Names For Free — Polymorphic Views of Names and Binders

Jean-Philippe Bernardy
Chalmers University of Technology and University of Gothenburg
bernardy@chalmers.se

Nicolas Pouillard
IT University Copenhagen
npou@itu.dk

Abstract

We propose a novel technique to represent names and binders in Haskell. The dynamic (run-time) representation is based on de Bruijn indices, but it features an interface to write and manipulate variables conviently, using Haskell-level lambdas and variables. The key idea is to use rich types: a subterm with an additional free variable is viewed either as $\forall v.v \to \mathsf{Term}(a + v)$ or $\exists v.v \times \mathsf{Term}(a+v)$ depending on weather it is constructed or analysed. We demonstrate on a number of examples how this approach approach permits to express terms construction and manipulation in a natural way, while retaining the good properties of representations based on de Bruijn indices.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]

Keywords name binding, polymorphism, parametricity, type-classes, nested types

1. Introduction

One of the main application areas of functional programming languages such as HASKELL is programming language technology. In particular, HASKELL programmers often find themselves manipulating data structures representing some higher-order object languages, featuring binders and names.

Yet, the most commonly used representations for names and binders yield code which is difficult to read, or error-prone to write and maintain. The techniques in question are often referred as "nominal", "de Bruijn indices" and "Higher-Order Abstract Syntax (HOAS)".

In the nominal approach, one typically uses some atomic type to represent names. Because a name is simply referred to the atom representing it, the nominal style is natural. The main issues with this technique are that variables must sometimes be renamed in order to avoid name capture (that is, if a binder refers to an already used name, variables might end up referring to the wrong binder). The need for renaming demands a way to generate fresh atoms. This side effect can be resolved with a supply for unique atoms or using an abstraction such as a monad but is disturbing if one wishes to write functional code. Additionally, nominal representations are not canonical. (For instance, two α-equivalent representations of the same term such as `λx.x` and `λy.y` may be different). Hence special care has to be taken to prevent user code to violate the abstraction barrier. Furthermore fresh name generation is an observable effect breaking referential transparency (`fresh x in x ≢ fresh x in x`). For instance a function generating fresh names and not properly using them to close abstractions becomes impure.

To avoid the problem of name capture, one can represent names canonically, for example by the number of binders, typically λ, to cross between an occurrence and its binding site (a de Bruijn index). This has the added benefit of making α-equivalent terms syntactically equal. In practice however, this representation makes it hard to manipulate terms: instead of calling things by name, programmers have to rely on their arithmetic abilities, which turns out to be error-prone. As soon as one has to deal with more than just a couple open bindings, it becomes easy to make mistakes.

Finally, one can use the binders of the host language (in our case HASKELL) to represent binders of the object language. This technique (called HOAS) does not suffer from name-capture problems nor does it involve arithmetic. However the presence of functions in the term representation mean that it is difficult to manipulate, and it may contain values which do not represent any term.

The contribution of this paper is a new programming interface for binders, which provides the ability to write terms in a natural style close to concrete syntax. We can for example build the application function of the untyped λ-calculus as follows:

```
-- Building the following term: λ f x → f x
apTm = lam $ λ f → lam $ λ x → var f ‘App‘ var x
```

and we are able to test if a term is eta-contractible using the following function:

```
canEta (Lam e) = unpack e $ λ x t → case t of
  App e1 (Var y) → y ‘isOccurenceOf‘ x &&
                   x ‘freshFor‘ e1
  _ → False
canEta _ = False
```

All the while, neither do we require a name supply, nor is there a risk for name capture. Testing terms for α-equivalence remains straightforward and representable terms are exactly those intended. The cost of this achievement is the use of somewhat more involved types for binders, and the use of extensions of the HASKELL type-system. The new construction is informally described and motivated in sec. 2. In sections 3 to 5 we present in detail the implementation of the technique as well as basic applications. Larger applications (normalization using hereditary substitutions, closure conversion and CPS transformation) are presented in sec. 6.

2. Overview

In this section we describe our interface, but before doing so we describe a simple implementation which can support this interface.

2.1 de Bruijn Indices

De Bruijn (1972) proposed to represent an occurrence of some variable `x` by counting the number of binders that one has to cross between the occurrence and the binding site of `x`. A direct implementation of the idea may yield the following representation of untyped λ-terms:

```
data Nat = O | S Nat
data TmB where
  VarB :: Nat → TmB
  AppB :: TmB → TmB → TmB
  LamB :: TmB → TmB
```

Using this representation, the implementation of the application function `λ f x → f x` is the following:

```
apB :: TmB
apB = LamB $ LamB $ VarB (S O) ‘AppB‘ VarB O
```

However, such a direct implementation is cumbersome and naïve. For instance it cannot statically distinguish bound and free variables. That is, a closed term has the same type as an open term.

Nested Abstract Syntax In functional programming languages such as HASKELL, it is possible to remedy to this situation by using nested data types and polymorphic recursion. That is, one parameterizes the type of terms by a type that can represent *free* variables. If the parameter is the empty type, terms are closed. If the parameter is the unit type, there is at most one free variable, etc. This representation is known as Nested Abstract Syntax (Bellegarde and Hook 1994; Bird and Paterson 1999; Altenkirch and Reus 1999).

```
data Tm a where
  Var :: a → Tm a
  App :: Tm a → Tm a → Tm a
  Lam :: Tm (Succ a) → Tm a
```

The recursive case `Lam` changes the type parameter, increasing its cardinality by one, since the body can refer to one more variable. Anticipating the amendments we propose, we define the type `Succ a` as a proper sum of `a` and the unit type `()` instead of `Maybe a` as customary. Because the sum is used in an asymmetric fashion (the left-hand-side corresponds to variables bound earlier and the right-hand-side to the freshly bound one), we give a special definition of sum written ▷, whose syntax reflects the intended semantics.

```
type Succ a = a ▷ ()

data a ▷ v = Old a | New v

bimap :: (a → a’) → (v → v’) →
         (a ▷ v) → (a’ ▷ v’)
bimap f _ (Old x) = Old (f x)
bimap _ g (New x) = New (g x)

untag :: a ▷ a → a
untag (Old x) = x
untag (New x) = x
```

