Hobbits for Haskell: A Library for Higher-Order Encodings in Functional Programming Languages

Edwin Westbrook
Rice University
emw4@rice.edu

Nicolas Frisby
University of Kansas
nfrisby@ittc.ku.edu

Paul Brauner
Rice University (Current affiliation: Google Inc)
polux@google.com

Abstract

Adequate encodings are a powerful programming tool, which eliminate whole classes of program bugs: they ensure that a program cannot generate ill-formed data, because such data is not part of the representation; and they also ensure that a program is well-defined, meaning that it cannot have different behaviors on different representations of the same piece of data. Unfortunately, it has proven difficult to define adequate encodings of programming languages themselves. Such encodings would be very useful in language processing tools such as interpreters, compilers, model-checking tools, etc., as these systems are often difficult to get correct. The key problem in representing programming languages is encoding binding constructs; previous approaches have serious limitations in either the operations they allow or the correctness guarantees they make.

In this paper, we introduce a new library for Haskell that allows the user to define and use higher-order encodings, a powerful technique for representing bindings. Our library allows straightforward recursion on bindings using pattern-matching, which is not possible in previous approaches. We then demonstrate our library on a medium-sized example, lambda-lifting, showing how our library can be used to make strong correctness guarantees at compile time.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features

General Terms Languages, Design, Theory

1. Introduction

Having the right data representation can vastly affect the ease with which we write correct programs. Ideally, one would like to use a representation that is an adequate encoding, meaning that the relationship between the data being represented and the representations of that data is a one-to-one correspondence. Adequate encodings eliminate whole classes of program bugs: they ensure that a program cannot generate ill-formed data, because such data is not part of the representation; and they also ensure that a program is well-defined, meaning that it cannot have different behaviors on different representations of the same piece of data.

Unfortunately, it has proven difficult to define adequate encodings of programming languages themselves. Such encodings would be very useful in language processing tools such as interpreters, compilers, model-checking tools, etc., as these systems are often difficult to get correct. The key problem in representing programming languages is in encoding binding constructs, such as the simply-typed $\lambda$-calculus function

$$\lambda x : A. x$$

Such constructs bind a name, in this case $x$, that can only be used inside the scope of the binding. Defining adequate encodings of bindings is difficult because of $\alpha$-Equivalence and scoping, which state respectively that bindings are equal up to renaming of bound names and that names cannot occur outside of bindings. For example, straightforward representations based on de Bruijn indices do not satisfy scoping because nothing in such representations prevents an occurrence of an unbound name.

There has been much research into encodings for bindings in type theory [11, 21, 23, 28], logic [9, 16], and programming languages [13, 22]. So far, however, the only approaches that can ensure both $\alpha$-Equivalence and Scoping — and that can be used in existing programming languages — are typed de Bruijn indices [2, 26] and Higher-Order Abstract Syntax (HOAS) [7, 14, 19, 29]. Both of these are difficult to use: typed de Bruijn indices require a good deal of arithmetic (both at the term and the type level); while HOAS requires all operations on bindings to be expressed as folds, which cannot directly express many desired operations (without adding significant support to the programming language as done by e.g. Pientka [21]). In fact, the state of the art generally requires using both representations, and switching between the two when necessary [2, 5].

In this work, we describe a library, HobbitLib (for Higher-Order Bindings), that allows higher-order encodings in the Haskell programming language. By a higher-order encoding, we mean an encoding of data with name-bindings in which each bound name has an associated type in the host language (which is Haskell, in this case). Readers familiar with HOAS will note that we use the term “higher-order encoding” in a more general sense than HOAS, where we view the latter as a specific instance of the former. Higher-order encodings allow us to build adequate encodings of typed programming languages. For example, we can build an encoding using our approach that does not allow the ill-typed (under simple typing) $\lambda$-term $\lambda x : A. x$. This in turn ensures that any program we write over such encodings, such as a compiler or interpreter, is guaranteed to produce well-formed, well-typed programs, removing whole classes of bugs from such systems.

At a high level, our approach is based on the Calculus of Nominal Inductive Constructions (CNIC), a type theory of bindings [30, 31]. CNIC encodes bindings with a construct called the $\nu$-abstraction, a higher-order construct which captures the expected properties of bindings and also allows useful features such as
pattern-matching on bindings and comparing names. At a low level, we show here that \( \nu \)-abstractions can be defined in functional languages like Haskell as pairs \((a, \text{body})\) of a fresh natural number \(n\) and a body \(\text{body}\) that is allowed to use \(n\) as a name. This is based on the observation by Gabbay and Pitts [9] that bindings are equivalent (in Nominal Logic) to pairs of a fresh name and a body of the binding. This definition makes it easy to define pattern-matching on bindings, a necessary ingredient in defining operations on data with bindings. In order to ensure that the number \(n\) stays fresh, we make these pairs opaque, allowing them to be manipulated only via a suite of operations guaranteed to maintain freshness.

The remainder of the paper is organized as follows. Section 2 defines exactly what properties we expect bindings to have and shows how our approach captures these properties, using the simply-typed \(\lambda\)-calculus as an example. Section 3 shows how to write operations, such as capture-avoiding substitution, on data representations using our approach to bindings. Section 4 gives a more extended example, lambda-lifting, implemented in \texttt{HOBBITLib}.

Section 5 explores the definition of \texttt{HOBBITLib} in Haskell, which includes both unsafe type casts and impure I/O operations. Section 6 proves that our approach is both type safe and pure, meaning both the unsafe casts used in \texttt{HOBBITLib} and its behavior; in actual programs, the user must use the functions defined in \texttt{HOBBITLib} to manipulate bindings.

As an example of using \texttt{HOBBITLib} to encode data with bindings, the simply-typed \(\lambda\)-terms can be encoded in \texttt{HOBBITLib} using the following GADT, where \texttt{Binding} \(a\) is the type of bindings \(\nu.n.e\) where \(n\) has type Name \(a\) and \(e\) has type \(b\):

\[
\texttt{data Term a where}
\]
\[
\texttt{Var :: Name a \rightarrow Term a}
\]
\[
\texttt{Lam :: Binding a (Term b) \rightarrow Term (a \rightarrow b)}
\]
\[
\texttt{App :: Term (a \rightarrow b) \rightarrow Term a \rightarrow Term b}
\]

Using this type, the simply-typed \(\lambda\)-terms can be encoded as follows, where \(\Theta\) is a mapping from bound variables to bound names in \texttt{HOBBITLib} and \(x \mapsto a\) builds the obvious extension of \(\Theta\):

\[
[\lambda x^n : A.t]^{\Theta} = \texttt{Var } \Theta(x)
\]
\[
[\lambda x : A.t]^{\Theta} = \texttt{Lam } (\nu.n.(\lambda x^n : A.t))
\]
\[
[t]^{\Theta} = \texttt{App } [\lambda x^n : A.t]^{\Theta} [t]^{\Theta}
\]

For example, the \(\lambda\)-term \(\lambda x : A.\lambda z : A.s z\) representing the church numeral 1 is encoded as follows:

\[
\texttt{Lam } (\nu.n. \texttt{Lam } (\nu.z. \texttt{App } (\texttt{Var } s) (\texttt{Var } z)))
\]

The benefit of this encoding is that it eliminates whole classes of program bugs. For one, the encoding is \textit{adequate}, meaning here that the values of type \(\texttt{Term } [A]_\chi\) with free names of type \(\texttt{Name } [B]_\chi\), through \(\texttt{Name } [B]_\chi\), are in bijective correspondence with the simply-typed lambda-terms of type \(A\) with free variables of type \(B_1\) through \(B_\chi\), where \([\_]_\chi\) is a suitable encoding of lambda-terms into Haskell types that maps function types to function types.\(^2\) This means that values of type \(\texttt{Term } a\) are guaranteed to be well-typed terms, and thus a program cannot accidentally generate (a representation of) an ill-formed term. The role that \texttt{HOBBITLib} plays in this guarantee is to ensure that (representations of) variables are always used in their correct scopes and at their correct types. Thus, for example, a user cannot write a function that accidentally removes a variable from its scope. In addition, due to the Freshness property, a function cannot accidentally capture a free variable when creating a new binding; avoiding accidental variable capture often requires complex code to get right. Finally, the \(\alpha\)-Equivalence property ensures that all representations for the same binding are indistinguishable, so the user cannot accidentally write a function that takes two different representations of the same term to different values.

