not a problem), we first implement a standard substitution-based definitional interpreter for a language with closed, call-by-value λ expressions.

2.1 Interpreting Closed Expressions

Below left is a data type for the abstract syntax of a language with λs, variables, applications, and numbers. On the right is the substitution function for the language. The function binds three parameters: (1) the variable name (*String*) to be substituted, (2) the expression the name should be replaced by, and (3) the expression in which substitution happens.

```
data Expr0
  = Lam0 String Expr0
  | Var0 String
  | App0 Expr0 Expr0
  | Num0 Int
```

```
subst0 :: String → Expr0 → Expr0 → Expr0
subst0 x s (Lam0 y e)   | x ≡ y     = Lam0 y e
                        | otherwise = Lam0 y (subst0 x s e)
subst0 x s (Var0 y)     | x ≡ y     = s
                        | otherwise = Var0 y
subst0 x s (App0 e1 e2) = App0 (subst0 x s e1) (subst0 x s e2)
subst0 _ _ (Num0 z)     = Num0 z
```

The main interesting case is the case for Lam_0. There are two sub-cases, declared using *guards* (the Boolean expressions after the vertical bar). The first sub-case is when the variable being substituted matches the bound variable ($x \equiv y$). Since the inner variable shadows the outer, the substitution is not propagated into the body. In the other case (*otherwise*), the substitution is propagated. This other case relies on an implicit assumption that the expression being substituted by x does not have y as a free variable. If we violate this assumption, the substitution function and interpreter $interp_0$ on the left below is not going to be capture avoiding. Below right is an example invocation of the interpreter.

```
interp0 :: Expr0 → Expr0
interp0 (Lam0 x e)   = Lam0 x e
interp0 (Var0 _)     = error "Free variable"
interp0 (App0 e1 e2) = case interp0 e1 of
  Lam0 x e → interp0 (subst0 x (interp0 e2) e)
  _        → error "Bad application"
interp0 (Num0 z)     = Num0 z
```

```
> interp0 (App0 (Lam0 "x" (Var0 "x"))
                (Num0 42))
Num0 42
```

2.2 Intermezzo: Capture-Avoiding Substitution Using Renaming

The substitution function $subst_0$ relies on an implicit assumption that expressions are closed; i.e., do not contain free variables. If we want to support *open expressions* (i.e., expressions that may contain free variables), we must take care to avoid variable capture. A traditional approach [20] is to rename variables during interpretation. Let $subst_{01}$ be a function whose cases are the same as $subst_0$, except for the Lam_0 case:

$$
\begin{array}{llll}
subst_{01}\ x\ s\ (Lam_0\ y\ e) & \mid x \equiv y & = Lam_0\ y\ e & \\
 & \mid otherwise & = \mathbf{let}\ z = fresh\ x\ y\ s\ e & \\
 & & \quad\ \mathbf{in}\ Lam_0\ z\ (subst_{01}\ x\ s\ (subst_{01}\ y\ (Var_0\ z)\ e)) &
\end{array}
$$

Here $fresh\ x\ y\ s\ e$ is a function that returns a fresh identifier if $x \notin FV(e)$ or $y \notin FV(s)$, or returns y otherwise. While this renaming based substitution strategy provides a relatively conceptually straightforward solution to the name capture problem, it requires an approach to generating fresh variables, and, since it performs two recursive calls to $subst_{01}$, it is inherently less efficient than the substitution function from § 2.1—even in a lazy language like Haskell. Furthermore, depending on how *fresh* is implemented, the interpreter may not preserve the names of λ-bound variables. In the next section we introduce an simple alternative substitution strategy which does not rename or generate fresh variables, and which has similar efficiency as substitution for closed expressions. The substitution strategy is capture-avoiding for languages that do not evaluate under binders.