Using the `Tm` representation, the implementation of the application function `λ f x → f x` is the following:

```
apNested :: Tm Zero
apNested = Lam $ Lam $ Var (Old $ New ())
                 ‘App‘ Var (New ())
```

As promised, the type is explicit about `apNested` being a closed term: this is ensured by using the empty type `Zero` as an argument to `Tm`.

```
data Zero -- no constructors
```

In passing, we remark that another type which faithfully captures closed terms is `∀ a. Tm a` — literally: the type of terms which are meaningful in any context. Indeed, because `a` is universally quantified, there is no way to construct an inhabitant of it; therefore one cannot possibly refer to any free variable. In particular one can instantiate `a` to be the type `Zero`.

However the main drawback of using de Bruijn indices remains: one must still count the number of binders between the declaration of a variable and its occurrences.

2.2 Referring to Bound Variables by Name

To address the issues just touched upon, we propose to build λ-abstractions with a function called `lam`. What matters the most is its type:

```
lam :: (∀ v. v → Tm (a ▷ v)) → Tm a
lam f = Lam (f ())
```

That is, instead of adding a concrete unique type (namely `()`) in the recursive parameter of `Tm`, we quantify universally over a type variable `v` and add this type variable to the type of free variables. The body of the lambda-abstraction receives an arbitrary value of type `v`, to be used at occurrences of the variable bound by `lam`.

The application function is then built as follows:

```
apTm_ :: Tm Zero
apTm_ = lam $ λ f → lam $ λ x → Var (Old (New f))
                           ‘App‘ Var (New x)
```

By unfolding the definition of `lam` in `apTm_` one recovers the definition of `apNested`.

Safety Using our approach, the binding structure, which can be identified as the *specification*, is written using the host language binders. However at variable occurrences, de Bruijn indices are still present in the form of the constructors `New` and `Old`, and are purely part of the *implementation*.

The type-checker then makes sure that the implementation matches the specification: for example if one now makes a mistake and forgets one `Old` when entering the term, the HASKELL type system rejects the definition.

```
oops_ = lam $ λ f → lam $ λ x → Var (New f)
                          ‘App‘ Var (New x)
-- Couldn’t match expected type ‘v1’
--             with actual type ‘v’
```

In fact, if all variables are introduced with the `lam` combinator the possibility of making a mistake in the *implementation* is nonexistent, if we ignore obviously diverging terms. Indeed, because the type `v` corresponding to a bound variable is universally quantified, the only way to construct a value of its type is to use the variable bound by `lam`. (In HASKELL one can use a diverging program;

however one has to make a conscious decision to produce a value of such an obviously empty type.)

In general, in a closed context, if one considers the expression `Var ((Old)ⁿ (New x))`, only one possible value of `n` is admissible. Indeed, anywhere in the formation of a term using `lam`, the type of variables is $a = a_0 \rhd v_0 \rhd v_1 \rhd \cdots \rhd v_n$ where $v_0, v_1, \ldots, v_n$are all distinct and universally quantified, and none of them occurs as part of a_0. Hence, there is only one injection function from a given v_i to `a`.

Auto-Inject Knowing that the injection functions are uniquely determined by their type, one may wish to infer them mechanically. Thanks the powerful instance search mechanism implemented in GHC, this is feasible. To this effect, we define a class `v ∈ a` capturing that `v` occurs as part of a context `a`:

```
class v ∈ a where
  inj :: v → a
```

We can then wrap the injection function and `Var` in a convenient package:

```
var :: ∀ v a. (v ∈ a) ⇒ v → Tm a
var = Var . inj
```

and the application function can be conveniently written:

```
-- Building the following term: λ f x → f x
apTm = lam $ λ f → lam $ λ x → var f ‘App‘ var x
```

In a nutshell, our de Bruijn indices are typed with the context where they are valid. If that context is sufficiently polymorphic, they can not be mistakenly used in a wrong context. Another intuition is that `New` and `Old` are building proofs of "context membership". Thus, when a de Bruijn index is given a maximally polymorphic context, it is similar to a well-scoped name.

So far, we have seen that by taking advantage of polymorphism, our interface allows to construct terms with de Bruijn indices, combined with the safety and convenience of named variables. In the next section we show how to use the same idea to provide the same advantages for the analysis and manipulation of terms.

2.3 Referring to Free Variables by Name

Often, one wants to be able to check if an occurrence of a variable is a reference to some previously bound variable. With de Bruijn indices, one must (yet again) count the number of binders traversed between the variable bindings and its potential occurrences — an error prone task. Here as well, we can take advantage of polymorphism to ensure that no mistake happens. We provide a combinator `unpack`, which hides the type of the newly bound variables (the type `()`) as an existentially quantified type `v`. The combinator `unpack` takes a binding structure (of type `Tm (Succ a)`) and gives a pair of a value `x` of type `v` and a sub-term of type `Tm (a ▷ v)`. Here we represent the existential using continuation-passing style instead of a data-type, as it appears more convenient to use this way. Because this combinator is not specific to our type `Tm` we generalize it to any type constructor `f`:

```
unpack :: f (Succ a) →
          (∀ v. v → f (a ▷ v) → r) → r
unpack e k = k () e
```

Because `v` is existentially bound, `x` can never be used in a computation. It only acts as a reference to a variable in a context, in a way which is only accessible to the type-checker. For instance, when facing a term `t` of type $Tm\ (a \rhd v_0 \rhd v_1 \rhd v)$, `x` refers to the last introduced free variable in `t`. Using `unpack`, one can write a function which can recognize an eta-contractible term as follows: (Recall that an a eta-contractible term has the form `λ x → e x`, where `x` does not occur free in `e`.)

```
canEta :: Tm Zero → Bool
canEta (Lam e) = unpack e $ λ x t → case t of
  App e1 (Var y) → y ‘isOccurenceOf‘ x &&
                   x ‘freshFor‘ e1
  _ → False
canEta _ = False
```

