Binders Unbound

Stephanie Weirich Brent Yorgey
University of Pennsylvania
{sweirich,byorgey}@cis.upenn.edu

Tim Sheard
Portland State University
sheard@cs.pdx.edu

Abstract

Implementors of compilers, program refactorers, theorem provers, proof checkers, and other systems that manipulate syntax know that dealing with name binding is difficult to do well. Operations such as α-equivalence and capture-avoiding substitution seem simple, yet subtle bugs often go undetected. Furthermore, their implementations are tedious, requiring "boilerplate" code that must be updated whenever the object language definition changes.

Many researchers have therefore sought to specify binding syntax declaratively, so that tools can correctly handle the details behind the scenes. This idea has been the inspiration for many new systems (such as Beluga, Delphin, FreshML, FreshOCaml, Cαml, FreshLib, and Ott) but there is still room for improvement in expressivity, simplicity and convenience.

In this paper, we present a new domain-specific language, UNBOUND, for specifying binding structure. Our language is particularly expressive—it supports multiple atom types, pattern binders, type annotations, recursive binders, and nested binding (necessary for telescopes, a feature found in dependently-typed languages). However, our specification language is also simple, consisting of just five basic combinators. We provide a formal semantics for this language derived from a locally nameless representation and prove that it satisfies a number of desirable properties.

We also present an implementation of our binding specification language as a GHC Haskell library implementing an embedded domain specific language (EDSL). By using Haskell type constructors to represent binding combinators, we implement the EDSL succinctly using datatype-generic programming. Our implementation supports a number of features necessary for practical programming, including flexibility in the treatment of user-defined types, best-effort name preservation (for error messages), and integration with Haskell's monad transformer library.

Categories and Subject Descriptors D.2.3 [*Coding Tools and Techniques*]; D.1.1 [*Applicative (Functional) Programming*]; E.1 [*Data Structures*]

General Terms Algorithms, Languages.

Keywords generic programming, Haskell, name binding, patterns

1. Introduction

Name binding is one of the most annoying parts of language implementations. Although functional programming languages such as Haskell and ML excel at the implementation of type checkers, compilers, and interpreters, there is an impedance mismatch between the free structure provided by algebraic datatypes and the syntax identified up to α-equivalence that we actually want to model. Although there are many techniques for implementing name binding, they require subtle invariants that pervade the system. Implementation flaws cause bugs that can be quite difficult to track down. Furthermore, the implementations themselves are tedious, requiring boilerplate code that must be maintained as the implemented language evolves. While such boilerplate is straightforward, it causes friction for developers who just want to get the job done.

And all this work is for something so "obvious" that it is often elided from language definitions!

There has been much research towards solving this problem. Recently introduced languages and tools provide primitive support for variable-binding, based on first-order ([6, 19, 28, 29]) and higher-order representations ([16, 18]). These tools handle the details behind the scenes, relieving programmers of the tedium and subtle bugs described above. However, these tools must also satisfy the practical needs of programmers, and here they fall short:

Expressiveness: These tools provide *binding specification languages* that specify what variables are bound where. As unary lexical scoping (binding a single variable in a single location) is not sufficient for many applications, many of these tools support a language for patterns in binding specifications. Despite this flexibility, it is still not enough—there are patterns that we would like to use that cannot be defined by existing specification languages.

Availability: Programmers want to write code in their language, and they want to do it directly. Tools that are wrappers for existing languages are preferred to completely new systems, but such tools still require update with each new version of the language. Libraries are more stable and have the added benefit of simple distribution, some degree of portability and familiar syntax.

Choice of implementations: These tools each provide only *one* implementation for any given binding specification. However, name binding involves a number of different operations and some implementations may favor one over the other. Programmers should be able to swap out implementations (or write their own) if they find one that works better with their application.

In this paper, we present a new domain-specific language, UNBOUND, for specifying binding structure that addresses these issues. Concretely, our contributions are as follows:

- We describe a small, compositional set of abstract combinators which form the entire basis for UNBOUND. This interface succinctly characterizes our specification language.
- We show, via examples (§ 2), that UNBOUND is nonetheless expressive. In particular, it supports multiple atom types, pattern

binders, type annotations, recursive binders, and nested binders. The last are necessary to model *telescopes* and are not supported by any existing specification language.

- We give a formal semantics for our specification language (§ 4) based on a locally nameless representation and prove its correctness (§ 5). Our choice of representation leads to a simple semantics and straightforward metatheory. Alternative meanings are also possible; the simplicity of ours makes it a good reference for more sophisticated implementations.
- We have implemented our framework as a Haskell library (§ 6), using Haskell's generic programming support to automatically derive standard operations. Our library is available for download from Hackage,[1] along with extensive documentation and examples. (Note GHC 7 is required.)

2. The UNBOUND Specification Language

We begin with a simple UNBOUND specification.[2] Functional programmers are accustomed to using algebraic datatypes to specify the abstract syntax of a programming language. UNBOUND introduces type combinators that encode binding structure into the algebraic datatype itself. For example, to represent the untyped lambda calculus, we use the E datatype below:

```
type N = Name E
data E = Var N
       | Lam (Bind N E)
       | App E E
```

The new abstract type Name represents variables, and is indexed by the type of values which can be substituted for them (here, E). For convenience, we define N as a synonym for Name E. Lambda abstractions are represented using the type Bind N E, indicating a name paired with an expression in which the name is bound. Application does not involve binding, so it is simply a pair of E values as expected.

UNBOUND uses this datatype definition to derive standard operations for working with syntax, such as α-equivalence, free variable calculation, and capture-avoiding substitution. For example, suppose we want to implement parallel reduction for untyped lambda calculus terms. This operation looks throughout a term for β- and η-reductions, even under lambda abstractions, transforming it into a simpler form. An implementation is shown in Figure 1. The signatures for the UNBOUND-derived operations that this code relies on are at the top of the figure. All of these functions are automatically derived by UNBOUND.

The function *red* has three cases. The *Var* case is trivial. The *Lam* case must handle the possibility of η-reduction, so we must break the lambda into its two constituent parts—its bound variable, and its body. Note that the type Bind N E type is abstract, so we cannot use pattern matching to extract its components. Instead, the monadic *unbind* operation decomposes the binding, ensuring that the name x does not conflict with other names currently in scope.

Once the body of the lambda expression has been reduced, the code checks to see if it can do an η-reduction. This is possible when the body is exactly the application of some other term e'' to the variable x, where x does not appear free in e''. If an η-reduction is not possible, the binding is reformed using the *bind* constructor for the Bind N E type. A similar unbinding occurs in the application case, when a β-reduction has been detected, followed by an invocation of a capture-avoiding substitution operation also provided by UNBOUND.

[1] `http://hackage.haskell.org/package/unbound/`

[2] While our examples are presented in Haskell, using our Haskell library, the examples themselves are language neutral.

UNBOUND operations used in this example:

```
bind   :: N → E → (Bind N E)
unbind :: Fresh m ⇒ (Bind N E) → m (N, E)
fv     :: E → Set N
subst  :: N → E → E
```

Parallel Reduction:

```
red :: Fresh m ⇒ E → m E
red (Var x) = return (Var x)
red (Lam b) = do
  (x, e) ← unbind b
  e'     ← red e
  case e' of    -- apply the eta-rule: (λ x.e x) = e
    App e'' (Var y) | x ≡ y ∧ ¬ (x ∈ fv e'') → return e''
    _ → return (Lam (bind x e'))
red (App e1 e2) = do
  e1' ← red e1
  case e1' of   -- apply the beta rule: (λ x.e) t = e[t/x]
    Lam b → do
      (x, e') ← unbind b
      e2'     ← red e2
      return (subst x e2' e')
    _ → do
      e2' ← red e2
      return (App e1' e2')
```

Figure 1. Parallel reduction for E

$T \in \mathbb{T}$	
Name T	Names for Ts
R	Regular datatype containing only terms
Bind P T	Bind pattern P in body T
$P \in \mathbb{P}$	
Name T	Single binding name
$\mathsf{R_P}$	Regular datatype containing only patterns
Embed T	Embedded term (§ 3.1)
Rebind P P	Nested binding pattern (§ 3.2)
Rec P	Recursive binding pattern (§ 3.3)

Figure 2. UNBOUND type combinators

At this point, one may ask: *Where do these operations come from*? *What do they mean*? *How do we know that the code given for red correctly implements parallel reduction for the lambda calculus*? The first question we answer in § 6 where we discuss the Haskell implementation of UNBOUND. The other questions motivate our semantics (§ 4) and the theorems we choose to prove about it (§ 5).

3. Beyond single binding

A key feature of UNBOUND is that programmers are not limited to binding a single variable at a time. Instead, the *bind* constructor takes a pattern of variables, and abstracts over all of them. A large class of types may be used as patterns. As a simple example, lists of names are patterns. If we wanted to allow the syntax $\lambda x\ y\ z \rightarrow \ldots$ as a convenient shorthand for $\lambda x \rightarrow \lambda y \rightarrow \lambda z \rightarrow \ldots$, where x, y,

and z are distinct names,[3] we could change our definition of E to the following:

$$\begin{aligned}\mathbf{data}\ E = &\ \mathit{Var}\ N\\ \mid &\ \mathit{Lam}\ (\mathsf{Bind}\ [N]\ E)\\ \mid &\ \mathit{App}\ E\ E\end{aligned}$$

In general, UNBOUND uses two sorts of types: those that may be used as *patterns*, where names are binding occurrences, and those that are *terms*, where names are references to binding sites. Figure 2 summarizes these two classes, written $\mathbb{P}$ and $\mathbb{T}$ respectively. The Bind type combinator takes a pattern type as its first argument and a term type as its second argument and returns a term type. Other term types include Name (representing free variables) and *regular* datatypes—those built using unit, base types, sums, products, and least fixpoint—that contain only term types. By convention, we use the metavariable P for pattern types and T for term types.