2.3 Interpreting Open Expressions with Renamingless Substitution

Let us revisit the interpretation function $interp_0$ from § 2.1. Because our interpreter eagerly applies substitutions whenever it can, and because evaluation always happens at the top-level, never under binders, we know the following. Whenever the interpreter reaches an application expression $e_1\ e_2$, we know that *any variable that occurs free in* e_2 *corresponds to a variable that was free to begin with.* The same goes for the expressions resulting from interpreting e_2. We can exploit this knowledge in our interpreter and substitution function. To this end, we introduce a dedicated expression form (the highlighted Clo_1 constructor below) which delimits expressions that have been closed under substitutions such that we never propagate substitutions past this closure delimiter:

$$
\begin{array}{l}
\mathbf{data}\ Expr_1 \\
\quad = Lam_1\ String\ Expr_1 \\
\quad \mid\ Var_1\ String \\
\quad \mid\ App_1\ Expr_1\ Expr_1 \\
\quad \mid\ Num_1\ Int \\
\quad \mid\ Clo_1\ Expr_1
\end{array}
$$

$$
\begin{array}{llll}
\multicolumn{4}{l}{subst_1 :: String \to Expr_1 \to Expr_1 \to Expr_1} \\
subst_1\ x\ s\ (Lam_1\ y\ e) & \mid x \equiv y & = Lam_1\ y\ e \\
 & \mid otherwise & = Lam_1\ y\ (subst_1\ x\ s\ e) \\
subst_1\ x\ s\ (Var_1\ y) & \mid x \equiv y & = s \\
 & \mid otherwise & = Var_1\ y \\
\multicolumn{4}{l}{subst_1\ x\ s\ (App_1\ e_1\ e_2) = App_1\ (subst_1\ x\ s\ e_1)\ (subst_1\ x\ s\ e_2)} \\
subst_1\ \_\ \_\ (Num_1\ z) & & = Num_1\ z \\
subst_1\ \_\ \_\ (Clo_1\ e) & & = Clo_1\ e
\end{array}
$$

Here $subst_1$ does not propagate substitutions into expressions delimited by Clo_1. The interpretation function $interp_1$ uses Clo_1 to close expressions before substituting (in the App_1 case), thereby avoiding name capture:

$$
\begin{array}{ll}
\multicolumn{2}{l}{interp_1 :: Expr_1 \to Expr_1} \\
interp_1\ (Lam_1\ x\ e) & = Lam_1\ x\ e \\
interp_1\ (Var_1\ x) & = Var_1\ x
\end{array}
$$

$$
\begin{aligned}
&\mathit{interp}_1\ (\mathit{App}_1\ e_1\ e_2) = \textbf{case}\ \mathit{interp}_1\ e_1\ \textbf{of} \\
&\quad \mathit{Lam}_1\ x\ e \to \mathit{interp}_1\ (\mathit{subst}_1\ x\ (\mathit{Clo}_1\ (\mathit{interp}_1\ e_2))\ e) \\
&\quad e_1' \qquad\quad\ \to \mathit{App}_1\ e_1'\ (\mathit{interp}_1\ e_2) \\
&\mathit{interp}_1\ (\mathit{Num}_1\ z) \qquad = \mathit{Num}_1\ z \\
&\mathit{interp}_1\ (\mathit{Clo}_1\ e) \qquad = e
\end{aligned}
$$

Whereas interp_0 explicitly crashes when encountering a free variable or when attempting to apply a non-function to a number, interp_1 may return a "stuck" term in case it encounters a free variable or an application expression that attempts to apply a value other than a function. The last case of interp_1 says that, when the interpreter encounters a closed expression, it "unpacks" the closure. This unpacking will not cause accidental capture: because interpretation only happens at the top-level, never under binders, unpacking can never cause variable capture!