In the above example, the two functions `isOccurenceOf` and `freshFor` use the `inj` function to lift `x` to a reference in the right context before comparing it to the occurrences. The calls to these functions do not get more complicated in the presence of multiple binders. For example, the code which recognizes the pattern `λ x y → e x` is as follows:

```
recognize :: Tm Zero → Bool
recognize t0 = case t0 of
    Lam f → unpack f $ λ x t1 → case t1 of
      Lam g → unpack g $ λ y t2 → case t2 of
        App e1 (Var y) → y ‘isOccurenceOf‘ x &&
                         x ‘freshFor‘ e1
        _ → False
      _ → False
    _ → False
```

Again, even though variables are represented by mere indices, the use of polymorphism allows the user to refer to them by name, using the instance search mechanism to fill in the details of implementation.

Pack It is easy to invert the job of `unpack`. Indeed, given a value `x` of type `v` and a term of type `Tm (a ▷ v)` one can reconstruct a binder as follows:

```
pack :: Functor tm ⇒ v → tm (a ▷ v) → tm (Succ a)
pack x = fmap (bimap id (const ()))
```

(The `Functor` constraint is harmless, as we will see in sec. 4.) As we can see, the value `x` is not used by pack. However it statically helps as a specification of the user intention: it makes sure the programmer relies on host-level variable names, and not indices.

A production-quality version of `pack` would allow to bind any free variable. Writing the constraint `Insert v a b` to mean that by removing the variable `v` from the context `b` one obtains `a`, then a generic `pack` would have the following type:

```
packGen :: ∀ f v a b w. (Functor f, Insert v a b) ⇒
           v → f b → (w → f (a ▷ w))
```

The implementation of `packGen` and `Insert` is a straightforward extension of `inj` and `(∈)`, but it does not fit here, so we defer it to the appendix.

In sum, the `pack` combinator makes it possible to give a nominal-style interface to binders. For example an alternative way to build the `Lam` constructor is the following:

```
lamP :: v → Tm (a ▷ v) → Tm a
lamP x t = Lam (pack x t)
```

3. Contexts

Having introduced our interface informally, we now begin a systematic description of is realization and the concepts it builds upon.

We have seen that the type of free variables essentially describes the context where they are meaningful. A context can either be empty (and we represent it by the type `Zero`) or not (which we can represent by the type `a ▷ v`).

An important function of the `v` type variable is to make sure programmers refer to the variable they intend to. For example, consider the following function, which takes a list of (free) variables and removes one of them from the list. It takes a list of variables in the context `a ▷ v` and returns a list in the context `a`. For extra safety, it also takes the name of the variable being removed, which is used only for type-checking purposes.

```
remove :: v → [a ▷ v] → [a]
remove _ xs = [x | Old x ← xs]
```

The function which computes the list of occurrences of free variables in a term can be directly transcribed from its nominal-style definition, thanks to the `unpack` combinator.

```
freeVars :: Tm a → [a]
freeVars (Var x) = [x]
freeVars (Lam b) = unpack b $ λ x t →
   remove x (freeVars t)
freeVars (App f a) = freeVars f ++ freeVars a
```

3.1 Names Are Polymorphic Indices

Checking whether two names are equal or not is necessary to implement a large class of term manipulation functions. To implement comparison between names, we provide the following two `Eq` instances. First, the `Zero` type is vacuously equipped with equality:

```
instance Eq Zero where
  (==) = magic

magic :: Zero → a
magic _ = error "impossible"
```

Second, if two indices refer to the first variable they are equal; otherwise we recurse. We stress that this equality inspects only the *indices*, not the values contained in the type. For example `New 0 == New 1` is `True`:

```
instance Eq a ⇒ Eq (a ▷ v) where
  New _ == New _ = True
  Old x == Old y = x == y
  _     == _     = False
```

Comparing naked de Bruijn indices for equality is an error prone operation, because one index might be valid in a context different from the other, and thus an arbitrary adjustment might be required. With Nested Abstract Syntax, the situation improves: by requiring equality to be performed between indices of the same type, a whole class of errors are prevented by type-checking. Some mistakes are possible though: given a index of type `a ▷ () ▷ ()`, a swap the last two variables might be the right thing to do, but one cannot decide if it is so from the types only. By making the contexts fully polymorphic as we propose, no mistake is possible. Hence the slogan: names are polymorphic indices.

Consequently, the derived equality instance of `Tm` gives α-equality, and is guaranteed safe in fully-polymorphic contexts.

3.2 Membership

Given the above representation of contexts, we can implement the relation of context membership by a type class `∈`, whose sole method performs the injection from a member of the context to the full context. The relation is defined by two inference rules, corresponding to finding the variable in the first position of the context, or further away in it, with the necessary injections:

```
instance v ∈ (a ▷ v) where
  inj = New

instance (v ∈ a) ⇒ v ∈ (a ▷ v') where
  inj = Old . inj
```

The cognoscenti will recognize the two above instances as *incoherent*, that is, if `v` and `v'` were instantiated to the same type, both instances would apply, but the injections would be different. Fortunately, this incoherence never triggers as long as one keeps the contexts maximally polymorphic contexts: `v` and `v'` will always be different.

We have seen before that the overloading of the `inj` function in the type class `∈` allows to automatically convert a type-level reference to a term into a properly tagged de Bruijn index, namely the function `var`.