The expressiveness of UNBOUND is determined by $\mathbb{P}$, the collection of types that can be used as patterns. These types include Names, of course, as well as regular datatypes that contain only other pattern types. This mean that some types, such as Int and String, can be used as both terms and patterns. We describe the three remaining UNBOUND pattern combinators (Embed, Rebind and Rec) in more detail in the next subsections.

As a more sophisticated example of pattern binding, consider adding pattern matching to the E language with a case statement. Each branch is encoded as Bind $\mathit{Pat}\ E$, where Pat is a new datatype representing object-language patterns. Every name occurring in a Pat will be bound in the respective body.

$$\begin{aligned}&\mathbf{data}\ \mathit{Pat} = \mathit{PVar}\ N \mid \mathit{PCon}\ \mathit{String}\ [\mathit{Pat}]\\ &\mathbf{data}\ E = \ldots\\ &\qquad \mid \mathit{Con}\ \mathit{String}\ [E] \qquad \text{-- data constructors}\\ &\qquad \mid \mathit{Case}\ E\ [\mathsf{Bind}\ \mathit{Pat}\ E] \quad \text{-- pattern matching}\end{aligned}$$

It is not hard to check that Pat is a valid pattern type (since it contains only Names and Strings), justifying its use as the first argument to Bind.

3.1 Embedding terms in patterns

In many situations it is convenient to be able to embed terms within patterns. Such embedded terms do not bind variables along with the rest of the pattern. For example, suppose we wanted to extend our E language with simple let-binding, let $x{=}e_1$ in e_2. Here x is bound in e_2 but not in e_1.

A semantically correct encoding puts the e_1 in the abstract syntax before binding x in e_2, "lifting" e_1 outside the binding so it does not participate.

$$\begin{aligned}&\mathbf{type}\ E_1 = E\\ &\mathbf{type}\ E_2 = E\\ &\mathbf{data}\ E = \ldots\\ &\qquad \mid \mathit{Let}\ E_1\ (\mathsf{Bind}\ N\ E_2)\end{aligned}$$

(We use the type synonyms E_1 and E_2 to indicate which sub-terms of Let correspond to e_1 and e_2.) However, this encoding forces us to write the terms in an unnatural order; moreover, it fails to express the relationship between the name being let-bound and its definition. We can craft a more satisfying solution using the embedding combinator Embed provided by UNBOUND:

$$\begin{aligned}&\mathbf{data}\ E = \ldots\\ &\qquad \mid \mathit{Let}\ (\mathsf{Bind}\ (N, \mathsf{Embed}\ E_1)\ E_2)\end{aligned}$$

Embed may only occur within pattern types, where it serves as an "escape hatch" for embedding terms which do not bind any names. Note that a term type within an Embed may itself contain pattern types (inside the left-hand side of Bind) which may contain Embedded term types, and so on.

This formulation with Embed also enables us to extend our let expressions to multiple bindings, by using a pattern list:

$$\begin{aligned}&\mathbf{data}\ E = \ldots\\ &\qquad \mid \mathit{Let}\ (\mathsf{Bind}\ [(N, \mathsf{Embed}\ E)]\ E)\end{aligned}$$

Without Embed, we would have to encode this binding specification by "unzipping" the list of name-definition pairs and lifting the definitions outside of the Bind:

$$\begin{aligned}&\mathbf{data}\ E = \ldots\\ &\qquad \mid \mathit{Let}\ [E]\ (\mathsf{Bind}\ [N]\ E)\end{aligned}$$

But this example is even worse than the corresponding encoding of let with a single binding. Not only does it force us to use an unnatural order and fail to encode the relationship between names and their respective definitions, it also admits "junk" terms where the lists are different lengths. Embed makes possible an encoding where names and their definitions are paired, as they should be.

3.2 Nested binding

Consider now a let* construct, let* $x_1{=}e_1, \ldots, x_n{=}e_n$ in e, where each x_i is bound in e and *also* in all the e_j with $j > i$. One way to encode this pattern is by iterating the encoding for single let bindings discussed above:

$$\begin{aligned}&\mathbf{data}\ \mathit{LetList} = \mathit{Body}\quad E\\ &\qquad \mid \mathit{Binding}\ (\mathsf{Bind}\ (N, \mathsf{Embed}\ E)\ \mathit{LetList})\\ &\mathbf{data}\ E \quad = \ldots\\ &\qquad \mid \mathit{LetStar}\ \mathit{LetList}\end{aligned}$$

This succeeds in capturing the binding structure of let*, and may be sufficient for some purposes. However, it does have one major drawback: in order to extract the body of the let* expression, we must first recurse through all the bindings. It is more convenient to encode a let* expression by *pairing* a list of bindings and a body, so the body can be accessed without first processing the bindings. As a first try, we could write

$$\begin{aligned}&\mathbf{data}\ E = \ldots\\ &\qquad \mid \mathit{LetStar}\ (\mathsf{Bind}\ [(N, \mathsf{Embed}\ E)]\ E)\end{aligned}$$

but this is just the multiple binding example from the previous subsection. With this specification, the x_i are bound *only* in the body of the let*, not in the definitions of subsequent variables. We evidently want a way to *nest* additional binding structure within the pattern of the outermost Bind.

UNBOUND provides a novel *rebinding* pattern combinator for precisely this purpose. Rebind $P_1\ P_2$ acts like the pattern type (P_1, P_2), except that P_1 also scopes over P_2, so the binders in P_1 may be referred to by terms embedded within P_2. (The fact that P_1 scopes over P_2 in this way has no effect on the *pattern* portion of P_2.) For example, consider the specification

$$\begin{aligned}&\mathbf{type}\ N_1 = \mathsf{Name}\ E\\ &\mathbf{type}\ N_2 = \mathsf{Name}\ E\\ &\mathsf{Bind}\ (\mathsf{Rebind}\ N_1\ (N_2, \mathsf{Embed}\ E_1))\ E_2\end{aligned}$$

Here N_1 and N_2 are bound in E_2, and additionally N_1 is also bound within E_1.

Using rebinding, we can faithfully encode the binding structure of let* as follows:

$$\begin{aligned}&\mathbf{data}\ \mathit{Lets} = \mathit{Nil}\\ &\qquad \mid \mathit{Cons}\ (\mathsf{Rebind}\ (N, \mathsf{Embed}\ E)\ \mathit{Lets})\\ &\mathbf{data}\ E \quad = \ldots\\ &\qquad \mid \mathit{LetStar}\ (\mathsf{Bind}\ \mathit{Lets}\ E)\end{aligned}$$

[3] Although UNBOUND supports shadowing, a single pattern must be linear (i.e. not contain repeated variables).

All the names within the sequence of definitions are bound in the body of the let*-expression (Bind $Lets$ E); additionally, each name (paired with its definition as an embedded term) is bound within any $Embeds$ occurring in the remainder of the sequence—that is, within the definitions of subsequent names.

Telescopes A particularly important example of a binding pattern that requires Rebind is a *telescope*. Telescopes were invented by de Bruijn [8] to model dependently-type systems. They are used frequently in specification of dependently-typed languages, including Epigram [13] and Agda [15].

A telescope, Δ, is a sequence of variables with their types:

$$x_1 : A_1, \ldots, x_n : A_n.$$

However, each variable scopes over the types that occur later in the telescope. For example, here x_1 may occur in A_2, A_3, and so on. The name *telescope* comes from the optical device, which is built as as a sequence of segments that slide into one another.

Telescopes are used for "aggregate binding". For example, consider the following (very simple) fragment of a dependently-typed language. In this language, functions can take multiple parameters and be applied to multiple arguments. However, because of dependent types, the type of each parameter is allowed to mention earlier parameters.

$$A, B, M, N \ ::= x \mid \Pi\Delta.B \mid \lambda\Delta.M \mid M\ (N_1 \ldots N_n)$$

Telescopes gather together all of the parameters of the function in both its definition ($\lambda\Delta.M$) and its type ($\Pi\Delta.B$). Because telescopes are essentially typing contexts, the typing rule for abstractions merely appends the telescope to the current typing context:

$$\frac{\Gamma, \Delta \vdash M : B}{\Gamma \vdash \lambda\Delta.M : \Pi\,\Delta.B}$$

Type checking the multi-applications requires an auxiliary judgment that determines if the vector of arguments "fits into" the telescope.

$$\frac{}{\Gamma \vdash : .} \qquad \frac{\Gamma \vdash N_1 : A \quad \Gamma \vdash N_2\,..\,N_n : \Delta[x \mapsto N_1]}{\Gamma \vdash N_1\,N_2\,..\,N_n : (x : A, \Delta)}$$

This judgment verifies that all of the arguments have the right types. Computing the result type of the application requires substituting all of the arguments for each binding variable in the the telescope.

$$\frac{\begin{array}{l}\Gamma \vdash M : \Pi\,\Delta.A \quad \Gamma \vdash N_1\,..\,N_n : \Delta \\ dom\,(\Delta) = x_1\,..\,x_n\end{array}}{\Gamma \vdash M(N_1\,..\,N_n) : B[x_1\,..\,x_n \mapsto N_1\,..\,N_n]}$$

This fragment demonstrates the important features of telescopes. Sometimes they are used as binding patterns and sometimes they are used as the types of vectors, independent of binding. Implementing this language using traditional unary binding would be annoying because one would have to traverse the entire telescope to see the body of the function or the body of the dependent type. That is not so much of an issue in this simple example, but the semantics of features like inductive families (with eliminators based on induction principles or dependent pattern matching) is greatly simplified by the aggregate binding that telescopes provide.

In UNBOUND, we can represent the language fragment above using Rebind:

$$\begin{array}{lll}\textbf{data}\ E & = & Var\ N \\ & \mid & Pi\ (\mathsf{Bind}\ Tele\ E) \\ & \mid & Lam\ (\mathsf{Bind}\ Tele\ E) \\ & \mid & App\ E\ [E] \\ \textbf{data}\ Tele & = & Empty \mid \mathsf{Rebind}\ (N, \mathsf{Embed}\ E)\ Tele\end{array}$$

Furthermore, UNBOUND automatically provides all of the machinery necessary for working with telescopes, including calculation of their binding variables, multiple substitution in terms, and substitution through the telescopes.