To illustrate how interp_1 works, let us consider how to interpret $((\lambda f.\, \lambda y.\, f\ 0)\ (\lambda z.\, y)\ 1)$. The rewrites below informally illustrate the interpretation process, where for brevity we use λ notation instead of the corresponding constructors in Haskell and $\lfloor e \rfloor$ instead of $\mathit{Clo}_1\ e$:

$$
\begin{aligned}
&\ \ \mathit{interp}_1\ ((\lambda f.\, \lambda y.\, f\ 0)\ (\lambda z.\, y)\ 1) \\
&\equiv \mathit{interp}_1\ ((\lambda y.\, \lfloor(\lambda z.\, y)\rfloor\ 0)\ 1) \\
&\equiv \mathit{interp}_1\ (\lfloor(\lambda z.\, y)\rfloor\ 0) \\
&\equiv y
\end{aligned}
$$

Unlike the renaming based substitution strategy discussed in § 2.2, our renamingless substitution strategy does not require renaming or generating fresh variables. Its efficiency is similar as substitution for closed expressions. It also preserves the names of binders. However, the renamingless substitution strategy in subst_1 and interp_1 relies on an assumption that evaluation does not happen under binders.

2.4 Limitation: Renamingless Substitution Does Not Support Evaluation Under Binders

The renamingless substitution strategy from § 2.3 assumes that the terms being closed have been closed under *all substitutions of variables bound in the context.* Interpretation strategies that evaluate under binders violate this assumption. For example, consider the interpreter given by $\mathit{normalize}_1$ whose highlighted recursive call performs evaluation under a λ binder:

$$
\begin{aligned}
&\mathit{normalize}_1 :: \mathit{Expr}_1 \to \mathit{Expr}_1 \\
&\mathit{normalize}_1\ (\mathit{Lam}_1\ x\ e) \quad = \mathit{Lam}_1\ x\ (\mathit{normalize}_1\ e) \\
&\mathit{normalize}_1\ (\mathit{Var}_1\ x) \qquad = \mathit{Var}_1\ x \\
&\mathit{normalize}_1\ (\mathit{App}_1\ e_1\ e_2) = \textbf{case}\ \mathit{normalize}_1\ e_1\ \textbf{of} \\
&\quad \mathit{Lam}_1\ x\ e \to \mathit{normalize}_1\ (\mathit{subst}_1\ x\ (\mathit{Clo}_1\ (\mathit{normalize}_1\ e_2))\ e) \\
&\quad e_1' \qquad\quad\ \to \mathit{App}_1\ e_1'\ (\mathit{normalize}_1\ e_2) \\
&\mathit{normalize}_1\ (\mathit{Num}_1\ z) \qquad = \mathit{Num}_1\ z \\
&\mathit{normalize}_1\ (\mathit{Clo}_1\ e) \qquad = e
\end{aligned}
$$

Just like interp_1, the $\mathit{normalize}_1$ function closes off terms before substituting. However, because $\mathit{normalize}_1$ evaluates under λ binders, closures may be prematurely unpacked, which may result in variable capture. For example, say we apply $(\lambda x.\, \lambda y.\, x)$ to the free variable

y. We would expect the result of evaluating this application to contain y as a free variable. However, using $normalize_1$, the free variable y is captured:

$$\begin{aligned} & normalize_1\ ((\lambda x.\, \lambda y.\, x)\ y) \\ \equiv\ & normalize_1\ (\lambda y.\, \lfloor y \rfloor) \\ \equiv\ & \lambda y.\, normalize_1\ \lfloor y \rfloor \\ \equiv\ & \lambda y.\, y \end{aligned}$$

The next section discusses a more general substitution strategy due to Berkling and Fehr [7] which does not have this limitation, which does not rename variables, and which is more efficient than the renaming based approach in § 2.2.

3 Berkling-Fehr Substitution

Motivated by how to implement a functional programming language based on Church's λ-calculus [10], Berkling and Fehr [7] introduced a modified version of Church's λ-calculus which uses a different kind of name binding and substitution. The key idea is to use a special operator (#) that acts on variables to neutralize the effect of one λ binding. For example, in the term $\lambda x.\, \lambda x.\, \#x$ the sub-term $\#x$ is a variable that references the *outermost* binding of x, whereas in $\lambda x.\, \lambda y.\, \#x$ the sub-term $\#x$ is a free variable.