Conversely, one can implement occurrence-check by combining `inj` with `(==)`: one first lifts the bound variable to the context of the chosen occurrence and then tests for equality.

```
isOccurenceOf :: (Eq a, v ∈ a) ⇒ a → v → Bool
x `isOccurenceOf` y = x == inj y
```

One can test if a variable is fresh for a given term as follows:

```
freshFor :: (Eq a, v ∈ a) ⇒ v → Tm a → Bool
x `freshFor` t = not (inj x `elem` freeVars t)
```

3.3 Inclusion

Another useful relation is context inclusion between contexts, which we also represent by a type class, named `⊆`. The sole method of the typeclass is again an injection, from the small context to the bigger one. The main application of `⊆` is in term weakening, presented at the end of sec. 4.1.

```
class a ⊆ b where
  injMany :: a → b
```

This time we have four instances: inclusion is reflexive; the empty context is the smallest one; adding a variable makes the context larger; and variable append `(▷ v)` is monotonic for inclusion.

```
instance a ⊆ a where injMany = id

instance Zero ⊆ a where injMany = magic

instance (a ⊆ b) ⇒ a ⊆ (b ▷ v) where
  injMany = Old . injMany

instance (a ⊆ b) ⇒ (a ▷ v) ⊆ (b ▷ v) where
  injMany = bimap injMany id
```

This last case uses the fact that `(▷)` is functorial in its first argument.

4. Term Structure

It is well-known that every term representation parameterized on the type of free variables should exhibit monadic structure, with substitution corresponding to the binding operator (Bellegarde and Hook 1994; Bird and Paterson 1999; Altenkirch and Reus 1999). That is, a `Monad tm` constraint means that a that a term representation `tm` is stable under substitution. In this section we review this structure, as well as other standard related structures on terms. These structures are perhaps easier to implement directly on a concrete term representation, rather than our interface. However, we give an implementation solely based on it, to demonstrate that it is complete with respect to these structures. By doing so, we also illustrate how to work with our interface in practice.

4.1 Renaming and Functors

The first, perhaps simplest, property of terms is that free variables can be renamed. This property is captured by the `Functor` structure.

The “renaming” to apply is given as a function `f` from `a` to `b` where `a` is the type of free variables of the input term (`Tm a`) and `b` is the type of free variables of the “renamed” term (`Tm b`). While the function `f` should be injective to be considered a renaming, the functor instance works well for any function `f`. The renaming operation then simply preserves the structure of the input term. At occurrence sites it uses `f` to rename free variables. At binding sites, `f` is upgraded from `(a → b)` to `(a ▷ v → b ▷ v)` using the functoriality of `(▷ v)` with `bimap f id`. Adapting the function `f` is necessary to protect the bound name from being altered by `f`, and thanks to our use of polymorphism, the type-checker ensures that we make no mistake in doing so.

```
instance Functor Tm where
  fmap f (Var x)   = Var (f x)
  fmap f (Lam b)   = unpack b $ λ x t →
                     lamP x $ fmap (bimap f id) t
  fmap f (App t u) = App (fmap f t) (fmap f u)
```

As usual satisfying functor laws implies that the structure is preserved by the functor action (`fmap`). The type for terms being a functor therefore means that applying a renaming is going to only affect the free variables and leave the structure untouched. That is, whatever the function `f` is doing, the bound names are not changing. The `Functor` laws are the following:

```
fmap id ≡ id
fmap (f . g) ≡ fmap f . fmap g
```

In terms of renaming, they mean that the identity function corresponds to not renaming anything and compositions of renaming functions corresponds to two sequential renaming operations.

Assuming only a functor structure, it is possible to write useful functions on terms which involve only renaming. A couple of examples follow.

First, let us assume an equality test on free variables. We can then write a function `rename (x,y) t` which replaces free occurrences of `x` in `t` by `y` and `swap (x,y) t` which exchanges free occurrences of `x` and `y` in `t`.

```
rename0 :: Eq a ⇒ (a, a) → a → a
rename0 (x,y) z | z == x    = y
                | otherwise = z

rename :: (Functor f, Eq a) ⇒ (a, a) → f a → f a
rename = fmap . rename0
```

```
swap0 :: Eq a ⇒ (a, a) → a → a
swap0 (x,y) z | z == y    = x
              | z == x    = y
              | otherwise = z

swap :: (Functor f, Eq a) ⇒ (a, a) → f a → f a
swap = fmap . swap0
```

Second, let us assume two arguments `a` and `b` related by the type class `⊆`. Thus we have `injMany` of type `a → b`, which can be seen as a renaming of free variables via the functorial structure of terms. By applying `fmap` to it, one obtains an arbitrary weakening from the context `a` to the bigger context `b`.

```
wk :: (Functor f, a ⊆ b) ⇒ f a → f b
wk = fmap injMany
```

Again, this arbitrary weakening function relieves the programmer from tediously counting indices and constructing an appropriate renaming function. We demonstrate this feature in sec. 6.

4.2 Substitution and Monads

Another useful property of terms is that they can be substituted for free variables in other terms. This property is captured algebraically by asserting that terms form a `Monad`, where `return` is the variable constructor and `>>=` acts as parallel substitution. Indeed, one can see a substitution from a context `a` to a context `b` as mapping from `a` to `Tm b`, (technically a morphism in the associated Kleisli category) and `(>>=)` applies a substitution everywhere in a term.

The definition of the `Monad` instance is straightforward for variable and application, and we isolate the handling of binders in the `(>>>=)` function.

```
instance Monad Tm where
  return = Var
  Var x   >>= θ = θ x
  Lam s   >>= θ = Lam (s >>>= θ)
  App t u >>= θ = App (t >>= θ) (u >>= θ)
```

At binding sites, one needs to lift the substitution so that it does not act on the newly bound variables, a behavior isolated in the helper `>>>=`. As for the `Functor` instance, the type system guarantees that no mistake is made. Perhaps noteworthy is that this operation is independent of the concrete term structure: we only “rename” with `fmap` and inject variables with `return`.

```
liftSubst :: (Functor tm, Monad tm) ⇒
          v → (a → tm b) → (a ▷ v) → tm (b ▷ v)
liftSubst _ θ (Old x) = fmap Old (θ x)
liftSubst _ θ (New x) = return (New x)
```

Substitution under a binder `(>>>=)` is then the wrapping of `liftSubst` between `unpack` and `pack`. It is uniform as well, and thus can be reused for every structure with binders.

```
(>>>=) :: (Functor tm, Monad tm) ⇒
          tm (Succ a) → (a → tm b) → tm (Succ b)
s >>>= θ = unpack s $ λ x t →
             pack x (t >>= liftSubst x θ)
```

For terms, the meaning of the monad laws can be interpreted as follows. The associativity law ensures that applying a composition of substitutions is equivalent to sequentially applying them, while the identity laws ensure that variables act indeed as such.