3.3 Recursive binding

Our E language is looking nice, but what if we want to add some recursion? We can try to encode a letrec construct,

$$\mathsf{letrec}\ x_1{=}e_1, \ldots, x_n{=}e_n\ \mathsf{in}\ e,$$

where this time, the x_i are bound in e as well as *all* the e_i. This is straightforward if we are willing to lift all the x_i out to the front:

$$\begin{array}{lll}E & = & \ldots \\ & \mid & Letrec\ (\mathsf{Bind}\ [N]\ ([E], E))\end{array}$$

However, the problems with this sort of encoding have already been discussed. We would like to encode letrec in such a way that names and definitions are paired.

Rebind doesn't help, because it forces us to separate binders from the terms over which they scope, just like Bind. We need a way to freely mix patterns and terms bound by the patterns in the same data structure. UNBOUND provides the Rec combinator for this purpose. In Rec P, names in the pattern P scope recursively over any terms embedded in P itself. However, Rec P itself is also a pattern, so names in P also scope externally over any term that binds Rec P. Intuitively, Rec is just a "recursive" version of Rebind.

An appropriate encoding of letrec is therefore:

$$\begin{array}{lll}E & = & \ldots \\ & \mid & Letrec\ (\mathsf{Bind}\ (\mathsf{Rec}\ [(N, \mathsf{Embed}\ E)])\ E)\end{array}$$

Here the pattern $[(N, \mathsf{Embed}\ E)]$ scopes over itself—hence all the names are bound in all the definitions—as well as over the body of the letrec.

4. Semantics

In the previous section, we gave a number of examples of specifying different sorts of binding patterns found in programming languages. However, we have been fairly vague about what those specifications actually mean. In this section, we fill in the details by giving it a semantics.

Our semantics comes in two parts. We first define the *representation* of syntax with binders, and specify smart constructors and destructors that work with this representation. Second, we define the *action* of UNBOUND operations (α-equivalence, free variable calculation and substitution) in terms of this representation. Once we have formally defined this semantics, we prove that it satisfies the properties we expect (§ 5) and discuss the correspondence between it and our concrete Haskell implementation (§ 6).

4.1 Representation

For simplicity, we use a *locally nameless* representation of terms with binding structure. This representation provides a straightforward semantics for UNBOUND, one that is both simple to implement and simple to prove things about. Using a locally nameless representation is an old idea—we give more details about its history in our discussion of § 8.

A locally nameless representation of terms with binding structure separates bound variables, represented by de Bruijn indices, from free variables, represented by *atoms*. (Atoms are often taken to be strings, but any countably infinite set with decidable equality will do.) This representation has the advantage that α-equivalence is simply structural equality. Distinguishing bound variables from free variables in this way also means that we do not need to keep track of the current *scope* of a free variable and shift it as we move from one scope to another, as we would with a purely de Bruijn-indexed representation.

$$
\begin{aligned}
\mathbb{A} &::= \{\mathsf{x}, \mathsf{y}, \mathsf{z}, \dots\} \\
b &::= j \mathbin{@} k \\
t &::= \mathsf{x} \mid b \mid \mathsf{K}\ t_1 \dots t_n \mid \mathsf{Bind}\ p\ t \\
p &::= -_{\mathsf{x}} \mid \mathsf{K}\ p_1 \dots p_n \mid \mathsf{Rebind}\ p\ p \mid \mathsf{Embed}\ t \mid \mathsf{Rec}\ p
\end{aligned}
$$

Figure 3. Syntax of atoms, indices, terms, and patterns

The locally nameless syntax we use to represent terms with binding structure is shown in Figure 3. As in Figure 2, we separate terms from patterns. Terms t have term types T, whereas patterns have pattern types P.

Names that appear in terms can either be free names, x, or bound names, b. Free names are drawn from the set $\mathbb{A}$ of atoms. (In the interest of simplicity, the semantics we describe here only includes a single sort for atoms; extending it to multiple atom sorts is straightforward.) Terms also include applications of constructor constants K to zero or more subterms. Note that constructor application covers all terms with some regular type R; in the semantics they are all handled in precisely the same way. Indeed, thanks to generic programming, this is actually a faithful reflection of our Haskell implementation, which handles all data constructors other than the special UNBOUND combinators uniformly.

Like terms, patterns can be formed by the application of constructors to subpatterns. Names inside patterns are *binders*, written $-_{\mathsf{x}}$, which represent binding occurrences of names. We denote binders with special syntax, $-_{\mathsf{x}}$, to emphasize that we should think of them as placeholders with an associated name.

The astute reader may note that we are punning a bit with our syntax: in earlier examples, Bind and friends showed up as *types*, whereas here they are playing the role of *data*. The resolution of the apparent inconsistency is that Bind, Embed, Rebind, and Rec are all *singleton types* with eponymous constructors.

For example, we can define an operation that lists all of the atoms that a pattern will bind as shown below.

$$
\begin{aligned}
&\mathit{binders} :: P \to [\mathbb{A}] \\
&\mathit{binders}\ -_{\mathsf{x}} &&= [x] \\
&\mathit{binders}\ (\mathsf{K}\ p_1 \dots p_n) &&= \mathit{binders}\ p_1 +\!\!+ \dots +\!\!+ \mathit{binders}\ p_n \\
&\mathit{binders}\ (\mathsf{Rebind}\ p_1\ p_2) &&= \mathit{binders}\ p_1 +\!\!+ \mathit{binders}\ p_2 \\
&\mathit{binders}\ (\mathsf{Embed}\ t) &&= \varnothing \\
&\mathit{binders}\ (\mathsf{Rec}\ p) &&= \mathit{binders}\ p
\end{aligned}
$$

Note that even though we are using Haskell-like syntax, this definition is type-directed. It works for any pattern of any pattern type; the type of $\mathit{binders}$, $P \to [\mathbb{A}]$, is an abbreviation for $\forall\ P : \mathbb{P}, P \to [\mathbb{A}]$. In our Haskell implementation, to be discussed in more detail in § 6, each clause of a definition such as this one corresponds to a method definition in a type class instance.

4.2 Names, indices and patterns

Bound names b consist of two natural number indices, $j \mathbin{@} k$. The first index j references a pattern, counting outwards from zero; the second index k, is an offset. It references a particular binder within the given pattern, counting from left to right, also starting from zero. For example, in

$$\mathsf{Bind}\ (-_{\mathsf{x}}, -_{\mathsf{y}}, -_{\mathsf{z}})\ (\mathsf{Bind}\ \ -_{\mathsf{q}}\ \ 1@2)$$

the bound variable $1@2$ refers to $-_{\mathsf{z}}$, the index-2 binder within the index-1 enclosing pattern.

Therefore, an important part of this representation is the connection between patterns and offsets. We make this connection precise with the operations nth and find (although we omit their definitions in the interest of space):

$$
\begin{aligned}
&\mathit{nth} :: P \to \mathbb{N} \to \mathsf{Maybe}\ \mathbb{A} \\
&\mathit{find} :: P \to \mathbb{A} \to \mathsf{Maybe}\ \mathbb{N}
\end{aligned}
$$

nth takes a pattern and a natural number n and finds the nth name bound in that pattern, failing if there are not enough binders. find takes a pattern and a name and finds the first index of that name in the pattern, failing if the name does not occur.

4.3 Open and close

The two most important operations for the locally nameless representation are *close* and *open*. The former is used for binding terms: it converts atoms (*i.e.* free names) to indices (*i.e.* bound names). The latter does the reverse, replacing indices that resolve to a particular binding location by free names.

We call the first operation close, as we are closing the term with respect to the free names listed in a pattern. Likewise, we call the inverse operation open, as we may use it to open up a binder in order to recurse through its subcomponents. These two operations are standard components for working with locally nameless representation [2, 10]. Here we modify them to close and open terms with respect to a *pattern* instead of a single variable, and also to close and open the terms embedded inside patterns.

The close operation is defined in Figure 4. It takes as input a natural number *level*, a pattern, and a term, and returns a new term where free variables matching binders in the pattern have been replaced by bound variables at the given level. In the free variable case, it uses find to look for a matching binder and generate the appropriate index if one is found. We also define a version of close for patterns, $\mathit{close}_{\mathsf{P}}$, whose job is to recurse through patterns looking for Embedded terms to which close can be applied. When recursing under a binder (Bind, Rebind, or Rec), both close and $\mathit{close}_{\mathsf{P}}$ increment the current level.

The open operation is also defined in Figure 4. It takes a natural number level, a pattern, and a term, and "opens" the term by interpreting bound variables at the given level as references into the pattern, replacing them by the free variable attached to the referenced binder.[4] Similarly to close, open is mutually defined with a pattern version $\mathit{open}_{\mathsf{P}}$. In the case where a bound variable is found which matches the current level, open uses nth to index into the pattern and pick out the free variable associated with the referenced binder.

We use $\mathit{close}\ \ p\ \ t$ and $\mathit{open}\ \ p\ \ t$ as convenient synonyms for $\mathit{close}\ 0\ p\ t$ and $\mathit{open}\ 0\ p\ t$, respectively.

4.4 Constructing terms and patterns

Figure 5 lists the smart constructors and destructors that are part of the interface to the type combinators exported by UNBOUND. In the next two subsections, we discuss the implementations of these operations.