Berkling and Fehr's # operator is related to De Bruijn indices [14] insofar as $\#^n x$ acts like an index that tells us to move n binders of x outwards. Indeed, if we were to restrict programs in Berkling and Fehr's calculus to use exactly one name, Berkling-Fehr substitution coincides with De Bruijn substitution. However, whereas De Bruijn indices can be notoriously difficult for humans to read (especially for beginners), Berkling-Fehr uses named variables such that indices only appear for substitutions that would otherwise have variable capture. This makes Berkling-Fehr variables easier to read for humans.

The definitions of shifting and substitution which we summarize in this section are taken from the work Berkling and Fehr [7] with virtually no changes. However, the language we implement is slightly different: they implement a modified λ-calculus with a call-by-name semantics, whereas we implement the same call-by-value language as in § 2. Our purpose of replicating their work is two-fold: to increase the awareness of Berkling-Fehr substitution and its seemingly nice properties, and to facilitate comparison with the renamingless approach we presented in § 2.3.

3.1 Interpreting Open Expressions with Berkling-Fehr Substitution

Below (left) is a syntax for λ expressions similarly to earlier, but now with Berkling-Fehr indices (right) instead of variables, where *Nat* is the type of natural numbers:

$$\begin{aligned} & \textbf{data}\ Expr_2 \\ & \quad = Lam_2\ String\ Expr_2 \\ & \quad \mid\ Var_2\ Index \\ & \quad \mid\ App_2\ Expr_2\ Expr_2 \\ & \quad \mid\ Num_2\ Int \end{aligned}$$

$$\textbf{data}\ Index = I\ \{ depth :: Nat, name :: String \}$$

Here the (record) data constructor $I\ n\ x$ corresponds to an n-ary application of the special # operator to the name x; i.e., $\#^n x$. We will refer to the n in $I\ n\ x$ as the *depth* of an index. As discussed above, a Berkling-Fehr index is similar to a De Bruijn index except that whereas a De Bruijn index tells us how many scopes to move out in order to locate a binder, a Berkling-Fehr index tells us how many scopes *that bind the same name* to move out in order to locate a binder. In what follows, we will sometimes use λ notation as informal syntactic sugar for the constructors in Haskell above. When doing so, we use “naked” variables x as informal syntactic sugar for a variable at depth 0; i.e., $\mathit{Var}_2\ (I\ 0\ x)$.

To define Berkling-Fehr substitution, we need a notion of *shifting*. Shifting is used when we propagate a substitution, say $x \mapsto e$ where x is a name and e is an expression, under a binder y. To this end, a shift increments the depth of all free occurrences of y in s by one. Such shifting guarantees that free occurrences of y in s are not accidentally captured.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{shift} :: \mathit{Index} \to \mathit{Expr}_2 \to \mathit{Expr}_2} \\
\mathit{shift}\ i\ (\mathit{Lam}_2\ x\ e) & \mid \mathit{name}\ i \equiv x & = \mathit{Lam}_2\ x\ (\mathit{shift}\ (\mathit{inc}\ i)\ e) \\
 & \mid \mathit{otherwise} & = \mathit{Lam}_2\ x\ (\mathit{shift}\ i\ e) \\
\mathit{shift}\ i\ (\mathit{Var}_2\ i') & \mid \mathit{name}\ i \equiv \mathit{name}\ i' & \\
 & \quad \wedge\ \mathit{depth}\ i \leqslant \mathit{depth}\ i' & = \mathit{Var}_2\ (\mathit{inc}\ i') \\
 & \mid \mathit{otherwise} & = \mathit{Var}_2\ i' \\
\multicolumn{3}{l}{\mathit{shift}\ i\ (\mathit{App}_2\ e_1\ e_2) = \mathit{App}_2\ (\mathit{shift}\ i\ e_1)\ (\mathit{shift}\ i\ e_2)} \\
\multicolumn{3}{l}{\mathit{shift}\ \_\ (\mathit{Num}_2\ z) \quad = \mathit{Num}_2\ z}
\end{array}
$$