3. Using Data with Bindings

In this section we show by example how to write operations that manipulate data with bindings using the primitives of \texttt{HOBBITLib}. We first cover some of the more basic primitives. Most of these primitives take a binding \(\nu.n.e\) and extract some value from \(e\) that is guaranteed not to contain a free. This is to ensure that the operation does not violate Scoping. \texttt{HOBBITLib} also contains a primitive to apply certain functions \(f\) to the body of a binding, that is, to take binding \(\nu.n.e\) and return \(\nu.n.(f \cdot e)\). Finally, Section 3.2 shows

\(^2\)Note that the bijection technically only holds for \textit{strict} values, since the lazy evaluation of Haskell allows the creation of infinite terms, meaning that \(\texttt{Term } a\) can in fact contain representations of infinite lambda terms.
how to pattern-match on the bodies of bindings by commuting constructors out of bindings.

In order to make pattern-matching (discussed below) easier to use, we generalize bindings to multi-bindings which bind zero or more fresh names. We write multi-bindings on paper as follows:

\[ \nu(n_1, \ldots, n_k).e \]

Similar to bindings, this construct is a pair of a list of fresh names and a body \( e \) in which those names are bound, where \( \nu \) is again an opaque constructor. The type of such a multi-binding is

\[ \text{Mb} \ (\text{CtxCons} \ldots \ (\text{CtxCons} \text{CtxNil} \ a_1) \ldots \ a_k) \ b \]

where \( n_1, \ldots, n_k \) are names of type Name \( a_1, \ldots, \text{Name} \ a_k \), respectively, and \( b \) has type \( b \). Note that the types are listed inside-out, meaning that the type of the name bound last in the multi-binding is listed first. This convention is useful in the common case, when more bindings are added to the inside of a multi-binding. This will become more clear in the examples below. The types CtxNil and CtxCons \( 1 \) are so-called phantom types \([10]\); i.e., they are only used to represent lists of types at the type level, and do not represent any data.

Figure 1 summarizes the external interface of HOBBITLib, other than the pattern-matching facility discussed in Section 3.2. This figure begins with a number of "helper" declarations.\(^3\) The first two lines declare the phantom types CtxNil and CtxCons for constructing type-level lists. The next declaration defines Bindings as multi-bindings that bind just one name. After this is a declaration for the type \( a :=: b \) of proof objects which prove that type \( a \) equals type \( b \). This is a GADT with one constructor, Refl, which proves that a \( :=: b \) for any type \( a \). Since Refl is the only constructor for \( a :=: b \), it is straightforward to see that a value of type \( a :=: b \) can only be constructed if type \( a \) and \( b \) are in fact the same types.

The next declaration in Figure 1 defines the type InCtx ctx a. This GADT defines proof objects that witness the fact that the type \( a \) is in the type context ctx. The first constructor, InCtxBase, is a proof that the last type in a type context is in that type context. The second constructor, InCtxStep, takes a proof that type \( a \) is in context ctx and builds a proof that a is in the result of adding any other type to the end of ctx. Given these declarations, it is straightforward to see that, given a value of type InCtx ctx a, it must be the case that a is listed somewhere in ctx.

The last helper declarations in Figure 1 define an append operation ctx1 ++: ctx2 on type contexts ctx1 and ctx2. This operation uses the Haskell type family functionality that allows for the definition of computations on types. The first instance declaration states that the type ctx1 ++: CtxNil is equal to the type ctx1, while the second states that ctx1 ++: (CtxCons ctx2 a) equals the result of appending ctx2 to the end of ctx1 and then appending \( a \). Figure 1 also defines the type IsAppend ctx1 ctx2 x of proof objects that witness the fact that appending ctx1 and ctx2 yields ctx. The definition of this type mirrors the definition of the ++: type function.

After these helper declarations, Figure 1 gives the binding-related operations of HOBBITLib. First is the nu operator, which is the way to create bindings in HOBBITLib. Intuitively, this operator creates a binding by generating a fresh name and passing it to a user-supplied function \( f \), which returns the body of the newly created binding. More technically, nu \( f \) generates a fresh name \( n \) and then reduces to \( \text{nu} f n \). where the reduction of \( f n \) can then occur inside the binding. For example, the code

\[
\text{Lam} \ (\text{nu} (\backslash n \rightarrow \text{Lam} \ (\text{nu} (z \rightarrow \text{App} \ (\text{Var} \ s) \ (\text{Var} \ z)))))
\]

creates the church numeral 1 given above. Note that "generates a fresh name" sounds at first like it violates Haskell's purity restriction that allows only side-effect-free code. Indeed, inside the HOBBITLib library, \( \text{nu} \) uses the infamous unsafePerformIO operation. We show in Section 6, however, that \( \text{nu} \) is effectively pure, meaning that its side effects cannot be observed (except inside HOBBITLib).

After \( \text{nu} \), Figure 1 declares emptyMb and elimEmptyMb. These intuitively add and remove, respectively, an empty multi-binding with no names around a value. Thus, for example, if \( i \) is an integer then emptyMb i intuitively returns the result \( \nu (i \cdot 3) \), and applying \( \text{elimEmptyMb} \) to this result removes the empty name list and returns \( i \) again.

Next comes the combineMb operation, which takes two nested multi-bindings and combines them into a single multi-binding containing the names of both multi-bindings. The type of combineMb expresses that it takes a multi-binding for ctx1 that contains a nested multi-binding for ctx2 and returns a multi-binding for

\[
\text{-- phantom types}
\text{data CtxNil}
\text{data CtxCons 1 a}
\text{-- bindings are one-name multi-bindings}
\text{type Binding a b = Mb (CtxCons CtxNil a) b}
\text{-- proofs of type equality}
data a :=: b where Refl :: a :=: a
\text{-- proofs that a type is in a context}
data InCtx ctx a where
\text{InCtxBase :: InCtx (CtxCons ctx x) a}
\text{InCtxStep :: InCtx ctx a \rightarrow InCtx (CtxCons ctx b) a}
\text{-- context append}
type family (ctx1 ++: ctx2)
type instance ctx1 ++: CtxNil = ctx1
type instance ctx1 ++: (CtxCons ctx2 a) = (CtxCons (ctx1 ++: ctx2) a)
\text{-- proofs of context append}
data IsAppend ctx1 ctx2 ctx where
\text{IsAppendBase :: IsAppend ctx CtxNil ctx}
\text{IsAppendStep :: IsAppend ctx1 ctx2 ctx \rightarrow IsAppend ctx1 (CtxCons ctx2 x) (CtxCons ctx x)}
\text{-- operations of HobbitLib}
\text{nu :: (Name a \rightarrow b) \rightarrow (Binding a b)}
\text{emptyMb :: a \rightarrow Mb CtxNil a}
\text{elimEmptyMb :: Mb CtxNil a \rightarrow a}
\text{combineMb :: Mb ctx1 (Mb ctx2 a) \rightarrow Mb (ctx1 ++: ctx2) a}
\text{separateMb :: IsAppend ctx1 ctx2 ctx \rightarrow Mb ctx b \rightarrow Mb ctx1 (Mb ctx2 b)}
\text{cmpName :: Name a \rightarrow Name b \rightarrow Maybe (a :=: b)}
\text{mbNameBoundP :: Mb ctx (Name a) \rightarrow Either (InCtx ctx a) (Name a)}
\text{mbToplevel :: SuperComb (a \rightarrow b) \rightarrow Mb ctx a \rightarrow Mb ctx b}
\]