We use close to define the constructors for binding abstractions. Closing a term with respect to the pattern binds the pattern variables in the term.

$$\mathit{bind}\ p\ t = \mathsf{Bind}\ p\ (\mathit{close}\ p\ t)$$

[4] In the locally nameless literature, open is sometimes defined as *bound-variable substitution* and generalized to replace bound variables with terms instead of with free variables. Such definitions save effort as often the next step after opening is substituting for the new free variable. In that case, the definition of open is a little more complex than what is presented here in order to deal with substituting terms with dangling bound variable references. We prefer our reference semantics to be as simple as possible so we avoid such complications.

$$
\begin{array}{lll}
\mathit{close} :: \mathbb{N} \to P \to T \to T & & \\
\mathit{close}\ l\ p\ b & = b & \\
\mathit{close}\ l\ p\ \mathsf{x} & = \mathbf{case}\ \mathit{find}\ p\ \mathsf{x}\ \mathbf{of} & \\
 & \quad \mathsf{Just}\ i \ \to l\mathsf{@}i & \\
 & \quad \mathsf{Nothing} \to \mathsf{x} & \\
\mathit{close}\ l\ p\ (\mathsf{K}\ t_1 \ldots t_n) & = \mathsf{K}\ (\mathit{close}\ l\ p\ t_1) \ldots & \\
 & \quad (\mathit{close}\ l\ p\ t_n) & \\
\mathit{close}\ l\ p\ (\mathsf{Bind}\ p'\ t) & = \mathsf{Bind}\ (\mathit{close}_\mathsf{P}\ l\ p\ p') & \\
 & \quad (\mathit{close}\ (l+1)\ p\ t) & \\
\mathit{close}_\mathsf{P} :: \mathbb{N} \to P \to P \to P & & \\
\mathit{close}_\mathsf{P}\ l\ p\ -_\mathsf{x} & = -_\mathsf{x} & \\
\mathit{close}_\mathsf{P}\ l\ p\ (\mathsf{K}\ p_1 \ldots p_n) & = \mathsf{K}\ (\mathit{close}_\mathsf{P}\ l\ p\ p_1) \ldots & \\
 & \quad (\mathit{close}_\mathsf{P}\ l\ p\ p_n) & \\
\mathit{close}_\mathsf{P}\ l\ p\ (\mathsf{Rebind}\ p_1\ p_2) & = \mathsf{Rebind}\ (\mathit{close}_\mathsf{P}\ l\ p\ p_1) & \\
 & \quad (\mathit{close}_\mathsf{P}\ (l+1)\ p\ p_2) & \\
\mathit{close}_\mathsf{P}\ l\ p\ (\mathsf{Embed}\ t) & = \mathsf{Embed}\ (\mathit{close}\ l\ p\ t) & \\
\mathit{close}_\mathsf{P}\ l\ p\ (\mathsf{Rec}\ p') & = \mathsf{Rec}\ (\mathit{close}_\mathsf{P}\ (l+1)\ p\ p') &
\end{array}
$$

$$
\begin{array}{lll}
\mathit{open} :: \mathbb{N} \to P \to T \to T & & \\
\mathit{open}\ l\ p\ (j\mathsf{@}k) \mid j \equiv l & = \mathbf{case}\ \mathit{nth}\ p\ k\ \mathbf{of} & \\
 & \quad \mathsf{Just}\ \mathsf{x} \ \to \mathsf{x} & \\
 & \quad \mathsf{Nothing} \to j\mathsf{@}k & \\
\mathit{open}\ l\ p\ \mathsf{x} & = \mathsf{x} & \\
\mathit{open}\ l\ p\ (\mathsf{K}\ t_1 \ldots t_n) & = \mathsf{K}\ (\mathit{open}\ l\ p\ t_1) \ldots & \\
 & \quad (\mathit{open}\ l\ p\ t_n) & \\
\mathit{open}\ l\ p\ (\mathsf{Bind}\ p'\ t) & = \mathsf{Bind}\ (\mathit{open}_\mathsf{P}\ l\ p\ p') & \\
 & \quad (\mathit{open}\ (l+1)\ p\ t) & \\
\mathit{open}_\mathsf{P} :: \mathbb{N} \to P \to P \to P & & \\
\mathit{open}_\mathsf{P}\ l\ p\ (\mathsf{K}\ p_1 \ldots p_n) & = \mathsf{K}\ (\mathit{open}_\mathsf{P}\ l\ p\ p_1) \ldots & \\
 & \quad (\mathit{open}_\mathsf{P}\ l\ p\ p_n) & \\
\mathit{open}_\mathsf{P}\ l\ p\ -_\mathsf{x} & = -_\mathsf{x} & \\
\mathit{open}_\mathsf{P}\ l\ p\ (\mathsf{Rebind}\ p_1\ p_2) & = \mathsf{Rebind}\ (\mathit{open}_\mathsf{P}\ l\ p\ p_1) & \\
 & \quad (\mathit{open}_\mathsf{P}\ (l+1)\ p\ p_2) & \\
\mathit{open}_\mathsf{P}\ l\ p\ (\mathsf{Embed}\ t) & = \mathsf{Embed}\ (\mathit{open}\ l\ p\ t) & \\
\mathit{open}_\mathsf{P}\ l\ p\ (\mathsf{Rec}\ p') & = \mathsf{Rec}\ (\mathit{open}_\mathsf{P}\ (l+1)\ p\ p') &
\end{array}
$$

Figure 4. *close* and *open*

Effectively, this replaces all free occurrences of variables that appear in the pattern with indices. For example, binding a pair of variables in a term that references both variables will produce indices that refer to the same pattern, but at different offsets. In contrast, nesting the binders produces indices that refer to the different binding locations, but each one at the same offset (the zeroth variable in the pattern).

$$
\begin{array}{lll}
\mathit{bind}\ (-_\mathsf{x}, -_\mathsf{y})\ (x, y) & = \mathsf{Bind}\ (-_\mathsf{x}, -_\mathsf{y})\ (0\mathsf{@}0, 0\mathsf{@}1) & \\
\mathit{bind}\ -_\mathsf{x}\ (\mathit{bind}\ -_\mathsf{y}\ (x, y)) & = \mathsf{Bind}\ -_\mathsf{x}\ (\mathsf{Bind}\ -_\mathsf{y} & \\
 & \qquad (1\mathsf{@}0, 0\mathsf{@}0)) &
\end{array}
$$

Likewise, for pattern combinators that introduce internal binding, we also use *close* to replace occurrences of the bound variable with indices. Note that in the case of recursive binding, we close the pattern with respect to itself.

$$
\begin{array}{lll}
\mathit{rebind}\ p_1\ p_2 & = \mathsf{Rebind}\ p_1\ (\mathit{close}_\mathsf{P}\ p_1\ p_2) & \\
\mathit{rec}\ \ p & = \mathsf{Rec}\ (\mathit{close}_\mathsf{P}\ p\ p) &
\end{array}
$$

Finally, *embed* does not need to do any closing, and merely applies the Embed constructor to the given term. Likewise, *unembed* merely returns the nested term.

$$
\begin{array}{ll}
\mathit{string2Name} & :: \mathit{String} \to \mathsf{Name}\ T \\
\mathit{name2String} & :: \mathsf{Name}\ T \to \mathit{String} \\
 & \\
\mathit{bind} & :: P \to T \to \mathsf{Bind}\ P\ T \\
\mathit{unbind} & :: \mathit{Fresh}\ m \Rightarrow \mathsf{Bind}\ P\ T \to m\ (P, T) \\
 & \\
\mathit{rebind} & :: P_1 \to P_2 \to \mathsf{Rebind}\ P_1\ P_2 \\
\mathit{unrebind} & :: \mathsf{Rebind}\ P_1\ P_2 \to (P_1, P_2) \\
 & \\
\mathit{rec} & :: P \to \mathsf{Rec}\ P \\
\mathit{unrec} & :: \mathsf{Rec}\ P \to P \\
 & \\
\mathit{embed} & :: T \to \mathsf{Embed}\ T \\
\mathit{unembed} & :: \mathsf{Embed}\ T \to T
\end{array}
$$

Figure 5. Constructors and destructors

$$
\begin{array}{lll}
\mathit{embed}\ \ t & & = \mathsf{Embed}\ t \\
\mathit{unembed}\ (\mathsf{Embed}\ t) & & = t
\end{array}
$$

4.5 Freshening and unbinding

Unbinding is not quite as straightforward as binding. Given a term $\mathsf{Bind}\ p\ t$, it is only safe to call $\mathit{open}\ p\ t$ if none of the binding variables of p clash with existing free variables in t. Therefore, before opening, we must first *freshen* p by assigning suitably fresh names to its binders. At this point, we leave the precise meaning of "suitably fresh" open to interpretation; some concrete alternatives are discussed in § 6.4. We omit the formal definition of freshening since it is straightforward: it simply walks over a pattern, assigning a suitably fresh name to each binder encountered, and stopping at occurrences of Embed, since these are not part of the pattern. In our implementation, freshening also returns a permutation which describes how the variables were renamed, but we omit that here.

We can now define *unbind* as the operation that freshens the binding variables in the pattern and then opens the body of the binder with the new pattern.

$$
\begin{array}{l}
\mathit{unbind}\ (\mathsf{Bind}\ p\ t) = (p', \mathit{open}\ p'\ t) \\
\quad \mathbf{where}\ \mathsf{freshen}\ p \to p'
\end{array}
$$

In contrast, the destructors for Rebind and Rec *do not* freshen before opening the rest of the pattern. Instead, they use the preexisting names:

$$
\begin{array}{lll}
\mathit{unrebind}\ (\mathsf{Rebind}\ p_1\ p_2) & = (p_1, \mathit{open}_\mathsf{P}\ p_1\ p_2) \\
\mathit{unrec}\ \ (\mathsf{Rec}\ p) & = \mathit{open}_\mathsf{P}\ p\ p
\end{array}
$$

One might wonder: why the difference?