The *shift* function binds an index as its first argument. The name of this index (e.g., x) denotes the name to be shifted. The depth of the index denotes the *cut-off* for the shift; i.e., how many #'s an x must at least be prefixed by before it is a free variable reference to x. For example, say we wish to shift all free references to x in the term $\lambda x.\, x\ (\#x)$. We should only shift $\#x$, not x, since x references the locally λ bound x. For this reason, the shift function uses a cut-off which is incremented when we move under binders by the same name as we are trying to shift. For example:

$$
\begin{array}{ll}
 & \mathit{shift}\ x\ (\lambda x.\, x\ (\#x)) \\
\equiv & \lambda x.\, (\mathit{shift}\ (\#x)\ x)\ (\mathit{shift}\ (\#x)\ (\#x)) \\
\equiv & \lambda x.\, x\ (\#\#x)
\end{array}
$$

The Berkling-Fehr substitution function subst_2 applies shifting to avoid variable capture when propagating substitutions under λ binders:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{subst}_2 :: \mathit{Index} \to \mathit{Expr}_2 \to \mathit{Expr}_2 \to \mathit{Expr}_2} \\
\mathit{subst}_2\ i\ s\ (\mathit{Lam}_2\ x\ e) & \mid \mathit{name}\ i \equiv x & = \mathit{Lam}_2\ x\ (\mathit{subst}_2\ (\mathit{inc}\ i)\ (\mathit{shift}\ (I\ 0\ x)\ s)\ e) \\
 & \mid \mathit{otherwise} & = \mathit{Lam}_2\ x\ (\mathit{subst}_2\ i\ (\mathit{shift}\ (I\ 0\ x)\ s)\ e) \\
\mathit{subst}_2\ i\ s\ (\mathit{Var}_2\ i') & \mid i \equiv i' & = s \\
 & \mid \mathit{otherwise} & = \mathit{Var}_2\ i' \\
\multicolumn{3}{l}{\mathit{subst}_2\ i\ s\ (\mathit{App}_2\ e_1\ e_2) = \mathit{App}_2\ (\mathit{subst}_2\ i\ s\ e_1)\ (\mathit{subst}_2\ i\ s\ e_2)} \\
\multicolumn{3}{l}{\mathit{subst}_2\ \_\ \_\ (\mathit{Num}_2\ z) \quad = \mathit{Num}_2\ z}
\end{array}
$$

To interpret an Expr_2 application $e_1\ e_2$, we first interpret e_1 to a function $\lambda x.\, e$, and then substitute x in the body e, such that occurrences of x at a higher depth are left untouched. But after we have substituted the bound occurrences of x in e, the depth of the remaining occurrences of x in e need to be decremented. To this end, we use an *unshift* function which decrements the depth of a given name, modulo a cut-off which now tells us what depth a name has to strictly be larger than in order for it to be a free variable to be unshifted:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{unshift} :: \mathit{Index} \to \mathit{Expr}_2 \to \mathit{Expr}_2} \\
\mathit{unshift}\ i\ (\mathit{Lam}_2\ x\ e) & \mid \mathit{name}\ i \equiv x & = \mathit{Lam}_2\ x\ (\mathit{unshift}\ (\mathit{inc}\ i)\ e)
\end{array}
$$

$$
\begin{array}{lll}
 & \mid \textit{otherwise} & = \textit{Lam}_2\ x\ (\textit{unshift}\ i\ e) \\
\textit{unshift}\ i\ (\textit{Var}_2\ i') & \mid \textit{name}\ i \equiv \textit{name}\ i' & \\
 & \quad \wedge\ \textit{depth}\ i < \textit{depth}\ i' & = \textit{Var}_2\ (\textit{dec}\ i') \\
 & \mid \textit{otherwise} & = \textit{Var}_2\ i' \\
\textit{unshift}\ i\ (\textit{App}_2\ \textit{t1}\ \textit{t2}) & = \textit{App}_2\ (\textit{unshift}\ i\ \textit{t1})\ (\textit{unshift}\ i\ \textit{t2}) & \\
\textit{unshift}\ \_\ (\textit{Num}_2\ z) & = \textit{Num}_2\ z &
\end{array}
$$