\( \begin{figure}[h]
\begin{verbatim}
\text{-- phantom types}
data CtxNil
data CtxCons 1 a
\text{-- bindings are one-name multi-bindings}
\text{type Binding a b = Mb (CtxCons CtxNil a) b}
\text{-- proofs of type equality}
data a :=: b where Refl :: a :=: a
\text{-- proofs that a type is in a context}
data InCtx ctx a where
\text{InCtxBase :: InCtx (CtxCons ctx x) a}
\text{InCtxStep :: InCtx ctx a \rightarrow InCtx (CtxCons ctx b) a}
\text{-- context append}
type family (ctx1 ++: ctx2)
type instance ctx1 ++: CtxNil = ctx1
type instance ctx1 ++: (CtxCons ctx2 a) = (CtxCons (ctx1 ++: ctx2) a)
\text{-- proofs of context append}
data IsAppend ctx1 ctx2 ctx where
\text{IsAppendBase :: IsAppend ctx CtxNil ctx}
\text{IsAppendStep :: IsAppend ctx1 ctx2 ctx \rightarrow IsAppend ctx1 (CtxCons ctx2 x) (CtxCons ctx x)}
\text{-- operations of HobbitLib}
\text{nu :: (Name a \rightarrow b) \rightarrow (Binding a b)}
\text{emptyMb :: a \rightarrow Mb CtxNil a}
\text{elimEmptyMb :: Mb CtxNil a \rightarrow a}
\text{combineMb :: Mb ctx1 (Mb ctx2 a) \rightarrow Mb (ctx1 ++: ctx2) a}
\text{separateMb :: IsAppend ctx1 ctx2 ctx \rightarrow Mb ctx b \rightarrow Mb ctx1 (Mb ctx2 b)}
\text{cmpName :: Name a \rightarrow Name b \rightarrow Maybe (a :=: b)}
\text{mbNameBoundP :: Mb ctx (Name a) \rightarrow Either (InCtx ctx a) (Name a)}
\text{mbToplevel :: SuperComb (a \rightarrow b) \rightarrow Mb ctx a \rightarrow Mb ctx b}
\end{verbatim}
\end{figure}
\)
ctx1 ++: ctx2. The inverse operation, $\text{separateMb}$, takes a multi-binding for $\text{ctx1} ++: \text{ctx2}$ and separates it into two nested multi-bindings for $\text{ctx1}$ and $\text{ctx2}$, respectively. For any given type context, however, there may be multiple ways to represent it as a context append $\text{ctx1} ++: \text{ctx2}$. The first argument to $\text{separateMb}$, therefore, both guarantee that the context can be divided in the requested way and also inform the Haskell type system which $\text{ctx1}$ and $\text{ctx2}$ are intended.

Names can be compared using the $\text{cmpName}$ operator, which takes any two names, possibly of different types, and compares them. If the names are equal, then $\text{cmpName}$ returns True, where True is a proof that the types of the two names are equal. This is because $\text{HOBBITLib}$ dictates that a name must have exactly one type, by the Typing property. If two names are not equal, then $\text{cmpName}$ returns the value Nothing.

The programmer can also test if a name inside a multi-binding is bound by that multi-binding using the $\text{mbNameBoundP}$ operation. This operation takes an input of type $\text{Mb ctx (Name a)}$ and returns either the name itself, if the name is not bound by the multi-binding, or a result of type $\text{Int a ctx a}$. Intuitively, the latter type is used because, if a name is bound by a multi-binding of names whose types are listed in the type context $\text{ctx}$, then its type must in $\text{ctx}$.

### 3.1 Operating under Bindings

The final operation listed in Figure 1 is $\text{mbToplevel}$, which allows functions to be applied inside bindings. This operation takes a function $f$ and a multi-binding $\nu(n_1, \ldots, n_k).e$ and returns the result of $\nu(n_1, \ldots, n_k).(f e)$. This operation is only safe, however, when $f$ has no free names; otherwise, the danger is that $f$ could contain some other binding for some $n_i$. This would violate Freshness, since the result would contain a binding for $n_i$ inside another binding for the same name. Stated differently, since $\text{HOBBITLib}$ cannot in general perform $\alpha$-conversion on bindings, it instead disallows cases where $\alpha$-conversion is required.

One sufficient condition for guaranteeing that the $f$ argument to $\text{mbToplevel}$ has no free names is to restrict it to super-combinators, i.e., expressions that are top-level, whose only free variables reference top-level definitions. Since the $\text{HOBBITLib}$ interface does not permit the top-level definition of a Name value (because of Scoping), this prevents $\text{mbToplevel}$ from violating Freshness.

To ensure that functions passed to $\text{mbToplevel}$ are super combinators, $\text{HOBBITLib}$ uses Haskell’s metaprogramming facility, Template Haskell [25]. Specifically, $\text{HOBBITLib}$ defines a type $\text{SuperComb}$ for representing super combinators, along with a function $\text{superComb}$ for creating elements of this type. The $\text{superComb}$ function takes a quoted Template Haskell expression and validates that the expression is indeed a super combinator, raising an error if not. In addition, $\text{HOBBITLib}$ leaves the $\text{SuperComb}$ type abstract, ensuring that any element of this type created by the user must be validated by $\text{superComb}$.

As an example, we demonstrate how to use $\text{superComb}$ and $\text{mbToplevel}$ to define a function that adds a binding to the inside of a multi-binding. This function is called $\text{mbAdd}$, as it intuitively lowers the body of a multi-binding into a context with more bindings, and can be defined as follows:

\[
\begin{align*}
\text{mbAdd} :: & \text{Mb ctx a} \rightarrow \text{Mb (CtxCons ctx b) a} \\
& \text{combineMb} = \text{mbToplevel \$ (superComb \[\text{Int} \text{nu\_const} \\text{Int}] \})
\end{align*}
\]

$\text{mbAdd}$ function works by first creating a quoted Template Haskell expression using the Template Haskell brackets ([ and ]), applying $\text{superComb}$ to validate the syntactic restrictions for super combinators, and then using the Template Haskell $\$$ operator to splice the wrapped quotation back in as the first argument of $\text{mbToplevel}$. Note that this pattern of usage ensures that $\text{superComb}$ is called at compile time, since it occurs inside the $\$ operator, and thus any errors raised by $\text{superComb}$ — because the argument is not a valid super combinator — are signaled before the code is run.

The quoted expression here composes $\text{const}$ and $\text{nu}$. The first of these, $\text{const}$, takes an expression $e$ and builds a function that returns $e$, ignoring its argument, while $\text{nu}$ then generates a fresh name and passes the name to the newly created function, thereby creating a binding whose body is $e$. The use of $\text{mbToplevel}$ then applies this function inside a multi-binding, creating a new binding inside the existing multi-binding of its argument. Finally, $\text{combineMb}$ combines these two bindings to create a single, unified multi-binding.

### 3.2 Pattern-Matching under Bindings

The most powerful facilities of $\text{HOBBITLib}$ are those allowing for pattern-matching on the contents of multi-bindings. Specifically, $\text{HOBBITLib}$ allows the user to match terms of type $\text{Mb ctx a}$ against patterns $\nu(n_1, \ldots, n_k).P$ where $\text{P}$ is a pattern of type $\text{a}$. To support this, we must solve two issues: we do not wish to expose the $\nu$ constructor to the programmer in order to ensure Freshness and Scoping, as discussed above; and, even without exposing the $\nu$ constructor directly, the most straightforward approach to pattern-matching under bindings can accidentally violate Scoping. We discuss these two problems and our solution in the remainder of this section.

To solve the first issue, that we do not wish to expose the $\nu$ constructor to the user, we use Haskell’s quasi-quotiation mechanism [12] to allow $\text{HOBBITLib}$ to automatically and safely generate Haskell patterns to match multi-bindings. If $\text{P}$ is a Haskell pattern, then the programmer can write $\text{[nuQQ | P |]}$ to match a multi-binding whose body matches $\text{P}$. We call such patterns $\nu$-patterns in the below. Internally, this syntax directs the Haskell compiler to call the quasi-quoter $\text{nuQQ}$, defined in the $\text{HOBBITLib}$ library, at compile time to generate the necessary pattern. As an example, the following defines a function $\text{mbBool}$ that intuitively lifts a boolean value out of a multi-binding:

\[
\begin{align*}
\text{mbBool} :: & \text{Mb ctx Bool} \rightarrow \text{Bool} \\
\text{mbBool} [\text{nuQQ True |}] = \text{True} \\
\text{mbBool} [\text{nuQQ False |}] = \text{False}
\end{align*}
\]

The first case matches $\nu(n_1, \ldots, n_k).\text{True}$ for any list of names $n_i$ through $n_k$. It then gives True, outside of the multi-binding, as the return value. Similarly, the second case matches a multi-binding containing False and returns False outside of the multi-binding. We use the term lifting function for functions like $\text{mbBool}$ which intuitively lift a value out of a multi-binding.