The reason is that when opening a Bind, we must generate fresh names for its binders. However, by the time we come to opening a *nested* binding, fresh names will have already been chosen for its binders when the enclosing Bind was opened. Hence there is no need to choose new names. In fact, we *must not* choose fresh names, since there may exist corresponding free variables over which we have no control. For example, consider the term $\mathsf{Bind}\ (\mathsf{Rebind}\ p_1\ p_2)\ t$, in which t may contain references to binders in p_1. If we use *unbind* to take it apart into $\mathsf{Rebind}\ p_1'\ p_2'$ and t', there will now be free variables in t' which match the names on binders in p_1'. Freshening p_1' again when opening the Rebind would destroy this connection, and in particular would mean that we could not reassemble the original term using *rebind* and *bind*. This is why Bind and Rebind must be distinct: if we used Bind everywhere, *unbind* would have no way of knowing whether it was opening a top-level Bind (which must first be freshened) or a nested one (which must not be).

$$
\begin{array}{lll}
\approx :: T \to T \to \mathit{Bool} & & \\
\mathsf{x} & \approx \mathsf{y} & = \mathsf{x} \equiv \mathsf{y} \\
b_1 & \approx b_2 & = b_1 \equiv b_2 \\
(\mathsf{K}\ s_1 \ldots s_n) & \approx (\mathsf{K}\ t_1 \ldots t_n) & = s_1 \approx t_1 \wedge \ldots \wedge s_n \approx t_n \\
(\mathsf{Bind}\ p_1\ t_1) & \approx (\mathsf{Bind}\ p_2\ t_2) & = p_1 \approx_\mathsf{P} p_2 \wedge t_1 \approx t_1 \\
\approx_\mathsf{P} :: P \to P \to \mathit{Bool} & & \\
\__\mathsf{x} & \approx_\mathsf{P} \__\mathsf{y} & = \mathsf{True} \\
(\mathsf{K}\ p_1 \ldots p_n) & \approx_\mathsf{P} (\mathsf{K}\ q_1 \ldots q_n) & = p_1 \approx_\mathsf{P} q_1 \wedge \ldots \\
 & & \quad \wedge\ p_n \approx_\mathsf{P} q_n \\
(\mathsf{Rebind}\ p_1\ p_2) & \approx_\mathsf{P} (\mathsf{Rebind}\ q_1\ q_2) & = p_1 \approx_\mathsf{P} q_1 \wedge p_2 \approx_\mathsf{P} q_2 \\
(\mathsf{Embed}\ t_1) & \approx_\mathsf{P} (\mathsf{Embed}\ t_2) & = t_1 \approx t_2 \\
(\mathsf{Rec}\ p_1) & \approx_\mathsf{P} (\mathsf{Rec}\ p_2) & = p_1 \approx_\mathsf{P} p_2
\end{array}
$$

$$
\begin{array}{lll}
\mathit{fv} :: T \to \mathit{Set}\ \mathbb{A} & & \\
\mathit{fv}\ \mathsf{x} & = \{\mathsf{x}\} & \\
\mathit{fv}\ b & = \varnothing & \\
\mathit{fv}\ (\mathsf{K}\ t_1 \ldots t_n) & = \mathit{fv}\ t_1 \cup \cdots \cup \mathit{fv}\ t_n & \\
\mathit{fv}\ (\mathsf{Bind}\ p\ t) & = \mathit{fv}_\mathsf{P}\ p \cup \mathit{fv}\ t &
\end{array}
$$

$$
\begin{array}{lll}
\mathit{fv}_\mathsf{P} :: P \to \mathit{Set}\ \mathbb{A} & & \\
\mathit{fv}_\mathsf{P}\ \__\mathsf{x} & = \varnothing & \\
\mathit{fv}_\mathsf{P}\ (\mathsf{K}\ p_1 \ldots p_n) & = \mathit{fv}_\mathsf{P}\ p_1 \cup \cdots \cup \mathit{fv}_\mathsf{P}\ p_n & \\
\mathit{fv}_\mathsf{P}\ (\mathsf{Rebind}\ p_1\ p_2) & = \mathit{fv}_\mathsf{P}\ p_1 \cup \mathit{fv}_\mathsf{P}\ p_2 & \\
\mathit{fv}_\mathsf{P}\ (\mathsf{Embed}\ t) & = \mathit{fv}\ t & \\
\mathit{fv}_\mathsf{P}\ (\mathsf{Rec}\ p) & = \mathit{fv}_\mathsf{P}\ p &
\end{array}
$$

$$
\begin{array}{lll}
\mathit{subst} :: \mathbb{A} \to T \to T \to T & & \\
\mathit{subst}\ \mathsf{x}\ s\ \mathsf{y} \mid \mathsf{x} \equiv \mathsf{y} & = s & \\
\quad \mid \mathit{otherwise} & = \mathsf{y} & \\
\mathit{subst}\ \mathsf{x}\ s\ b & = b & \\
\mathit{subst}\ \mathsf{x}\ s\ (\mathsf{K}\ t_1 \ldots t_n) & = \mathsf{K}\ (\mathit{subst}\ \mathsf{x}\ s\ t_1) \ldots (\mathit{subst}\ \mathsf{x}\ s\ t_n) & \\
\mathit{subst}\ \mathsf{x}\ s\ (\mathsf{Bind}\ p\ t) & = \mathsf{Bind}\ (\mathit{subst}_\mathsf{P}\ \mathsf{x}\ s\ p)\ (\mathit{subst}\ \mathsf{x}\ s\ t) &
\end{array}
$$

$$
\begin{array}{lll}
\mathit{subst}_\mathsf{P} :: \mathbb{A} \to T \to P \to P & & \\
\mathit{subst}_\mathsf{P}\ \mathsf{x}\ s\ \__\mathsf{y} & = \__\mathsf{y} & \\
\mathit{subst}_\mathsf{P}\ \mathsf{x}\ s\ (\mathsf{K}\ p_1 \ldots p_n) & = \mathsf{K}\ (\mathit{subst}_\mathsf{P}\ \mathsf{x}\ s\ p_1) \ldots & \\
 & \quad (\mathit{subst}_\mathsf{P}\ \mathsf{x}\ s\ p_n) & \\
\mathit{subst}_\mathsf{P}\ \mathsf{x}\ s\ (\mathsf{Rebind}\ p_1\ p_2) & = \mathsf{Rebind}\ (\mathit{subst}_\mathsf{P}\ \mathsf{x}\ s\ p_1) & \\
 & \quad (\mathit{subst}_\mathsf{P}\ \mathsf{x}\ s\ p_2) & \\
\mathit{subst}_\mathsf{P}\ \mathsf{x}\ s\ (\mathsf{Embed}\ t) & = \mathsf{Embed}\ (\mathit{subst}\ \mathsf{x}\ s\ t) & \\
\mathit{subst}_\mathsf{P}\ \mathsf{x}\ s\ (\mathsf{Rec}\ p) & = \mathsf{Rec}\ (\mathit{subst}_\mathsf{P}\ \mathsf{x}\ s\ p) &
\end{array}
$$

Figure 6. α-equivalence, free variables, and substitution

4.6 Free variables, α-equivalence and substitution

Now we come to the real payoff of our representation, as we specify the basic UNBOUND operations of α-equivalence, free variable calculation, and capture-avoiding substitution, all shown in Figure 6. Their specifications are entirely straightforward—and, as described in the next section, proving things about them is not much harder!

The α-equivalence relation on terms is defined mutually with a notion of equivalence for patterns which ignores binders and checks that embedded terms are α-equivalent. This α-equivalence relation is *essentially* structural equality—the only reason it is not precisely structural equality is that name annotations on binders are ignored in PEQ_BINDER.

Computing free variables is equally straightforward. Since free and bound variables are distinguished syntactically, we need only recurse through terms and patterns collecting all the free variables we find.

Finally, we define substitution into terms and patterns. If we see the free variable we are substituting for, we replace it with the term being substituted; otherwise we simply recurse. We need do nothing special when recursing through binders: since free and bound variables are distinguished syntactically, we are in no danger of mistaking a bound variable for a free variable we should substitute for, or of accidentally capturing any free variables in the substituted term.

5. Metatheory

We've now defined a simple semantics for our pattern specification language in terms of a locally nameless representation. But how can we know whether this semantics is at all meaningful? Well, we prove stuff, of course!

In the previous section, we noted how straightforward the definitions of various operations were. Likewise, most of the proofs regarding these operations are also straightforward (but full of fiddly details). We therefore omit most of the proofs and give only brief sketches of a few. The fact that the proofs are straightforward is a testament to the elegance of the locally nameless representation.

There is already a lot of work to draw on for the metatheory of locally nameless representations in the single binder case [1, 2]; much of the metatheory here can be seen as an extension of this work.

5.1 Local closure

One important property of the locally nameless representation is that only some terms are good representations. In particular, there is some "junk" in our representation, and we would like to know that we (and users of our library) never need to deal with it. The *local closure* relation in Figure 7 is an invariant for our representation. This relation excludes terms with "dangling" bound variables. For example the term $0@0$, a bound variable with no surrounding binder, is not locally closed.[5]

By making the type combinators of our library abstract, we can demonstrate a central property of our interface: *users of the library can only construct locally closed patterns and terms*.

Theorem 1 (Constructors and destructors preserve local closure)**.** *All exported term and pattern constructors and destructors preserve local closure.*

Proof. Requires considering the action of each constructor and destructor individually, appealing to a number of properties about the interaction of local closure, open and close. □

Theorem 2 (Substitution preserves local closure)**.**

- *If* $\mathsf{LC}\ t$ *and* $\mathsf{LC}\ t'$ *then* $\mathsf{LC}\ \mathit{subst}\ \mathsf{x}\ t\ t'$.
- *If* $\mathsf{LC}\ t$ *and* $\mathsf{LC}\ p$ *then* $\mathsf{LC}\ \mathit{subst}_\mathsf{P}\ \mathsf{x}\ t\ p$.

Proof. Straightforward induction using a generalized version of local closure. □

Lemma 3 (Freshening preserves local closure)**.** *If* $\mathsf{LC}\ p$ *and* $\mathsf{freshen}\ p \to p'$*, then* $\mathsf{LC}\ p'$.