We can write useful functions for terms based only on the Monad structure. For example, given the membership (∈), one can provide the a generic combinator to reference to a variable within any term structure:

```
var :: (Monad tm, v ∈ a) ⇒ v → tm a
var = return . inj
```

One can also substitute an arbitrary variable:

```
substitute :: (Monad tm, Eq a, v ∈ a) ⇒
              v → tm a → tm a → tm a
substitute x t u = u >>= λ y →
     if y `isOccurenceOf` x then t else return y
```

One might however also want to remove the substituted variable from the context while performing the substitution:

```
substituteOut :: Monad tm ⇒
                 v → tm a → tm (a ▷ v) → tm a
substituteOut x t u = u >>= λ y → case y of
     New _ → t
     Old x → return x
```

4.3 Traversable

Functors enable to apply any pure function `f :: a → b` to the elements of a structure to get a new structure holding the images of `f`. Traversable structures enable to apply an effectful function `f :: a → m b` where `m` can be any `Applicative` functor. An `Applicative` functor is strictly more powerful than a `Functor` and strictly less powerful than a `Monad`. Any `Monad` is an `Applicative` and any `Applicative` is a `Functor`. To be traversed a structure only needs an applicative and therefore support monadic actions directly (McBride and Paterson 2007).

```
instance Traversable Tm where
  traverse f (Var x)   = Var <$> f x
  traverse f (App t u) =
    App <$> traverse f t <*> traverse f u
  traverse f (Lam t)   =
    unpack t $ λ x b →
      lamP x <$> traverse (bitraverse f pure) b
```

In order to traverse name abstractions, indices need to be traversed as well. The type (▷) is a bi-functor and is bi-traversable. The function `bitraverse` is given two effectful functions, one for each case:

```
bitraverse :: Functor f ⇒ (a     → f a')
                        → (b     → f b')
                        → (a ▷ b → f (a' ▷ b'))
bitraverse f _ (Old x) = Old <$> f x
bitraverse _ g (New x) = New <$> g x
```

An example of a useful effect to apply is throwing an exception, implemented for example as the `Maybe` monad. If a term has no free variable, then it can be converted from the type `Tm a` to `Tm Zero` (or equivalently `∀ b. Tm b`), but this requires a dynamic check. It may seem like a complicated implementation is necessary, but in fact it is a direct application of the `traverse` function.

```
closed :: Traversable tm ⇒ tm a → Maybe (tm b)
closed = traverse (const Nothing)
```

Thanks to terms being an instance of `Traversable` they are also `Foldable` meaning that we can combine all the elements of the structure (i.e. the occurrences of free variables in the term) using any `Monoid`. One particular monoid is the free monoid of lists. Consequently, `Data.Foldable.toList` is computing the free variables of a term and `Data.Foldable.elem` can be used to build `freshFor`:

```
freeVars' :: Tm a → [a]
freeVars' = toList

freshFor' :: (Eq a, v ∈ a) ⇒ v → Tm a → Bool
x `freshFor'` t = not (inj x `elem` t)
```

5. Scopes

Armed with an intuitive understanding of safe interfaces to manipulate de Bruijn indices, and the knowledge that one can abstract over any substitutive structure by using standard type-classes, we can recapitulate and succinctly describe the essence of our constructions.

In Nested Abstract Syntax, a binder introducing one variable in scope, for an arbitrary term structure `tm` is represented as follows:

```
type SuccScope tm a = tm (Succ a)
```

In essence, we propose two new, dual representations of binders, one based on universal quantification, the other one based on existential quantification.

```
type UnivScope  tm a = ∀ v.  v → tm (a ▷ v)
type ExistScope tm a = ∃ v. (v ,  tm (a ▷ v))
```

The above syntax for existentials is not supported in HASKELL, so we must use one of the lightweight encodings available. In the absence of view patterns, a CPS encoding is convenient for programming (so we used this so far), but a datatype representation is more convenient when dealing with scopes only:

```
data ExistScope tm a where
  E :: v → tm (a ▷ v) → ExistScope tm a
```

As we have observed on a number of examples, these representations are dual from a usage perspective: the universal-based representation allows safe the construction of terms, while the existential-based representation allows safe analysis of terms. Strictly speaking, safety holds only if one disregards non-termination and `seq`, but because the values of type `v` are never used for computation, mistakenly using a diverging term in place of a witness of variable name is far-fetched.

For the above reason, we do not commit to either side, and use the suitable representation on a case-by-case basis. This flexibility is possible because these scope representations (`SuccScope`, `UnivScope` and `ExistScope`) are isomorphic. In the following we exhibit the conversion functions between `SuccScope` one side and either `UnivScope` or `ExistScope`) on the other. We then prove that they form isomorphisms, assuming an idealized HASKELL lacking non-termination and `seq`.

5.1 `UnivScope tm a` ≅ `SuccScope tm a`

The conversion functions witnessing the isomorphism are the following.

```
succToUniv :: Functor tm ⇒
              SuccScope tm a → UnivScope tm a
succToUniv t = λ x → bimap id (const x) <$> t

univToSucc :: UnivScope tm a → SuccScope tm a
univToSucc f = f ()
```

The `univToSucc` function has not been given a name in the previous sections, but was implicitly used in the definition of `lam`. This is the first occurrence of the `succToUniv` function.