This approach is not possible for types that effectively have infinitely many constructors, such as the integers. For such types, $\text{HOBBITLib}$ exports special-purpose lifting functions; at present, these include the following:

\[
\begin{align*}
\text{mbInt} :: & \text{Mb ctx Int} \rightarrow \text{Int} \\
\text{mbChar} :: & \text{Mb ctx Char} \rightarrow \text{Char} \\
\text{mbString} :: & \text{Mb ctx String} \rightarrow \text{String}
\end{align*}
\]

Note that the last function, $\text{mbString}$, can in fact be defined by the user using $\text{mbChar}$ and pattern-matching over lists, but this operation is needed often enough that we include it in $\text{HOBBITLib}$. In addition, a user-defined $\text{mbString}$ function would in fact take time linear in the size of the string, while the internally defined version is constant time. These internally-defined lifting functions also behave slightly differently than user-defined lifting functions with regards to termination, as the latter require the body of a multi-binding to have a weak head normal form. This is not usually a problem in practice, however.

The second issue that arises in defining pattern-matching under bindings is that the most straightforward approach can lead to
violation of Scoping. This has to do with how variables are handled in \(\nu\)-patterns. Consider, for example, the following function:

\[
\text{unsafe :: Mb ctx (Name a) \to Name a}
\]

\[
\text{unsafe \ [\nuQQ \ | \ x \ | \ ] = x}
\]

Intuitively, this function matches any name inside a multi-binding, including names that are bound by the multi-binding itself, and removes that name from the multi-binding. Note that such a function can also be unsafe for types other than \(\text{Name a}\) such as the \(\text{Term}\) a type defined above that can contain names.

To address this issue, the pattern generated by the \([\nuQQ\] P \[\]\) quasiquotation implicitly “re-wraps” the multi-binding of the argument around all variables bound by \(P\). Thus pattern variables from \(\nu\)-patterns are never separated from the context where their names are guaranteed to be bound, and so Scoping is preserved. The \text{unsafe} function above is actually ill-typed; the type of \(x\) is again \(\text{Mb ctx (Name a)}\), so changing the type signature accordingly would essentially yield the identity function. The \(\nuQQ\) quasi-quoter is defined in Section 5.

As an example, the following computes the length of a list inside a multi-binding:

\[
\text{mbLen :: Mb ctx [a] \to Int}
\]

\[
\text{mbLen \ [\nuQQ \ | \ [] \ | \ ] = 0}
\]

\[
\text{mbLen \ [\nuQQ \ | \ x:1 \ | \ ] = 1 + mbLen x}
\]

The first case matches the empty list inside a multi-binding and returns \(0\). The second case matches a cons \(x:1\) inside a multi-binding and returns \(1\) plus the result of the recursive call on \(x\). Recall that \(1\) has type \(\text{Mb ctx [a]}\) on the right-hand side; the \(\nuQQ\) quasi-quoter has already re-wrapped the tail of the list in the multi-binding to preserve Scoping. Note again that allowing the tail of the list to escape the scope of its multi-binding could violate Scoping if, for example, it is a list of names, but this might not be known until run-time as the type of \text{mbLen} is polymorphic in \(a\).

3.3 Illustrative Examples

In this section, we illustrate the use of \text{HOBBITLIB} by giving two small examples using the \text{Term} type of simply-typed lambda-terms defined above. A larger example is given in Section 4. The examples are listed in Figure 2, along with some helper declarations which shall be described below. Include a function \text{eq} that tests equality of two terms and a function \text{subst} which performs capture-avoiding, type-safe substitution of a term into a binding. Note that both of these are defined using helper functions, \text{meq} and \text{msubst}, respectively, that operate on a term in a multi-binding. This pattern of defining a function on type \(a\) by defining a function over multi-bindings containing \(a\) is common in \text{HOBBITLIB}. At a high level, this approach is essentially avoiding \(\alpha\)-conversion by maintaining a context of all the bound names that have been seen during the traversal. Note that this is different from approaches such as Beluga [21] where all the bound names must be listed with the term; instead, in \text{HOBBITLIB}, there is always the possibility that a name is bound outside the current scope in which the current function is being called.

The \text{meq} function takes two terms in multi-bindings and tests if the terms are equal. Note that it does not require the two terms to have the same type or the multi-bindings to have the same type contexts, as this would make it much more complex to write the case for \text{App}. The first case matches terms that are both variables, i.e., that both use the \text{Var} constructor. It then uses \text{mbNameBoundP} to determine if both variables are bound in their respective multi-bindings: if so, then \text{inCtxSameLen} (whose code is omitted) is called on the \text{InCtx} proofs returned, to determine if both names were bound at the same place in the term; if both names are free then they are compared with \text{cmpeqBool} (whose code is omitted);

\[
\text{-- test if two InCtx proofs have}
\]

\[
\text{-- the same length (code omitted)}
\]

\[
\text{inCtxSameLen :: InCtx ctx1 a1 \to InCtx ctx2 a2 \to Bool}
\]

\[
\text{-- boolean version of cmpName (code omitted)}
\]

\[
\text{cmpeqBool :: Name a \to Name b \to Bool}
\]

\[
\text{-- equality under multi-bindings}
\]

\[
\text{meq :: (Mb ctx1 (Term a1)) \to (Mb ctx2 (Term a2)) \to Bool}
\]

\[
\text{meq \ [\nuQQ | \ Var bv1 | \ ] [\nuQQ | \ Var bv2 | \ ] =}
\]

\[
\text{case (mbNameBoundP bv1, mbNameBoundP bv2) of}
\]

\[
\text{(Left pl, Left p2) \to inCtxSameLen pl p2}
\]

\[
\text{(Right n1, Right n2) \to cmpNameBool n1 n2}
\]

\[
\text{\_ \_ = False}
\]

\[
\text{-- equality of terms}
\]

\[
\text{eq :: Term a \to Term b \to Bool}
\]

\[
\text{eq t u = meq (emptyMb t) (emptyMb u)}
\]

\[
\text{-- tuples whose types are indexed by ctx}
\]

\[
\text{data MapCtx f ctx where}
\]

\[
\text{Empty :: MapCtx f CtxNil}
\]

\[
\text{hs [] :: MapCtx f ctx \to f a \to MapCtx f (CtxCons ctx a)}
\]

\[
\text{-- tuple lookup (code omitted)}
\]

\[
\text{ctxLookup :: InCtx ctx a \to MapCtx f ctx \to f a}
\]

\[
\text{-- multi-arity substitution}
\]

\[
\text{msubst :: Mb ctx (Term a) \to MapCtx Term ctx \to Term a}
\]

\[
\text{msubst \ [\nuQQ | \ Var bn | \ ] ts =}
\]

\[
\text{case mbNameBoundP bn of}
\]

\[
\text{Left p \to ctxLookup p ts}
\]

\[
\text{Right n \to \_ \_}
\]

\[
\text{msubst \ [\nuQQ | \ App f a | \ ] ts =}
\]

\[
\text{App (msubst f ts) (msubst a ts)}
\]

\[
\text{msubst \ [\nuQQ | \ Lam b | \ ] ts =}
\]

\[
\text{Lam \ $ nu \$ \_ \_}
\]

\[
\text{msubst (combineMb b) (ts :> Var n) -- substituting a single term}
\]

\[
\text{subst :: Binding a (Term b) \to Term a \to Term b}
\]

\[
\text{subst b t = msubst (Empty :> t) b}
\]

Figure 2. Equality and Substitution in HOBBITLIB

otherwise, if one is bound and the other is free, they cannot be equal, and \text{False} is returned.