Proof. Easy induction; freshening only changes names on binders, which the LC relation ignores. □

Next, we show that α-equivalence is an equivalence relation that is respected by the operations of our library.

[5] This relation is an extension of McKinna and Pollack's VClosed relation [14].

$\boxed{\mathsf{LC}\ t}$ t is locally closed

$$\frac{}{\mathsf{LC}\ \mathsf{x}}\ \text{LC\_FREE}$$

$$\frac{\mathsf{LC}\ t_1 \quad .. \quad \mathsf{LC}\ t_n}{\mathsf{LC}\ \mathsf{K}\ t_1\ ..\ t_n}\ \text{LC\_CON}$$

$$\frac{\mathsf{LC}\ p \quad \mathsf{LC}\ (\mathit{open}\ p\ t)}{\mathsf{LC}\ \mathsf{Bind}\ p\ t}\ \text{LC\_BIND}$$

$\boxed{\mathsf{LC}\ p}$ p is locally closed

$$\frac{}{\mathsf{LC}\ \__{\mathsf{x}}}\ \text{LCP\_BINDER}$$

$$\frac{\mathsf{LC}\ p_1 \quad .. \quad \mathsf{LC}\ p_n}{\mathsf{LC}\ \mathsf{K}\ p_1\ ..\ p_n}\ \text{LCP\_CON}$$

$$\frac{\mathsf{LC}\ p_1 \quad \mathsf{LC}\ (\mathit{open}_{\mathsf{P}}\ p_1\ p_2)}{\mathsf{LC}\ \mathsf{Rebind}\ p_1\ p_2}\ \text{LCP\_REBIND}$$

$$\frac{\mathsf{LC}\ t}{\mathsf{LC}\ \mathsf{Embed}\ t}\ \text{LCP\_EMBED}$$

$$\frac{\mathsf{LC}\ (\mathit{open}_{\mathsf{P}}\ p\ p)}{\mathsf{LC}\ \mathsf{Rec}\ p}\ \text{LCP\_REC}$$

Figure 7. Local closure of terms and patterns

Theorem 4. *$- \approx -$ is an equivalence.*

Proof. Reflexivity, symmetry, and transitivity can each be established by straightforward induction. □

Theorem 5 (fv respects α-equivalence)**.** *If $t_1 \approx t_2$, then $fv\ t_1 = fv\ t_2$.*

Proof. Straightforward induction on $fv\ \ t_1$ along with a similar proof for fv_{P}. □

Theorem 6 (Substitution respects α-equivalence)**.** *If $t_1 \approx t_2$ and $s_1 \approx s_2$, then $[x \mapsto s_1]t_1 \approx [x \mapsto s_2]t_2$.*

Proof. Straightforward induction on the derivation of $t_1 \approx t_2$, along with a similar proof for pattern substitution. □

We next specify how the operations of α-equivalence, free variable calculation, and substitution interact with Bind. The proofs of these remaining theorems rely on properties about the interactions between $close$ and each of the operations.

The first theorem states the interaction between binding and α-equivalence. It states that two bindings are α-equivalent when we can freshen two patterns to the same new result, and then show that their bodies are α-equivalent under a consistent renaming. Below, π_1 and π_2 are the permutations returned by freshen and $\pi_1 \cdot t_1$ is the application of a permutation to a term.

Theorem 7. *If* freshen $p_1 \to p, \pi_1$ *and* freshen $p_2 \to p, \pi_2$ *and $\pi_1 \cdot t_1 \approx \pi_2 \cdot t_2$ then $bind\ p_1\ t_1 \approx bind\ p_2\ t_2$.*

The second theorem specifies the behavior of fv for binders.

Theorem 8. *$fv\ (bind\ p\ t) = fv_P\ p \cup (fv\ t - binders\ p)$.*

Finally, we specify the conditions when substitutions are permitted to commute through bindings.

```
type N = Name E
data E = Var N
       | App E E
       | Lam (Bind N E)
  deriving Show
$(derive [''E])
instance Alpha E
instance Subst E E where
  isvar (Var n) = Just (SubstName n)
  isvar _       = Nothing
```

Figure 8. Representing the untyped lambda calculus

Theorem 9. *If $\{x\} \cup fv\ t$ is disjoint from binders p, then $subst\ x\ t\ (bind\ p\ t') = bind\ (subst_P\ x\ t\ p)\ (subst\ x\ t\ t')$.*

6. Implementation

The UNBOUND specification language is implemented as an embedded domain specific language (EDSL) in GHC Haskell, including all of the functionality described above and more. Terms and patterns are normal Haskell datatypes, and combinators such as Name, Bind and Rebind are abstract types provided by our library. The implementation of UNBOUND closely follows the semantics that we presented in the previous section, with UNBOUND operations such as fv, $subst$ and $\cdot \approx \cdot$ provided for user-defined datatypes via generic programming.

Below, we give an overview of our library, first by giving a short example of how it may be used in a Haskell program, and then discussing the implementation details. Figure 9 summarizes the important UNBOUND operations that we discuss in this section.

Figure 8 shows a definition of the untyped lambda calculus using UNBOUND. The first part of the figure is the same as in § 2. The rest of the figure includes the small amount of "boilerplate" necessary to use UNBOUND with the type E.

We implement UNBOUND using the REPLIB generic programming library [31]. REPLIB works by producing generic representation instances for each type. Roughly, a representation instance records the *structure* of a datatype by analyzing its **data** declaration. In this case, the call $\$(derive\ [''\ E])$ uses Template Haskell [27] to generate the generic representation for E. This structure information is used to automatically generate particular functions over E on demand.

The following line in the figure declares E to be an instance of the $Alpha$ type class, which governs α-equality, free variable and freshening operations. Happily, default implementations for the methods of $Alpha$ are defined generically. Guided by the occurrences of Bind and Name in the definition of E, the default definitions of these methods behave exactly like their specifications in the semantics section.

Capture-avoiding substitution is governed by the $Subst$ class, and requires a tiny bit of work on our part: we must indicate where variables are located in datatypes. Beyond that, the generic default implementation suffices. In general, the type of subst, declares that values of type b may be substituted for free variables occurring in values of type a, so the $Subst\ E\ E$ instance shown declares that E values may be substituted for Vars in other E values.

By making $Subst$ a multiparameter type class we have flexibility and safety. Imagine a different declaration of the type E which contains both variables abstracting E, and type variables abstracting Typ. An instance $Subst\ E\ E$ declares that E variables can be replaced with Es, and another instance $Subst\ \ Typ\ \ E$ would de-

$\cdot \approx \cdot$	$:: Alpha\ a \Rightarrow a \to a \to Bool$	-- alpha equivalence
$acompare$	$:: Alpha\ a \Rightarrow a \to a \to Ordering$	-- alpha-respecting comparison
fv	$:: (Alpha\ a, Rep\ b, Collection\ c) \Rightarrow a \to c\ (\mathsf{Name}\ b)$	-- free names (single sort)
$fvAny$	$:: (Alpha\ a, Collection\ c) \Rightarrow a \to c\ AnyName$	-- free names (all)
fv_{P}	$:: (Alpha\ a, Rep\ b, Collection\ c) \Rightarrow a \to c\ (\mathsf{Name}\ b)$	-- free names in annotations (single)
$fv_{\mathsf{P}}Any$	$:: (Alpha\ a, Collection\ c) \Rightarrow a \to c\ AnyName$	-- free names in annotations (all)
$binders$	$:: (Alpha\ a, Rep\ b, Collection\ c) \Rightarrow a \to c\ (\mathsf{Name}\ b)$	-- binding names (single sort)
$bindersAny$	$:: (Alpha\ a, Collection\ c) \Rightarrow a \to c\ AnyName$	-- binding names (all)
$freshen$	$:: (Alpha\ a, Fresh\ m) \Rightarrow a \to m\ (a, Perm\ AnyName)$	-- rename with fresh variables (returns a permutation)
$swaps$	$:: Alpha\ a \Rightarrow Perm\ AnyName \to a \to a$	-- permute variables
$subst$	$:: Subst\ b\ a \Rightarrow \mathsf{Name}\ b \to b \to a \to a$	-- single substitution
$substs$	$:: Subst\ b\ a \Rightarrow [(\mathsf{Name}\ b, b)] \to a \to a$	-- simultaneous substitution

Figure 9. Overview of selected UNBOUND operations

clare that Typ variables can be replaced with Typs inside Es. In the latter case, we would use the default definition of $isvar$ as there is no way to replace Typ variables with Typs an get an E. In fact, the type indices of Name and $SubstName$ would not allow us to give a definition of $isvar$ that would confuse Typ and E variables.

The operations fv, $bind$, $unbind$ and $subst$ are implemented in terms of the $Alpha$ and $Subst$ type classes. Therefore, Figure 8 provides all the necessary definitions for the parallel reduction example in Figure 1 (which is valid Haskell code).

6.1 Multi-sorted names and $AnyName$

Instead of a single homogeneous set of atomic names, UNBOUND has multisorted names. Consider the type declarations in Figure 9. Names are indexed by a type, and the type of $subst$ ensures that only things of type t may be substituted for t-indexed names. The fv, fv_{P}, and $binders$ functions are also polymorphic in their result type, ignoring names whose type index does not match the requested result type. In this way, one may calculate, say, just the free term variables or just the free type variables from an expression. These functions are overloaded, so type inference determines precisely what sort of names will be calculated, and what sort of data structure (list, set, etc.) will be used to collect them.

However, sometimes we would like to know all free names, no matter what their sort. Therefore, UNBOUND also provides the type $AnyName$, which existentially hides the type index on a name, and the functions $fvAny$ and $bindersAny$ which return *all* appropriate names wrapped in $AnyName$ constructors.