We prove first that `UnivScope` is a proper representation of `SuccScope`, that is `univToSucc . succToUniv ≡ id`. This can be done by simple equational reasoning:

```
  univToSucc (succToUniv t)
≡ {- by def -}
  univToSucc (λ x → bimap id (const x) <$> t)
≡ {- by def -}
  bimap id (const ()) <$> t
≡ {- by () having just one element -}
  bimap id id <$> t
≡ {- by (bi)functor laws -}
  t
```

The second property (`succToUniv . univToSucc ≡ id`) means that there is no ‘‘junk’’ in the representation: one cannot represent more terms in `UnivScope` than in `SuccScope`. It is more difficult to prove, as it and relies on parametricity and in turn on the lack of junk (non-termination or `seq`) in the host language. Hence we need to use the free theorem for a value `f` of type `UnivScope tm a`. Transcoding `UnivScope tm a` to a relation by using Paterson's version (Fegaras and Sheard 1996) of the abstraction theorem (Reynolds 1983; Bernardy et al. 2012), assuming additionally that `tm` is a functor. We obtain the following lemma:

```
∀ v₁:*.  ∀ v₂:*. ∀ v:v₁ → v₂.
∀ x₁:v₁. ∀ x₂:*. v x₁ ≡ x₂.
∀ g:(a ▷ v₁) → (a ▷ v₂).
(∀ y:v₁. New (v y) ≡ g (New y)) →
(∀ n:a.  Old n      ≡ g (Old n)) →
f x₂ ≡ g <$> f x₁
```

We can then specialize v_1 and x_1 to `()`, `v` to `const` x_2, and `g` to `bimap id v`. By definition, `g` satisfies the conditions of the lemma and we get:

```
f x ≡ bimap id (const x) <$> f ()
```

We can then reason equationally:

```
  f
≡ {- by the above -}
  λ x → bimap id (const x) <$> f ()
≡ {- by def -}
  succToUniv (f ())
≡ {- by def -}
  succToUniv (univToSucc f)
```

5.2 `ExistScope tm a` ≅ `SuccScope tm a`

The conversion functions witnessing the isomorphism are the following.

```
succToExist :: SuccScope tm a → ExistScope tm a
succToExist = E ()

existToSucc :: Functor tm ⇒
               ExistScope tm a → SuccScope tm a
existToSucc (E _ t) = bimap id (const ()) <$> t
```

One can recognise the functions `pack` and `unpack` as CPS versions of `existToSucc` and `succToExist`.

The proof of `existToSucc . succToExist ≡ id` (no junk) is nearly identical to the first proof about `UnivScope` and hence omitted. To prove `succToExist . existToSucc ≡ id`, we first remark that by definition:

```
succToExist (existToSucc (E y t)) ≡
  E () (fmap (bimap id (const ())) t)
```

It remains to show that `E y t` is equivalent to the right-hand side of the above equation. To do so, we consider any observation function `o` of type `∀ v. v → tm (a ▷ v) → K` for some constant type `K`, and show that it returns the same result if applied to `y` and `t` or `()` and `fmap (bimap id (const ()))` `t`. This fact is a consequence of the free theorem associated with `o`:

```
∀ v₁:*.  ∀ v₂:*. ∀ v:v₁ → v₂.
∀ x₁:v₁. ∀ x₂:*. v x₁ ≡ x₂.
∀ t₁:tm (a ▷ v₁). ∀ t₂:tm (a ▷ v₂).
(∀ g:(a ▷ v₁) → (a ▷ v₂).
 (∀ y:v₁. New (v y) ≡ g (New y)) →
 (∀ n:a.  Old n      ≡ g (Old n)) →
 t₂ ≡ fmap g t₁) →
o x₂ t₂ ≡ o x₁ t₁
```

Indeed, after specializing x_2 to `()` and `v` to `const ()`, the last condition amounts to `t₂ ≡ fmap (bimap id (const ())) t₁`, and we get the desired result.

5.3 A Matter of Style

We have seen that `ExistScope` is well-suited for term analysis, while `UnivScope` is well-suited for term construction. What about term *transformations*, which combine both aspects? In this case, one is free to choose either interface. This can be illustrated by showing both alternatives for the `Lam` case of the `fmap` function. (The `App` and `Var` cases are identical.) Because the second version is more concise, we prefer it in the upcoming examples, but the other choice is equally valid.

```
fmap' f (Lam b)
   = unpack b $ λ x t → lamP x (fmap (bimap f id) t)
fmap' f (Lam b)
   = lam (λ x → fmap (bimap f id) (b `atVar` x))
```

When using `succToUniv`, the type of the second argument of `succToUniv` should always be a type variable in order to have maximally polymorphic contexts. To remind us of this requirement when writing code, we give the alias `atVar` for `succToUniv`. (Similarly, to guarantee safety, the first argument of `pack` (encapsulated here in `lamP`) must be maximally polymorphic.)

5.4 Scope Representations and Term Representations

By using an interface such as ours, term representations can be made agnostic to the particular scope representation one might choose. In other words, if some interface appears well-suited to a given application domain, one might choose it as the scope representation in the implementation. Typically, this choice is be guided by performance considerations. Within this paper we favor code concision instead, and therefore in sec. 6.1 we use `ExistScope`, and in sections 6.2 and 6.3 we use `UnivScope`.

6. Bigger Examples

6.1 Normalization using hereditary substitution

A standard test of binder representations is how well they support normalization. In this section we show how to implement normalization using our constructions.