The second case of \text{meq} matches two applications, i.e., two terms in multi-bindings that both start with \text{App}. This case recurses on the two pairs of subterms, checking if the two functions and arguments are equal. The third case, when both terms start with \text{Lam}, is similar, except that arguments to \text{Lam} are themselves bindings, so the results of the \(\nu\)-patterns are bindings inside multi-bindings. Thus \text{combineMb} is called to combine the inner and outer bindings, resulting in a pair of terms inside larger multi-bindings, and the recursion then proceeds on the results. This case illustrates why type contexts are represented inside-out, as discussed in the beginning of Section 3, since \(\nu\)-patterns for binding constructs like \text{Lam} result in single bindings inside of multi-bindings. The final case matches terms with different constructors, in which case \text{False} is returned.
The eq function, for comparing terms that are not in multi-bindings, is then defined by first calling empty\texttt{mb} on both terms to put them inside empty multi-bindings, and then by calling meq on the results.

The second example in Figure 2 defines the function \texttt{absubst}, which performs multi-arity substitution. The first argument to \texttt{absubst} is a term of type Term a inside a multi-binding with context ctx. The second argument is a tuple containing one lambda-term for each type listed in ctx, i.e., containing a lambda-term of type b for each name of type b bound by the multi-binding in the first argument. Intuitively, these terms are being substituted for the bound names. Note that it is possible instead to define an \texttt{α} function, for comparing terms that are not in multi-bindings, using a tuple argument whose code is omitted) is called to return the term in the tuple argument that has the same type as the bound name; otherwise, the name is free and is removed from the multi-binding, and so is returned as the result. The second case of \texttt{mubst} for \texttt{App}, simply recurses on the two subterms.

The third case, for \texttt{Lam} recurses on the result of calling \texttt{combine\texttt{mb}} on the argument to \texttt{Lam}, as in meq. The result is put inside a new application of \texttt{Lam}, which uses \texttt{nu} to bind a new name \texttt{n}. Note that we cannot re-use the old name that was previously bound at this location, because of the possibility that one of the terms being substituted into the body of the \texttt{Lam} (i.e., one of the terms listed in the second argument) uses the same name. Thus this sort of re-use could potentially violate Freshness. The second argument, which is the tuple of terms to use for bound names, is then extended to contain the new bound name \texttt{n}. This explains why substitution is defined here as multi-arity substitution, in order to “freshen up” the bound names. Note that it is possible instead to define an \texttt{σ} conversion operation in a similar manner, using a tuple argument of type \texttt{MapCtx Name ctx} to contain the fresh names to be used in place of the old ones, but combining the operation of freshening up the names with the definition of substitution yields a definition of substitution that is quadratic instead of cubic. The function \texttt{subst}, which substitutes a single term into another, is then defined with a call to \texttt{absubst} using a one-element \texttt{MapCtx} tuple.

4. Extended Example: Lambda-Lifting

In this section, we demonstrate how HOBBITLIF can be used on a larger example, lambda-lifting (also known as closure conversion; see, e.g., [18]). Using HOBBITLIF, our approach defines lambda-lifting as a function over an adequate encoding of simply-typed lambda-terms with top-level declarations. This means that our lambda-lifting function is \textit{statically guaranteed} by the Haskell compiler to be type-preserving, meaning that it can only produce an output of the same type as the input. The code described here has been implemented, type-checked, and tested using GHC version 7.0.3, and takes 170 lines of non-comment, non-whitespace Haskell (not including tests, pretty-printing, and extensions). It can be downloaded with HOBBITLIF at http://www.cs.rice.edu/~emw4/hobbits.tgz

The goal of lambda-lifting is to transform a functional program so that all functions are top-level declarations, thus making it easier to compile. To do this, each lambda-abstraction that occurs in a term is “lifted” to be a top-level declaration. The process of lifting a lambda also abstracts over all variables that occur free in that lambda, since those variables will not be bound at the top level. The original occurrence of the lambda is then replaced by a call to this new top-level declaration, applying it to all the variables that were free in the original lambda.

As an example, consider the following term:

F_1 = \lambda g. \lambda x. g (g x)

To lambda-lift this term, we lift the inner lambda (the argument to \texttt{f}) to be a top-level declaration. Since this lambda has the variable \texttt{g} free, we must abstract over \texttt{g} when we lift it. The occurrence of the inner lambda then gets replaced by a call to the top-level declaration applied to the free variable \texttt{g}. The entire term is then lifted as well, since it is itself a lambda, and the result is as follows, where we use let for top-level declarations and use capital letters for the variables they bind:

let F_2 = \lambda g. \lambda x. g (g x)

Note that the declaration for \texttt{F_2}, which corresponds to the inner lambda, has an extra lambda-abstraction for the variable \texttt{g}. The original occurrence of the inner lambda has been replaced by a call to \texttt{F_2}, which is applied to the variable \texttt{g}.

Figure 3 defines two types derived from the \texttt{Term} type given in Section 2, one for the source terms to be lambda-lifted and one for the resulting terms. In these new types, bound names are separated into two sorts by their types: names of type D a (for some a) are bound by declarations, while those of type L a are bound by lambda-abstractions of the parameter list of a declaration. The source type, again called \texttt{Term}, merely adds the type L to the bindings and variable occurrences in the original \texttt{Term} type. Because lambda-lifting replaces all lambdas with top-level declarations, the result type, which we call \texttt{DTerm}, has no lambda constructor. It also has separate constructors \texttt{Var} and \texttt{DVar} for occurrences of lambda-bound and declaration-bound variables, respectively.

Figure 3 also defines both the type \texttt{Decl a}, which represents the non-empty parameter list and body of a top-level declaration, and the type \texttt{Decls a}, which represents a term of type a inside a list of zero or more top-level declarations. The \texttt{DeclBase} and \texttt{DeclCons} constructors re-use the \texttt{L} type, since each binding corresponds to a lambda. Both constructors’ types are analogous to that of the \texttt{Lam} constructor, with the important distinction that the \texttt{DeclBase} constructor maps a term inside a binding to a declaration. The \texttt{DeclBase} constructor represents a term with no declarations, while \texttt{DeclCons} adds a declaration of some type b. This constructor takes a term of type b, representing the value of the declaration, along with a \texttt{Decls} a inside a binding that binds a name for the
-- free variable lists and sub-context proofs
data MbLName ctx a where
    MbLName :: Mb ctx (Name (L a)) -> MbLName ctx (L a)
type FVList ctx fvs = MapCtx (MbLName ctx) fvs

-- the result type for fvUnion
data FVUnionRet ctx fvs1 fvs2 where
    FVUnionRet :: FVList ctx fvs ->
    SubCtx fvs1 fvs2 ->
    FVList ctx fvs1 fvs2

-- taking the union of two FVLists
fvUnion :: FVList ctx fvs1 -> FVList ctx fvs2 ->
    FVUnionRet ctx fvs1 fvs2

Figure 4. Lambda-Lifting: Free Variable Lists

declaration. The bound name has type D b to indicate it is a declar-

We present our approach to lambda-lifting in the remainder of this section. Section 4.1 first introduces a data type of lists of variables which occur free in terms, and briefly describes a subset relation and a union operation on these lists. These are needed to describe the free variables of the body of a lambda-abstraction in a statically-checkable manner. Section 4.2 next introduces a type of term skeletons, which are intermediate representations of the bodies of lambda-abstractions after lifting. Finally, Section 4.3 gives the main function \texttt{lambdaLifting}, which makes essential use of the continuation monad to move computations inside top-level declaration bindings.

4.1 Free Variable Lists

In this section, we explain briefly how our approach represents and operates on lists of free variables. Figure 4 defines the type FVList ctx fvs of lists of names, one for each of the types in the type context ctx (note that we use “type context” interchangeably with “list of types” in the below). Each of these has a type of the form L a, which we refer to as an L-type, and each name is inside a multi-binding for ctx. Intuitively, this type is used for the lambda-bound variables which occur free in a term being lambda-lifed. The context ctx lists the types of the names bound by lambdas which so far have been traversed by lambda-lifing. Note that nothing in this type requires the names to be bound in ctx, since lambda-

The figure also defines the type SubCtx ctx’ ctx, which intuitu-

Finally, Figure 4 declares the function \texttt{fvUnion} (code omitted) to take the union of two free variable lists. The result type states that there is some type context fvs such that the two input contexts fvs1 and fvs2 are sub-contexts of it. Because of the existential, stating this type requires introducing a new GADT, \texttt{FVUnionRet}.