6.2 The $Alpha$ type class

One way in which our implementation differs from our semantics is that while the semantics statically differentiates between *terms* and *patterns*, the implementation does not. REPLIB limitations that instead of having two type classes $Term\ a$ and $Pattern\ a$, we must have a single type class $Alpha\ a$ which serves both purposes. This conflation means that our implementation cannot statically prevent meaningless types which use a pattern as a term (i.e. $\mathsf{Embed}\ (\mathsf{Embed}\ N)$) or a term as a pattern (i.e. $\mathsf{Bind}\ (\mathsf{Bind}\ N\ E)\ E$) from being used in a binding specification.[6]

However, this conflation does have an advantage. The operations of the $Alpha$ class are actually parameterized by a *mode* which determines whether the type is being used as a term or a pattern. For example, the parameterized free variable function fv' in the Name instance collects the Name in term mode (because it is a free name) but ignores it in pattern mode (because it is part of a pattern binding). Many types (such as products and sums) are parametric in the mode and use the same behavior in both cases.

6.3 Specific instances for $Alpha$

Default implementations for $Alpha$ methods are defined via generic programming, but they may be overridden for greater control or customization. This capability is necessary in practical uses of UNBOUND for specific types.[7]

For example, suppose we would like to tag variables with source position information in our abstract syntax:

$$\begin{aligned}&\mathbf{data}\ E = \ldots\\ &\qquad \mid Var\ SourcePos\ N\end{aligned}$$

To make E an instance of $Alpha$, we need $SourcePos$ to also be an instance of $Alpha$, because it appears inside the E type. If we would like α-equivalence to ignore source positions, we can simply override the default definition of α-equivalence for $SourcePos$. By identifying all source positions as equivalent, we ensure that expressions appearing in different positions can still be determined to be α-equivalent.[8]

$$\begin{aligned}&\mathbf{instance}\ Alpha\ SourcePos\ \mathbf{where}\\ &\quad aeq'\ \_\ \_\ \_ = \mathsf{True}\end{aligned}$$

6.4 Freshness monads

Since the freshen operation relies on the generation of fresh names, operations which make use of it (such as $unbind$) must execute within a monad. The $Fresh$ type class, shown in Figure 10, governs those monads which can be used for this purpose, and is used to avoid tying users down to one particular concrete monad.

The $Fresh$ class is quite simple: it requires only a single operation $fresh$, which takes a name as input and generates a new name (based on the given name) which is guaranteed to be "globally fresh" in some appropriate sense. For example, a simple concrete implementation might keep track of a global counter which is incremented every time $fresh$ is called, appending the new counter value to the given base variable name.

However, this is unsatisfactory in many instances. For example, an implementation of a pretty-printer for the lambda calculus based on $fresh$ might format the term $\lambda -_{\mathtt{x}} \lambda -_{\mathtt{y}} \lambda -_{\mathtt{z}}$ (2@0 1@0) 0@0 as `λ x1 -> λ y2 -> λ z3 -> (x1 y2) z3`. We can see perfectly

[6] We do, however, provide dynamic checks $isPat$ and $isTerm$ that can be used to ensure the invariants are maintained.

[7] This capability for overriding generic functions is inspired by the SYB3 library [12].

[8] The first argument to aeq' is the *mode* information mentioned earlier.

```
class Monad m ⇒ Fresh m where
  fresh :: Name a → m (Name a)

class Monad m ⇒ LFresh m where
  lfresh :: Rep a ⇒ Name a → m (Name a)
  avoid :: [AnyName] → m a → m a
```

Figure 10. The *Fresh* and *LFresh* type classes

well that the numeric suffixes are unnecessary, since the existing names do not clash, but the *fresh* operation does not have enough information to make this assessment.

For this reason, we also provide the more sophisticated *LFresh* type class for monads which can generate *locally fresh* names. *lfresh* works much like *fresh*, but it has more to go on: *avoid ns m* allows us to specify that the names *ns* should be avoided by *lfresh* in the subcomputation *m*. Unlike *fresh*, *lfresh* is not guaranteed to pick *globally* fresh names; it only guarantees not to pick names proscribed by an enclosing call to *avoid*. The intention is that it will return its argument unchanged when that name is not specifically to be avoided.

Using *LFresh*, a pretty-printer for the lambda calculus can be written which uses *avoid* every time it recurses under a binder, so that names are chosen fresh with respect to exactly those names currently in scope.

UNBOUND provides several concrete implementations for *Fresh* and *LFresh*, including transformer versions for adding fresh name generation capabilities to existing monads, and instances allowing their use with all the standard monad transformers in the `transformers` package.[9]

6.5 Simultaneous unbinding

Up to now, we have seen only examples of opening a single abstraction with *arbitrary* fresh names. However, some situations require simultaneously opening two or more abstractions with the *same* fresh names. For example, in order to check the convertibility of two LF Π-types, we must open them with the same fresh name and recursively check the convertibility of the bodies.

In order to simultaneously open two abstractions Bind p_1 t_1 and Bind p_2 t_2, we require only that p_1 and p_2 have the same number of binders. Requiring a stronger match between p_1 and p_2 would be unnecessarily limiting. For example, continuing our LF checking example, p_1 and p_2 might contain type annotations which are convertible but not α-equivalent.

Therefore, UNBOUND provides a function $unbind_2$ that simultaneously opens two related bindings.

```
unbind2 :: (Fresh m, Alpha p1, Alpha p2,
                     Alpha t1, Alpha t2) ⇒
           Bind p1 t1 → Bind p2 t2 →
           m (Maybe (p1, t1, p2, t2))
unbind2 (B p1 t1) (B p2 t2) = do
  case mkPerm (bindersAny p1) (bindersAny p2) of
    Just π → do
      (p1', π') ← freshen p1
      return (Just (p1', open p1' t1,
                    swaps (π' ∘ π) p2, open p1' t2))
    Nothing → return Nothing
```

This function works by first matching the binding variables of the two patterns together to create a permutation π. This operation will fail if the patterns bind different numbers of variables. Next, it freshens the first pattern p_1 and uses the result to open t_1 and t_2. Finally, it must compose the permutation from freshening p_1 with that from the match, and use the new permutation to rename the second pattern.

[9] `http://hackage.haskell.org/package/transformers`

7. Discussion

7.1 Nominal semantics

We have presented a semantics for the UNBOUND specification language in terms of a locally nameless representation, but this is not our only possible choice. UNBOUND could also be specified via an equivalent *nominal* semantics [17], and we are working in parallel on a nominal-style Haskell implementation. Such an alternative semantics would provide differences in running time/space, but otherwise would behave identically to the locally nameless version. In future work, we plan to formalize the precise connection between the two formulations and prove their equivalence with respect to the abstract interface provided by the library.

Although a nominal semantics might appear more natural to think about, in our experience the locally nameless semantics is far easier to understand when it comes to generalized binding patterns, especially with nesting. Therefore, an important contribution of this work is the identification of a *simple* semantics for pattern binding.

7.2 Cαml-style specifications

François Pottier's Cαml system [19] features a single-argument *abstraction* constructor, inside which patterns and terms (both bound and unbound) can be mixed. Directly inside an abstraction is a pattern, with terms embedded via outer (indicating a term outside the scope of the pattern) or inner (indicating that the pattern binds names in the term). Cαml's abstraction constructor $\langle p \rangle$ is easily definable with UNBOUND as Bind (Rec p) (). Within that pattern, occurrences of Embed are analogous to occurrences of inner in Cαml. To account for Cαml's outer scope specification, we generalize the Embed combinator by adding a natural number subscript. When encountering Embed_n while doing an *open* or *close* operation, we *decrement* the level by n. Hence, Embed_0 corresponds to the original Embed, and Embed_1 corresponds to Cαml's outer construct, since it shifts the scope of an embedded term out to the next enclosing level.

In UNBOUND we implement indexed embedding by adding a new type combinator Shift P. This type increments the index of its argument, so Shift (Embed T) corresponds to Embed_1 T, and Shift (Shift (Embed T)) is Embed_2 T. Operationally, all Shift does is decrement the binding level when the pattern is opened or closed so that variables will resolve to an outer scope.

7.3 UNBOUND in practice

We have been using UNBOUND in the TRELLYS project, in the context of type checking and evaluation of an experimental dependently-typed language. In this context, UNBOUND support for telescopes is essential. Our experience with TRELLYS has allowed us to find and correct a few bugs in our implementation of UNBOUND, but for the most part the use of UNBOUND has been unremarkable, in the sense that it seems to "just work".

TRELLYS is an ideal client for UNBOUND, in that it is a prototype language with a greater emphasis on semantics than performance. Because UNBOUND is implemented using generic programming, it will be slower than a hand-coded implementation [23]. If necessary, we could easily replace the generic definitions with hand-coded operations by overriding the *Alpha* type class instances.

The locally nameless representation does have some performance concerns with respect to its use of *open* and *close*. While the standard operations *fv*, *subst*, and $\cdot \approx \cdot$ are linear in the size

of terms, operations that are defined in the freshness monad must open and close the terms for each binding level, which could be expensive. Although we have not had any difficulties of this sort in TRELLYS (our experiments are still small) it is possible that with larger programs and deeper binding depths, these operations could dominate. To mitigate this difficulty, UNBOUND supports an "experts-only" interface, where critical operations can be written directly over the terms (in a manner similar to our implementations of fv, $subst$ and $\cdot \approx \cdot$).

We have not explored the interaction of UNBOUND with standard optimizations [26]. For example, by caching free names, an implementation of substitution could stop early if the name being substituted for is not cached. If we remove names from bound patterns (which are preserved only for error messages) the locally nameless representation interacts nicely with hash-consing, as all α-equivalent terms have the same representation.

We are also working on bringing our nominal implementation of UNBOUND up to date with our reference locally nameless implementation. Both implementations provide the same interface to clients. When using the $LFresh$ monad, the nominal implementation could avoid freshening when unbinding patterns if the patterns were already "sufficiently" fresh. However, there are trade-offs involved; for example, α-equivalence can be more expensive with the nominal version.