The following type captures normal forms of the untyped λ-calculus: a normal form is either an abstraction over a normal form or a neutral term (a variable applied to some normal forms). In this definition we use an existential-based version of scopes, which we splice in the `LamNo` constructor.

```
data No a where
  LamNo :: v → No (a ▷ v) → No a
  Neutr :: a → [No a] → No a
```

The key to this normalization procedure is that normal forms are stable under hereditary substitution (Nanevski et al. 2008). The function performing a hereditary substitution substitutes variables for their value, while reducing redexes on the fly.

```
instance Monad No where
  return x = Neutr x []
  LamNo x t  >>= θ = LamNo x (t >>= liftSubst x θ)
  Neutr f ts >>= θ = foldl app (θ f)((>>= θ)<$>ts)
```

The most notable feature of this substitution is the use of `app` to normalize redexes:

```
app :: No a → No a → No a
app (LamNo x t)  u = substituteOut x u t
app (Neutr f ts) u = Neutr f (ts++[u])
```

The normalize is then a simple recursion on the term structure:

```
norm :: Tm a → No a
norm (Var x)   = return x
norm (App t u) = app (norm t) (norm u)
norm (Lam b)   = unpack b $ λ x t →
                   LamNo x (norm t)
```

6.2 Closure Conversion

A common phase in the compilation of functional languages is closure conversion. The goal of closure conversion is make explicit the creation and opening of closures, essentially implementing lexical scope. What follows is a definition of closure conversion, as can be found in a textbook (in fact this version is slightly adapted from Guillemette and Monnier (2007)). In it, we use a hat to distinguish object-level abstractions ($\hat{\lambda}$) from host-level ones. Similarly, the @ sign is used for object-level applications.

The characteristic that interests us in this definition is that it is written in nominal style. For instance, it pretends that by matching on a $\hat{\lambda}$-abstraction, one obtains a name x and an expression e, and it is silent about the issues of freshness and transport of names between contexts. In the rest of the section, we construct an implementation which essentially retains these characteristics.

$$
\begin{aligned}
[\![x]\!] &= x \\
[\![\hat{\lambda}x.e]\!] &= \mathsf{closure}\ (\hat{\lambda}x\ x_{env}.e_{body})\ e_{env} \\
&\quad \mathsf{where}\ \ y_1,\ldots,y_n = FV(e) - \{x\} \\
&\quad\qquad\quad e_{body} \quad = [\![e]\!][x_{env}.i/y_i] \\
&\quad\qquad\quad e_{env} \quad\ = \langle y_1,\ldots,y_n\rangle \\
[\![e_1@e_2]\!] &= \mathsf{let}\ (x_f, x_{env}) = \mathsf{open}\ [\![e_1]\!] \\
&\quad \mathsf{in}\ x_f\langle[\![e_2]\!], x_{env}\rangle
\end{aligned}
$$

The first step in implementing the above function is to define the target language. It features variables and applications as usual. Most importantly, it has a constructor for `Closure`s, composed of a body and an environment. The body of closures have exactly two free variables: `vx` for the parameter of the closure and `venv` for its environment. These variables are represented by two `UnivScope`s, which we splice in the type of the constructor. An environment is realized by a `Tuple`. Inside the closure, elements of the environment are accessed via their `Index` in the tuple. Finally, the `LetOpen` construction allows to access the components of a closure (its first argument) in an arbitrary expression (its second argument). This arbitrary expression has two extra free variables: `vf` for the code of the closure and `venv` for its environment.

```
data LC a where
  VarLC :: a → LC a
  AppLC :: LC a → LC a → LC a
  Closure :: (∀ vx venv. vx → venv →
             LC (Zero ▷ venv ▷ vx)) →
             LC a → LC a
  Tuple :: [LC a] → LC a
  Index :: LC a → Int → LC a
  LetOpen :: LC a → (∀ vf venv. vf → venv →
                     LC (a ▷ vf ▷ venv)) → LC a
```

This representation is an instance of `Functor` and `Monad`, and the corresponding code offers no surprise. We give an infix alias for `AppLC`, named `$$`.

Closure conversion can then be implemented as a function from `Tm a` to `LC a`. The case of variables is trivial. For an abstraction, one must construct a closure, whose environment contains each of the free variables in the body. The application must open the closure, explicitly applying the argument and the environment.

The implementation closely follows the mathematical definition given above. The work to manage variables explicitly is limited to the lifting of the substitution $[x_{env}.i/y_i]$, and an application of `wk`. Additionally, the substitution performed `wk` is inferred automatically by GHC.

```
cc :: Eq a ⇒ Tm a → LC a
cc (Var x) = VarLC x
cc t0@(Lam b) =
  let yn = nub $ freeVars t0
  in Closure (λ x env → cc (b ‘atVar‘ x) >>=
                    liftSubst x (idxFrom yn env))
             (Tuple $ map VarLC yn)
cc (App e1 e2) =
  LetOpen (cc e1)
          (λ f x → var f $$ wk (cc e2) $$ var x)

idxFrom :: Eq a ⇒ [a] → v → a → LC (Zero ▷ v)
idxFrom yn env z = Index (var env) $
                   fromJust (elemIndex z yn)
```

A notable difference between the above implementation and that of Guillemette and Monnier is the following. They first modify the function to take an additional substitution argument, citing the difficulty to support a direct implementation with de Bruijn indices. We need not do any such modification: our interface is natural enough to support a direct implementation of the algorithm.

6.3 CPS Transform

The next example is a transformation to continuation-passing style (CPS) based partially on work by Chlipala (2008) and Guillemette and Monnier (2008). The main objective of the transformation is to make the order of evaluation explicit, by let-binding every intermediate `Value` in a specific order. To this end, we target a special representation, where every intermediate result is named. We allow for `Value`s to be pairs, so we can easily replace each argument with a pair of an argument and a continuation.

```
data TmC a where
  HaltC :: Value a → TmC a
  AppC  :: Value a → Value a → TmC a
  LetC  :: Value a → TmC (Succ a) → TmC a

data Value a where
  LamC  :: TmC (Succ a) → Value a
  PairC :: Value a → Value a → Value a
  VarC  :: a → Value a
  FstC  :: a → Value a
  SndC  :: a → Value a
```

We do not use `Value`s directly, but instead their composition with injection.

```
varC :: (v ∈ a) ⇒ v → Value a
letC :: Value a → UnivScope TmC a → TmC a
lamC :: UnivScope TmC a → Value a
fstC :: (v ∈ a) ⇒ v → Value a
sndC :: (v ∈ a) ⇒ v → Value a
```

Free variables in `TmC` can be renamed, thus it enjoys a functor structure, with a straightforward implementation found in appendix. However, this new syntax `TmC` is not stable under substitution. Building a monadic structure would be more involved, and is directly tied to the transformation we perform and the operational semantics of the language, so we omit it.