4.2 Term Skeletons

In order to abstract over all the free variables in a term, our ap-

In particular, the example, closedness is only needed for the lambda-

Note that, although not being able to express closedness in \texttt{HOBBITLIB} is an issue, we do not feel that it is a major problem. In this particular example, closedness is only needed for the lambda-bound variables, and we still use \texttt{HOBBITLIB} names for declaration variables; this simplifies the definition of lambda-lifting, as it does not require lowering code to add declaration variables to the scope whenever a computation proceeds under a declaration bind-

Further, there are many situations where it is not especially problematic to choose the first option above and signal a runtime error when encountering non-closed names. This can in fact lead to simpler code, as the programmer need not structure the code in such a way that the type system can track closedness. In addition, that authors believe both that it is possible to extend \texttt{HOBBITLIB} to include a notion of closedness, and also that lambda-lifting can be rewritten to not even use an intermediate representation, thereby avoiding this problem altogether. These possibilities are discussed below in Section 8.

Figure 5 defines the type \texttt{STerm ctx a} of “skeletons” of the bodies of lambda-abstractions. These represent terms of type a with no lambda-abstractions, whose free (lambda-bound) variables are all listed in ctx. We use a typed \texttt{deBruijn} representation for lambda-bound variables to achieve this, representing each name
with an InCtx proof. We also include the constructor SWeaken, which takes a term in a sub-context of ctx and weakens it to a term in context ctx. Otherwise this type mirrors DTerm, without lambda-abstractions. Note that we still use HOBBITLIB names for declaration variables, yielding a simpler type than if typed deBruijn indices were used for these names as well.

Term skeletons can be converted back to the DTerm representation simply by substituting names for the free (lambda-bound) variables. This is done with the function skelSubst (code omitted), which is similar to the function nsub of Section 3.3.

The type FVSTerm ctx lctx a intuitively represents a list of free variables along with a term skeleton of type a using only the variables in this list. The free variables are given in two pieces: the first argument to the FVSTerm constructor is an FVList giving names in multi-bindings for ctx; but the skeleton can also contain names whose types are listed in lctx. Intuitively, this is because the skeleton is the result of lambda-lifting a sequence of nested lambda-abstractions that introduced variables for the types in lctx. Thus, when the skeleton is converted back to a DTerm, lambda-abstractions for these variables will be added unconditionally. The remaining variables will then be abstracted as well.

Finally, Figure 5 also declares the function fvSSepLVars (code omitted). This function intuitively takes a skeleton and its free variable list and separates out the last variables bound in the current context. This is represented at the type level as an operation on FVSTerm which moves the suffix of the ctx type argument to be the lctx type argument. This function is called just before the body of a lambda-abstraction is turned back into a lambda-abstraction, in order to abstract the free variables first and then form the lambda-abstractions found in the original term.

### 4.3 Lambda-Lifting in the Continuation Monad

We now come to the final result, which uses all the above to perform lambda-lifting. The key difficulty is in formulating the case for lambdas, which intuitively requires adding top-level declarations around the term currently being transformed. Stated differently, lambda-lifting proceeds by structural recursion on the body of a lambda-abstraction, but when it comes to another lambda, it must stop the current computation, add a top-level declaration for that lambda, and then resume the current computation inside the binding for the new top-level declaration. To do this, we use the continuation monad Cont along with Felleisen’s C operator [8]. The C operator allows us to capture (and abort) the current continuation, which represents a skeleton term being computed for the current point in the code. We can then add a top-level declaration, for the lambda being lifted, and resume the current computation inside that top-level declaration. This requires fixing the result type of the Cont operator to be Decls b for some b, so we can apply the DeclCons constructor outside the current continuation.

The code for lambda-lifting is given in Figure 6. It first defines the return type LLBodyRet b ctx a as a continuation monad with result type Decls a and whose return type is an unmapped term of type a in ctx. Next is an auxiliary definition of the type LCtx ctx of type contexts containing only types of the form L a, called L-contexts. The figure then defines the main function, LLBody, for lambda-lifting the bodies of lambdas. This function takes an auxiliary first argument, where it collects the types of the lambda-bound variables it sees as it enters inside the bodies of lambda-abstractions. The case for a normal variable returns that variable as bound variables it sees as it enters inside the bodies of lambda-abstractions. Finally, Felleisen’s C operator, which is called cont in the Cont monad, is invoked to move the current computation inside a new binding for a declaration variable. This new binding is created using the DeclCons constructor. The first argument to DeclCons, which is the declaration being bound, is created by calling the functions freeParams and boundTypes (type and code are omitted) to build a declaration with parameter bindings corresponding first to the free variables and then to the bound variables around the lambda-lifted body, and then by calling skelSubst to convert the body back to a DTerm using these lambda-abstracted variable. The second argument of DeclCons is then created by re-scooping the current continuation inside a call to nu, using the newly bound declaration variable d applied to the free variables used in the declaration as the result passed to this continuation.

Finally, Figure 6 defines lambdaLift. This function calls LLBody on the input term (using emptyMb to coerce it to a multi-binding with zero names) to create a monadic computation. It then runs the
resulting computation, passing a top-level continuation which does two things: it takes the free variable list returned by the computation and removes all the multi-bindings for the empty context using \(\text{elimEmptyMb}\); and it then uses this variable list to pass to \(\text{skelSubst}\), in order to convert the returned skeleton back to a DTerm.

5. Inside HOBBITLIB

We now describe the implementation of HOBBITLIB. The key declarations are given in Figure 7; the remaining operations are straightforward, and so are omitted for brevity. Internally, the type \(\text{Name}\) is represented as an integer, while the type \(\text{Mb ctx b}\) is a pair of a list of integers, corresponding to the names bound by the multi-binding, along with the body \(b\). To generate fresh names, we define a function \(\text{fresh\_name}\) which increments a global pointer, \(\text{counter}\), and returns the current value. The \(\text{nu}\) operation then calls \(\text{fresh\_name}\) to create a fresh name and then passes it to its argument \(\tau\). It then uses Haskell’s \(\text{seq}\) operator to ensure that the fresh name is evaluated strictly.

There is a little bit of trickery required to get \(\text{nu}\) to work correctly in the face of compiler optimizations, because of the uses of \(\text{unsafePerformIO}\). First, the \(\text{fresh\_name}\) function takes a dummy argument \(a\) and allocates a pointer to it. This is to make sure the monadic computation looks like it depends on \(a\), so that the compiler does not decide the monadic computation is a constant that can be lifted out of the body of \(\text{fresh\_name}\). Also, \(\text{counter}\) is marked with the \#INLINE pragma so the compiler cannot accidentally inline the code for creating \(\text{counter}\), causing a new pointer to be allocated for each use. The \(\text{fresh\_name}\) function does not require this pragma, however, because its monadic computation is intended to be performed each time it is called. Note that \(\text{nu}\) does allow common subexpression elimination (CSE) because, if two calls to \(\text{nu}\) use the same argument \(\tau\), then it is ok for them to use the same integer for their bound names, as we are ensuring \(\alpha\)-Equivalence.

The \(\text{cmpName}\) operation is relatively straightforward, using \(==\) to compare two names, represented by integers, for equality. If the names are equal, however, there is no guarantee from the Haskell type system that their associated types are equal. Since each name always has at most a single type, however, we know that the two types must be the same, and we thus call the Haskell cast function \(\text{unsafeCoerce}\) at the proof \(\text{Refl}\) to a proof of equality of the types associated with the names. The \(\text{mbNameBoundP}\) function compares a name with the names bound in a multi-binding using \(\text{elemIndex}\) to get the position of the name in the list. If the name is not in the list we know it can safely be removed from the multi-binding, so it is returned, otherwise \(\text{unsafeLookupCtxt}\) is called (code omitted), which contains a similar call to \(\text{unsafeCoerce}\) to cast an \(\text{InCtxt}\) proof to the right type.