Using *unshift*, we can now implement an interpreter that does evaluation under λs and that uses capture-avoiding substitution:

$$
\begin{array}{l}
\textit{normalize}_2 :: \textit{Expr}_2 \rightarrow \textit{Expr}_2 \\
\textit{normalize}_2\ (\textit{Lam}_2\ x\ e)\quad = \textit{Lam}_2\ x\ (\textit{normalize}_2\ e) \\
\textit{normalize}_2\ (\textit{Var}_2\ i)\qquad = \textit{Var}_2\ i \\
\textit{normalize}_2\ (\textit{App}_2\ e_1\ e_2) = \textbf{case}\ \textit{normalize}_2\ e_1\ \textbf{of} \\
\quad \textit{Lam}_2\ x\ e \rightarrow \textit{unshift}\ (I\ 0\ x)\ (\textit{normalize}_2\ (\textit{subst}_2\ (I\ 0\ x)\ (\textit{normalize}_2\ e_2)\ e)) \\
\quad e_1' \qquad\quad \rightarrow \textit{App}_2\ e_1'\ (\textit{normalize}_2\ e_2) \\
\textit{normalize}_2\ (\textit{Num}_2\ z)\quad = \textit{Num}_2\ z
\end{array}
$$

For example, the problematic program from § 2.4 now yields a result with a free variable, as expected:

$$\textit{normalize}_2\ ((\lambda x.\, \lambda y.\, x)\ y)\ \equiv\ \lambda y.\, \#y$$

3.2 Relation to Renamingless Substitution

On the surface, the techniques involved in Berkling-Fehr substitution and our renamingless substitution strategy from § 2 seem rather different. A common point between the two is that they avoid renaming by strategically closing off certain variables to protect them from substitutions from lexically closer binders, and strategically reopening those variables to substitutions coming from lexically distant binders.

The renamingless substitution strategy achieves this by using a syntactic and rather coarse-grained discipline which closes entire sub-branches over all possible substitutions. When the interpreter reaches a closed sub-expression, it is re-opened. As discussed, this discipline works well for languages that do not perform evaluation under binders. While we demonstrated the technique using a call-by-value language in § 2, the technique is equally applicable to call-by-name interpretation. But not for languages that perform evaluation under binders.

Berkling-Fehr substitution uses a more fine-grained approach to strategically close off variables to protect them from substitutions from lexically closer binders, by shifting free occurrences of variables when moving under a binder. When a binder is eliminated, terms are unshifted. This fine-grained approach is not subject to the same limitations as the renamingless approach from § 2.3. Indeed, in their paper, Berkling and Fehr [7] prove that their notion of substitution and their modified λ-calculus is consistent with Church's λ calculus. Since shifting and unshifting requires more recursion over terms than the simpler renamingless approach from § 2, Berkling-Fehr substitution is less efficient. However, it is still more efficient than the renaming approach discussed in § 2.2.

As discussed, Berkling-Fehr substitution is closely related to De Bruijn indices, the main difference being that Berkling-Fehr use names and are more readable. To work around the readability issue with De Bruijn indices, one might also combine a named and De Bruijn approach where variable nodes comprise *both* a name *and* a De Bruijn index. But that leaves the question of how to disambiguate programs with ambiguous name. For example, how do we represent the Berkling-Fehr indexed expression $\lambda x.\,\lambda x.\,\#x$ using this hypothetical combined De Bruijn/named approach? Berkling-Fehr indices seem to strike an attractive balance between being practical and readable.

4 Related Work

In this paper we explored two techniques for capture avoiding substitution that avoids renaming, for the purpose of implementing static name binding in languages with λs. The topic of evaluating λ expressions has a long and rich history. Summarizing it all is beyond the scope of this paper; for overviews see, e.g., the works of Barendregt [6] or Cardone and Hindley [8]. We discuss a few of the papers that are most closely related to the techniques we have described.