One important contribution of UNBOUND is that it provides an abstract interface to name binding. Clients can write their code against this interface, and, depending on their particular application, choose the most appropriate implementation. Importantly, our locally nameless implementation provides a simple reference semantics for this interface, and alternative implementations may use this semantics to prove their correctness.

8. Related Work

Locally nameless representation The locally nameless representation dates back to the introduction of de Bruijn indices, and is mentioned in the conclusion of de Bruijn's paper [7]. The key idea is even older. It rests on the separation of names into two distinct classes: variables (for locally bound variables) and parameters (for free, or globally bound variables) and goes back to Kleene [11], Prawitz [22] and Gentzen [9]. The full history of the locally nameless representation is outside the scope of this paper, but we refer to Aydemir et al [2] and Charguéraud [4] which discusses it in detail. Instead, we focus on the interaction between this representation, generic programming and binding specifications.

Charguéraud's paper [4] also gives several examples of locally-nameless representations of languages with specific binding forms, including binding a list of variables, pattern matching (with embedded terms), and recursive bindings. However, he does not consider a compositional framework for describing generalized binding.

Zappa Nardelli's locally nameless backend [32] for the OTT tool [25] automatically generates definitions for the Coq proof assistent given a specification of a language (with single binding only). These definitions include a locally nameless representation of the syntax, $open$ and $close$ operations, α-equivalence, substitution, and free variable calculation. The LNgen tool of Aydemir and Weirich [1] augments this output with generic proofs about this representation, including many of the properties of § 5.

Tools for general bindings The OTT tool provides an expressive specification language for generalized binding in programming languages. In conjunction with a specification of the abstract syntax, OTT allows the definition of *bindspecs*: functions that arbitrarily select the binding variables that appear in terms and bind them elsewhere in the abstract syntax. They give a semantics of this specification language using a representation with concrete variable names [24] and show that under appropriate conditions, their concrete substitution functions respect α-equivalence and coincide with capture-avoiding substitution.

Inspired by the OTT specification language, Urban and Kaliszyk recently extended the Nominal Isabelle proof assistant with support for general bindings [30]. Their system works by using the OTT binding specifications (with some restrictions) to define α-equivalence classes of syntax with binders which they use to model nominal-logic specifications. While a direct comparison is difficult, their restrictions prevent variables from being bound by two different bindspecs, which seems necessary for telescopes. On the other hand, they also add two forms of *set* bindings to their specifications, allowing binders to be equivalent up to permutation and weakening of their patterns.

As discussed in § 7.2, there is a close connection between UNBOUND and Francois Pottier's Cαml system [19], based on a nominal semantics for binding. One major difference is that Cαml explicitly does not include support for nested binders. Another difference is that Cαml is an external tool that performs a preprocessing step, whereas UNBOUND is a library. However, this is not a fundamental difference; Cαml could be made into a library as well if OCaml had better support for generic programming (likewise, UNBOUND could be ported to languages without support for generic programming by making it into an external tool).

Cheney's FreshLib [6] is a Haskell library which served as an inspiration for UNBOUND. Like UNBOUND, it uses generic programming to automatically define α-equivalence and substitution functions (although FreshLib is based on a nominal semantics for name binding, so the generic operations that establish its semantics are different). FreshLib also supports some forms of generalized binding, but does not give a generic treatment of patterns.

Other tools based on nominal logic include FreshML [29] and FreshOCaml [28]. However, they support limited forms of binding patterns which do not include embeddings, recursive or nested bindings. Likewise, the Haskell Nominal Toolkit [3] is a library that supports single binding for a fixed term structure.

9. Future work

Although we believe UNBOUND is useful in its current state, there are several directions in which we would like to extend it.

Other forms of "exotic" binding Cheney [5] gives a catalogue of "exotic" binding, renaming, and structural congruence situations. Although UNBOUND can express many of these examples, we hope to extend UNBOUND with better support for "global" binding, such as that used for objects and modules. Furthermore, we would also like to add support for set binding similar to Nominal Isabelle [30]. However, implementing $unbind_2$ in the presence of such bindings is a nontrivial task.

User-defined names The current implementation of our library imposes $String$ as the type of atomic names. However, there is nothing particularly special about $String$s; in theory, any type with decidable equality could be used. Inspired by Cheney's FreshLib [6], we intend to explore extending our library to support the use of arbitrary user-supplied types as atomic names. We have so far avoided this generalization to simplify definition of the $Alpha$ type class. On the other hand, the additional flexibility may also help us to share code among our different implementations.

Better static distinction between names and patterns As discussed in § 6.2, an infelicity of our current design is the ability to get terms and patterns mixed up. UNBOUND inherits this limitation from REPLIB; it is possible that an alternative framework for generic programming would perform better in this respect.

Scoping UNBOUND does not keep track of the scope of names once they have been unbound. While this leads to a familiar and flexible interface, it does not rule out bugs that could occur from names escaping their scope. Pottier *et al.* have made some progress in this respect [20, 21], and we would like to explore a variant interface for UNBOUND that provides this tighter control.

Mechanized metatheory The UNBOUND specification language seems ideal for incorporation into tools like Ott, LNgen and Nominal Isabelle that assist in the formalization of programming language metatheory. Indeed, locally nameless representations have already proved useful for that sort of reasoning.

10. Conclusion

UNBOUND is an expressive specification language for generalized binding structures, defined with a simple compositional semantics, proven correct, and immediately available to the GHC user community. Because it supports the rapid development of typecheckers, compilers, and interpreters, it is a valuable tool for exploration in programming language design. Furthermore, we hope that the design of UNBOUND itself will be a model for future work on library support for expressive binding structure.

Acknowledgments

Chris Casinghino, Vilhelm Sjöberg and the rest of the TRELLYS team provided valuable feedback on this work. Thanks are also due to the Penn PLClub for feedback on an early draft of this paper. This material is based upon work supported by the National Science Foundation under grant 0910500.

References

[1] B. Aydemir and S. Weirich. LNgen: Tool support for locally nameless representations. Technical Report MS-CIS-10-24, Computer and Information Science, University of Pennsylvania, June 2010.

[2] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 3–15, Jan. 2008.

[3] C. Calvès. A Haskell nominal toolkit. `http://www.inf.kcl.ac.uk/pg/calves/hnt/`, Jan. 2009.

[4] A. Charguéraud. The locally nameless representation. *Journal of Automated Reasoning, Special Issue on the POPLmark Challenge*, 2011. To appear.

[5] J. Cheney. Toward a general theory of names: binding and scope. In *Proceedings of the 3rd ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding*, MERLIN '05, pages 33–40, New York, NY, USA, 2005. ACM.

[6] J. Cheney. Scrap your nameplate: (functional pearl). In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 180–191, New York, NY, USA, 2005. ACM. ISBN 1-59593-064-7.

[7] 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(5):381–392, 1972.

[8] N. G. de Bruijn. Telescopic mappings in typed lambda calculus. *Information and Computation*, 91(2):189 – 204, 1991.

[9] G. Gentzen. *The Collected Papers of Gerhard Gentzen*. North-Holland, 1969. Edited by Mandred Szabo.

[10] A. D. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In J. Joyce and C. Seger, editors, *Higher-order Logic Theorem Proving And Its Applications, Proceedings, 1993*, volume 780 of *Lecture Notes in Computer Science*, pages 414–426. Springer, 1994.

[11] S. Kleene. *Introduction to Metamathematics*. ban Nostrand Rheinhold, Princeton, USA, 1952.

[12] R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*, pages 204–215. ACM Press, Sept. 2005.

[13] C. McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999. Available from `http://www.lfcs.informatics.ed.ac.uk/reports/00/ECS-LFCS-00-419/`.

[14] J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3–4):373–409, 1999.

[15] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[16] B. Pientka and J. Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In J. Giesl and R. Hähnle, editors, *IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 15–21. Springer, 2010.

[17] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[18] A. Poswolsky and C. Schürmann. System description: Delphin – a functional programming language for deductive systems. *Electron. Notes Theor. Comput. Sci.*, 228:113–120, January 2009.

[19] F. Pottier. An overview of Cαml. In *ACM Workshop on ML*, pages 27–52, Mar. 2006.

[20] F. Pottier. Static name control for FreshML. In *In IEEE Symposium on Logic in Computer Science*, pages 356–365, 2007.

[21] N. Pouillard and F. Pottier. A fresh look at programming with names and binders. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 217–228, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3.

[22] D. Prawitz. *Natural Deduction: Proof Theoretical Study*. Almquist and Wiksell, Stockholm, 1965.

[23] A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. *SIGPLAN Not.*, 44:111–122, September 2008.

[24] S. Sarkar, P. Sewell, and F. Zappa Nardelli. Binding and substitution. `www.cl.cam.ac.uk/~pes20/ott/bind-doc.pdf`, Oct. 2007.

[25] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1):71–122, 2010.

[26] Z. Shao. Implementing typed intermediate language. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 313–323, Baltimore, Maryland, September 1998.

[27] T. Sheard and S. P. Jones. Template meta-programming for Haskell. *SIGPLAN Not.*, 37:60–75, December 2002.

[28] M. Shinwell and A. Pitts. *Fresh Objective Caml User Manual*. Cambridge University Computer Laboratory, Feb. 2005.

[29] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*, pages 263–274. ACM Press, Aug. 2003.

[30] C. Urban and C. Kaliszyk. General bindings and alpha-equivalence in Nominal Isabelle. In *20th European Symposium on Programming (ESOP'11)*, Mar. 2011. To appear.

[31] S. Weirich. RepLib: A library for derivable type classes. In *Haskell Workshop*, pages 1–12, Portland, OR, USA, Sept. 2006.

[32] F. Zappa Nardelli. A locally-nameless backend for Ott. `moscova.inria.fr/~zappa/projects/ln_ott/`, 2009.