As discussed in Section 3.1, the \(\text{mbToplevel}\) operator uses the type \(\text{SuperComb}\) a type to ensure safety, by only allowing super combinators to be applied under multi-bindings. This type is isomorphic to \(a\), with the \(\text{SuperComb}\) constructor applied, and \(\text{mbToplevel}\) simply removes this constructor from its first input and applies the resulting function to the body of the \(\nu\)-abstraction in the second input. To ensure that the elements of this type are always super combinators, HOBBITLIB hides this data type (as with \(\text{name}\) and \(\text{mb}\)), thus only allowing super combinators to be created by the \(\text{superComb}\) function. This function creates a monadic computation, inside Template Haskell’s \(\text{Q}\) monad, which first calls the \(\text{isSuperComb}\) function to determine if the argument is in fact a super combinator. The code for this latter function is omitted, but it essentially traverses the input and determines that all Haskell variables that occur are either global or are bound inside the input term. (This is done by checking that all constructors for the Template Haskell \(\text{Name}\) type are either \(\text{NameG}\) or \(\text{NameU}\).) If this check passes, then the \(\text{superComb}\) function simply wraps the input expression in an application of the (hidden) \(\text{SuperComb}\) constructor; otherwise a compile-time error is signaled.

To implement pattern-matching under bindings, the quasi-quoter \(\text{nuQQ}\) contains a slightly simplified parser for Haskell patterns, transforming them as follows. First, the whole pattern is wrapped in an application of the \(\text{Mb}\) constructor applied to a fresh variable name, say \(\text{name}\). Then each variable \(x\) in the pattern is replaced by the \(\text{view\_pattern}\) \(\text{(Mb\_names \(\to\) \(x\))}\). Recall that a view pattern \((f \to P)\) first applies \(f\) to its argument and then matches against \(P\). In our case, the argument to each of the view patterns is the corresponding subterm of the entire \(\nu\)-pattern’s argument. Thus \(\text{nuQQ}\) generates a pattern that distributes the multi-binding context of its argument’s constructor into the subterms. The user cannot directly refer to any of the subterms outside of their multi-bindings, since such values are never even bound to a name. Furthermore, each subterm’s multi-binding is also constrained to have the same phantom type argument as the multi-binding being matched.

---

**Figure 7. Implementation of HOBBITLIB**
In this section, we argue that the multi-bindings implemented by HOBBITLIB satisfy Freshness, \(\alpha\)-Equivalence, Scoping, and Typing. These properties ensure in turn that the user cannot observe the imperative actions of \(\alpha\) other than the fact that different bindings yield unequal names (as required by Freshness). The latter holds by Typing, since the only unsafe casts in the HOBBITLIB implementation are used to prove that two types for the same name are equal.

To demonstrate that these properties hold, we first introduce the \(\alpha\)-well-formed terms, an extension of Scoping that is needed to prove \(\alpha\)-Equivalence. We then prove two lemmas, that the \(\alpha\)-well-formed terms are closed under reduction and that \(\alpha\)-equivalence of \(\alpha\)-well-formed terms is a bisimulation for reduction; we use a call-by-need notion of reduction to model Haskell's lazy evaluation. Scoping and \(\alpha\)-Equivalence follow immediately. Freshness is a consequence of these lemmas because, although it is possible to reduce to a term that uses the same name for nested bindings, this cannot be observed by a program. This is because any reduction on such a term is equivalent to a reduction on an \(\alpha\)-equivalent term where the inner binding uses a different name. The fourth property, Typing, then follows by a straightforward proof that the typing judgment is also preserved under \(\alpha\)-equivalence. For reasons of brevity, however, we do not give a type system nor prove this fact here but it is a straightforward proof using standard techniques. We also omit proofs of lemmas for brevity, but these are all by straightforward induction on terms.

We begin by formalizing an operational semantics for Haskell with HOBBITLIB, given in Figure 8. This is based on the call-by-need semantics of Ariola et al. [1]. The top half of the figure defines five syntactic classes: the operators \(\text{op}\), the terms \(M\), the values \(V\), the let-contexts \(L\), and the evaluation contexts \(E\). The operators include (abbreviated names for) the \(\text{nu}\), \(\text{cmp}\), \(\text{cmp}\), \(\text{bound}\), and \(\text{bound}\) operators of HOBBITLIB, along with a case construct for pattern-matching. The top operator always includes a top-level function \(F\), which is simply a closed term (having no free names or variables), while the case operator is always annotated with a list of constructors over which it matches. (Normally the constructors are determined by the type of the scrutinee, but we are avoiding giving a type system here, for brevity.) Note that, for brevity, we are actually formalizing only a simplified version of HOBBITLIB with only single bindings instead of multi-bindings; the results generalize easily to the case with multi-bindings. Thus we do not include the operators related to multi-bindings.

The terms include variables \(x\), constructors \(c\), names \(n\), operators, applications \(M\ M\), \(\lambda\)-abstractions \(\lambda x.\ M\), \(\nu\)-abstractions \(\nu n.\ M\), and let-expressions. The let-expressions are included to model lazy environments, where the value for a variable \(x\), bound by a let, is only substituted for \(x\) when the value of \(x\) is needed. This is discussed in more detail by Ariola et al. [1]. Terms are considered equal up to renaming of bound variables \(x\) but not of bound names \(n\), just as in the actual implementation of HOBBITLIB. We write this equality as \(\equiv\). We also write \(\equiv\) for equality up to renaming of bound variables and names, where “\(\alpha\)-equivalence” in the below (without the capital “E”) refers to this latter relation. The values include the terms that represent valid results of computations.

The remaining two syntactic classes define two classes of term contexts. These represent terms with a single hole, written \([\_]\). If \(C\) is a term context, we write \(C[M]\) for the result of replacing (in a non-capture-avoiding manner) the hole in \(C\) with \(M\). The let-contexts \(L\) are holes in the bodies of zero or more let-expressions. These essentially represent environments. The evaluation contexts are used to define where evaluation is allowed to take place: at the top of a term; to the left of an application; in the value bound by a \(\nu\)-abstraction; in the value bound by a let-expression when that value is needed in the body; in the argument of an operator, or the second argument of the two-argument operator \(\text{cmp}\); or inside a \(\nu\)-abstraction that is the argument to \(\text{bound}\) or \(\text{case}\). We also define the answers as the set of terms \(L[V]\) of values in the bodies of zero or more let-expressions. We write \(A\) for answers.

The bottom half of Figure 8 give reduction rules for evaluating terms. These are split into: the first rule, which closes reduction under evaluation contexts; the rules for operators; and the rules for functions and environments given by Ariola et al. [1]. Most of the rules for operators are straightforward. \(\text{nu}\) creates a binding with a fresh name. \(\text{cmp}\) returns \(\text{True}\) or \(\text{False}\) (which are constructors that are assumed to exist), where we are again simplifying HOBBITLIB in a non-essential way by omitting type equality proofs. Similarly, \(\text{bound}\) returns either \(\text{Nothing}\) or \(\text{Just}\) \(n\) (for \(n \neq n'\)). The \(\text{top}\) rule for \(\text{op}\) is not needed since \(\text{op}\) is a context-free operator.

Note that case can also match constructors inside \(\nu\)-abstractions, so again we include the let-context \(L\) inside the \(\nu\)-abstraction in the argument.

The last rules in the figure turn an application into a body in an environment, substitute a bound value in an environment for a variable when the value is needed, and commute environments upwards in a term as needed. The interested reader can consult the work of Ariola et al. [1] for more discussion of these rules.

We implicitly assume in the below that all occurrences of operators in terms are \(\text{fully applied}\), meaning that they occur, possibly inside an environment, applied to an argument; i.e., they have the form \(L[\text{op}] M\) for some \(M\). The \(\text{cmp}\) operator is binary, so must have two arguments, meaning occurrences have the form \(L_1[\text{cmp}] M_1\) \(M_2\). We also define the \(\text{normal forms}\) as the terms that do not reduce, and the \(\text{ill-formed terms}\) as those that contain an application of a name or a \(\nu\)-abstraction. We can then prove
the following two lemmas, which state that our semantics behaves how we want it to and that it does not have too few or two many cases.