In their formalization of λ calculus and type theory, McKinna and Pollack [17] consider a system that uses named substitution without renaming, for a particular notion of open terms. They consider a syntax that distinguishes two classes of names: *parameters* and *variables*. *Variable substitution* does not affect parameters, and *parameter substitution* does not affect variables. Their notion of variable substitution is defined for terms that are *variable-closed*, but which may be *parameter-open*. Thus, by encoding free variables as parameters, their system can be used to compute with open terms. However, syntactically distinguishing free variables this way seems to presupposes a static binding analysis. The approach we discussed in § 2.3 does not presuppose such static analysis.

Our paper considers how to interpret open terms. There exist several calculi in the literature for evaluating open terms. Accatolli and Guerrieri [2] gives an overview of several of these calculi for *open call-by-value*, which is the class of languages that the interpreters in § 2 and§ 3 interpret. In their paper, Accatolli and Guerrieri focus on the meta-theory of these calculi. For their meta-theoretical study they rely on an unspecified notion of capture-avoiding substitution. In this paper, we explore how to define such capture-avoiding substitution in a way that does not perform renaming. While the meta-theory of Berkling-Fehr substitution has been studied [7], the meta-theory of the substitution strategy in § 2.3 remains an open question.

5 Conclusion

We have discussed two techniques for implementing capture avoiding substitution in definitional interpreters in a way that does not require renaming of bound variables. One of the techniques relies on a coarse-grained but simple discipline for closing terms, is known to not support interpretation strategies that evaluate under binders, and has (to the best of our knowledge) not been studied meta-theoretically. The other technique is an existing technique from the literature. While this technique is less efficient, it is more fine-grained and so does is not subject to the same limitations as the first technique. It also has a well-established meta-theory. Neither of the two techniques seem to be widely known or at least not widely applied. With this work, we hope to increase awareness of these techniques.

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 Beniamino Accattoli and Giulio Guerrieri. Open call-by-value. In Atsushi Igarashi, editor, *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, volume 10017 of *Lecture Notes in Computer Science*, pages 206–226, 2016. `doi:10.1007/978-3-319-47958-3\_12`.

3 Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 666–679. ACM, 2017. `doi:http://dl.acm.org/citation.cfm?id=3009866`.

4 Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for imperative languages. *Proc. ACM Program. Lang.*, 2(POPL):16:1–16:34, 2018. `doi:10.1145/3158104`.

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 Henk Barendregt. The impact of the lambda calculus in logic and computer science. *Bull. Symb. Log.*, 3(2):181–215, 1997. `doi:10.2307/421013`.

7 K. J. Berkling and Fehr E. A modification of the λ-calculus as a base for functional programming languages. In *ICALP 1982*. Springer Berlin Heidelberg, 1982.

8 Felice Cardone and J. Roger Hindley. *Logic from Russell to Church*, volume 5 of *Handbook of the History of Logic*, chapter History of Lambda-calculus and Combinatory Logic. Elsevier, 2009.

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

10 Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932. URL: `http://www.jstor.org/stable/1968337`.

11 Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936. URL: `http://dx.doi.org/10.2307/2371045`, `doi:10.2307/2371045`.

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

13 Nils Anders Danielsson. Operational semantics using the partiality monad. In Peter Thiemann and Robby Bruce Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 127–138. ACM, 2012. `doi:http://doi.acm.org/10.1145/2364527.2364546`.

14 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`.

15 Shriram Krishnamurthi. Programming languages: Application and interpretation. `https://www.plai.org/3/2/PLAI%20Version%203.2.0%20electronic.pdf`, 2002. Accessed: 2022-12-01.

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

17 James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *J. Autom. Reason.*, 23(3-4):373–409, 1999. `doi:10.1023/A:1006294005493`.

18 Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. In *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 589–615, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. `doi:10.1007/978-3-662-49498-1_23`.

19 Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

20 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`.

21 John C. Reynolds. Definitional interpreters for higher-order programming languages. *High. Order Symb. Comput.*, 11(4):363–397, 1998. doi:10.1023/A:1010027404223.

22 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.

23 Jeremy Siek. *Essentials of Compilation*. MIT Press, 2022.