**Lemma 1 (Normal Forms).** Any closed, fully applied, normal form \( M \) is either an answer or is an ill-formed term.

**Lemma 2 (Determinism).** For any \( M \), there is at most one \( M' \) (up to \( \equiv \)) such that \( M \rightarrow M' \).

We would like to prove that reduction is insensitive to \( \alpha \)-equivalence, as discussed above. The one rule that is problematic, however, is substitution (the second rule from Ariola et al.), as substitution is not capture-avoiding for names. As an example, consider the following term:

\[
\text{let } x = n \text{ in } \text{boundP } (\nu n. x)
\]

Reducing this term twice substitutes \( n \) for \( x \) and then reduces the application of \( \text{boundP} \) to \( \text{True} \). If we \( \alpha \)-convert the \( \nu \)-abstraction, however, the term reduces to \( \text{False} \). To avoid this situation, we must ensure that no substitution can ever cause a name to be substituted under a binder for it. In order to prevent this, however, we must also preclude \( \nu \)-abstractions from occurring inside \( \lambda \)-abstractions, or we could have the term

\[
(\lambda x. \text{boundP } (\nu n. x)) n
\]

which reduces to the above in one step. Thus we make the following definition:

**Definition 1.** \( M \) is \( \alpha \)-well-formed, written \( \WF_\alpha(M) \), iff:

1. All variables \( x \) and names \( n \) occur bound;
2. No \( \nu \)-abstractions occur inside \( \lambda \)-abstractions;
3. If we fully substitute \( M \) then we do not substitute a name \( n \) into a \( \nu \)-abstraction for \( n \), where fully substituting \( M \) means repeatedly replacing occurrences of \( x \) by \( N \) in the bodies of

   \[
   \text{let } x = N \text{ bindings}
   \]

Using this definition, we can now prove the main lemmas of this section:

**Lemma 3 (Preservation of \( \WF_\alpha(\cdot) \)).** If \( M \rightarrow M' \) for \( \WF_\alpha(M) \) then \( \WF_\alpha(M') \).

**Lemma 4 (Bisimulation for \( \WF_\alpha(\cdot) \)).** If \( M \equiv N \) for \( \WF_\alpha(M) \) and \( \WF_\alpha(N) \), and if \( M \rightarrow M' \) then \( N \rightarrow N' \) for some \( N' \equiv M' \).

7. Related Work

There are a number of approaches to representing name-bindings in the literature. The oldest approach, deBruijn indices [6], represents variables as the number of binders between the occurrence and the binding of the variable. A key benefit of deBruijn indices is that they are conceptually simple. Operations like substitution that move terms inside other bindings, however, require subtle index math to update the number used by bound variables. This index math is often a source of bugs, as it is difficult to get right. Although it is possible to extend deBruijn indices to associate type information with bound names [2, 26], this approach becomes an encoding of “term of type \( A \) in context \( \Gamma \)” instead of just “term of type \( A \)”. This approach requires heavy use of functions to manipulate the context \( \Gamma \), which effectively perform the same index math as untyped deBruijn indices but on proof objects. This makes programs that manipulate bindings more awkward to write.

The Locally Nameless approach of McBride and McKinna [13] uses deBruijn indices only for bound names, using a different type (such as strings) for free variables. McBride and McKinna show that the index math required by deBruijn indices can be limited to two operations, abstract and instantiate, for Locally Nameless representations, thereby greatly easing the burden of the programmer. The authors do not know of any research that has associated types with names under the Locally Nameless approach, but such an approach would either be similar to typed deBruijn indices, requiring a context \( \Gamma \) for the free variables of a term, or it would not constrain the free variables of a term in any way.

There has been much research into representations based on Higher-Order Abstract Syntax (HOAS). This terminology, coined by Pfennig and Elliott [19], refers to the use of functions in the host language to represent name-bindings. The benefits of HOAS are that functional languages already ensure \( \alpha \)-Equivalence, Scoping, and Typing for functions, and so these properties are “for free” in HOAS. Substitution is also for free, using function application in the host language. We refer to the main drawbacks of HOAS, that variables cannot be compared. The other main drawback is that HOAS does not give Freshness, because variables in the host language cannot be considered unequal. This leads to one of the main drawbacks of HOAS, that variables cannot be compared. The other main drawback is that it is impossible to pattern-match under host language functions, since the body of a function can depend on its input.

A number of techniques have been investigated for getting around the limitations of HOAS. Meijer and Hutton [14] showed how to define operations over HOAS while simultaneously defining their inverses. Fegaras and Sheard [7] showed that an inverse is not needed if name-binding functions are guaranteed to be parametric (meaning that they do not examine their arguments) and if a free variable constructor is added to datatypes being examined. Washburn and Weirich [29] built on this approach, showing how to use parametric polymorphism to ensure that functions meant to represent name-bindings are parametric. This is difficult to use in practice, however, because it requires operations on HOAS encodings to have complex, higher-order types. Further, these approaches can only write operations which are folds, which cannot express operations like equality tests or getting the free variables of a term.

A number of approaches have also shown how to add support for HOAS into the host language itself. These include \( \alpha \)-Prolog [15], Twelf [20], the modal calculus of Schürmann et al. [24], Bedwyr [3], Beluga [21], and Delphin [23]. These are all special-purpose languages, however, and it is not clear how to use the approaches in more mainstream languages like Haskell.

8. Conclusion

In this paper, we have described a novel library, HOBBITLIB, for higher-order encodings. Higher-order encodings make it possible to define adequate encodings of typed programming languages, thus eliminating whole classes of bugs in compilers, interpreters, and other language processing tools. Specifically, functions over adequate encodings of typed programming languages are statically guaranteed to preserve well-typedness of the programs being manipulated. HOBBITLIB represents name-bindings as pairs of the name being bound and the body of the binding, allowing pattern-matching on the bodies of bindings to be easily defined in the host language. In order to ensure Freshness, \( \alpha \)-Equivalence, Scoping, and Typing, HOBBITLIB hides this representation, exposing a set of combinators to allow the user to manipulate name-bindings. We then show how HOBBITLIB can be used for more complex examples, such as lambda-lifting.

A natural question to ask at this point is: how much farther can we push HOBBITLIB? There are a number of things it still cannot do. As one example, the CNIC language (which was the motivation for HOBBITLIB) contains an elimination form for name-bindings. This elimination form allows a user to remove a name-binding from around its body by supplying a fresh name to replace the bound name. Ensuring that the user-supplied name is fresh seems
to require support in the Haskell type system that cannot easily be described by a library.

Another useful feature of HOBBITLib would be to add support for closedness, as discussed in Section 4.2. Since functions can be applied inside name-bindings, their inputs could always potentially have free variables; this is captured by the type of \( \text{mbNameBoundP} \), which always has the possibility that a name can be removed from a multi-binding because it is not bound there. It would be useful to have a closedness type, to indicate statically that a value does not have free names. This would make functions that cannot handle free names, such as an interpreter (which does not know what to do with a free variable), a little nicer to write; currently, such functions must include a case for free variables which simply reports an error. Further, closedness types would allow imperative operations to occur inside name-bindings, if we require all writes to pointers to store only closed values to prevent scope extrusion [4]. The fact that imperative operations cannot occur inside name-bindings in the current HOBBITLib can be expressed by the fact that the `10` and `19` type constructors do not commute. It is well known that modal type systems are useful for expressing closedness, but this would also require special support from the Haskell type system. A different approach might be to use environment classifiers [27], which would not require any modifications to Haskell.

Acknowledgments

This work was supported in part by NSF award CCF-0964520. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation. The authors would like to acknowledge the PL group at KU for lots of great feedback after reading a draft of this paper in their reading group. We would also like to thank Stephanie Weirich for helpful discussions and Simon Peyton-Jones for suggesting the lambda-lifting example.

References