We implement a one-pass CPS transform (administrative redexes are not created). This is done by passing a host-language continuation to the transformation. At the top-level the halting continuation is used. A definition of the transformation using mathematical notation could be written as follows.

$$
\begin{aligned}
[\![x]\!]\,\kappa &= \kappa\,x\\
[\![e_1 \mathbin{@} e_2]\!]\,\kappa &= [\![e_1]\!](\lambda f.\\
&\qquad [\![e_2]\!](\lambda x.\\
&\qquad f \mathbin{@} \langle x, \kappa\rangle))\\
[\![\hat{\lambda} x.e]\!]\kappa &= \mathsf{let}\, f = \hat{\lambda} p.\ \ \mathsf{let}\, x_1 = \mathsf{fst}\, p\, \mathsf{in}\\
&\qquad\qquad\qquad \mathsf{let}\, x_2 = \mathsf{snd}\, p\, \mathsf{in}\\
&\qquad\qquad\qquad [\![e[x_1/x]]\!](\lambda r.\, x_2 \mathbin{@} r)\\
&\quad\ \mathsf{in}\, \kappa\, f
\end{aligned}
$$

The implementation follows the above definition, except for the following minor differences. For the `Lam` case, the only deviation is an occurrence of `wk`. In the `App` case, we have an additional reification of the host-level continuation as a proper `Value` using the `lamC` function. In the variable case, we must pass the variable `v` to the continuation. Doing so yields a value of type `TmC (a ▷ a)`. To obtain a result of the right type it suffices to remove the extra tagging introduced by `a ▷ a` everywhere in the term, using `(untag <$>)`. Besides, we use a number of instances of `wk`, and for each of them GHC is able to infer the substitution to perform.

```
cps0 :: Tm a → TmC a
cps0 t = cps t $ HaltC . varC

cps :: Tm a → (∀ v. v → TmC (a ▷ v)) → TmC a
cps (Var x)     k = untag <$> k x
cps (App e1 e2) k =
  cps e1 $ λ x1 →
  cps (wk e2) $ λ x2 →
  AppC (varC x1)
       (PairC (varC x2)
              (lamC (λ x → wk $ k x)))
cps (Lam e)    k =
  letC
    (lamC $ λp →
       letC (fstC p) $ λ x1 →
       letC (sndC p) $ λ x2 →
       cps (wk $ e ‘atVar‘ x1) $ λr →
       AppC (varC x2) (varC r)) k
```

It is folklore that a CPS transformation is easier to implement with higher-order abstract syntax (Guillemette and Monnier 2008; Washburn and Weirich 2003). Our interface for name abstractions features a form of higher-order representation. (Namely, a quantification, over a universally quantified type.) However limited, this higher-order aspect is enough to allow an easy implementation of the CPS transform.

7. Related Work

Representing names and binders in a safe and convenient manner is a long-standing issue, with an extensive body of work devoted to it. A survey is far beyond the scope of this paper. Hence, we limit our comparison the work that we judge most relevant, or whose contrasts with our proposal is the most revealing.

However, we do not limit our comparison to interfaces for names and binders, but also look at terms representations. Indeed, we have noted in sec. 5.3 that every term representation embodies an interface for binders.

7.1 `Fin`

Another approach already used and described by Altenkirch (1993); McBride and McKinna (2004) is to index terms, names, etc. by a number, a bound. This bound is the maximum number of distinct free variables allowed in the value. This rule is enforced in two parts: variables have to be strictly lower than their bound, and the bound is incremented by one when crossing a name abstraction (a λ-abstraction for instance).

The type `Fin n` is used for variables and represents natural numbers strictly lower than `n`. The name `Fin n` comes from the fact that it defines finite sets of size `n`.

We can draw a link with Nested Abstract Syntax. Indeed, as with the type `Succ ((▷ ()))` or `Maybe`), the type `Fin (suc n)` has exactly one more element than the type `Fin n`. However, these approaches are not equivalent for at least two reasons. Nested Abstract Syntax can accept any type to represent variables. This makes the structure more like a container and this allows to exhibit the substitutive structure of terms as monads. The `Fin` approach has advantages as well: the representation is concrete and closer to the original approach of de Brujin. In particular the representation of free and bound variables is more regular, and it may be more amenable to the optimization of variables as machine integers.

7.2 Higher-Order Abstract Syntax (HOAS)

A way to represent bindings of an object language is via the bindings of the host language. One naive translation of this idea yields the following term representation:

```
data TmH = LamH (TmH → TmH) | AppH TmH TmH
```

An issue with this kind of representation is the presence of so-called "exotic terms": a function of type `TmH → TmH` which performs pattern matching on its argument does not necessarily represent a term of the object language. A proper realization of the HOAS idea should only allow functions which use their argument for substitution.

It has been observed before that one can implement this restriction by using polymorphism. This observation also underlies the safety of our `UnivScope` representation.

Another disadvantage of HOAS is the negative occurrence of the recursive type, which makes it tricky to analyze terms (Washburn and Weirich 2003).

7.3 Syntax for free

Atkey (2009) revisited the polymorphic encoding of the HOAS representation of the untyped lambda calculus. By constructing a model of System F's parametricity in COQ he could formally prove that polymorphism rules out the exotic terms. Name abstractions, while represented by computational functions, cannot react to the shape of their argument and thus behave as substitutions. Here is this representation in HASKELL:

```
type TmF = ∀ a. ((a → a) → a)   -- lam
              → (a → a → a)    -- app
              → a
```

And our familiar application function:

```
apTmF :: TmF
apTmF lam app = lam $ λ f → lam $ λ x → f ‘app‘ x
```

Being a polymorphic encoding, this technique is limited to analyze terms via folds (catamorphism). Indeed, there is no known safe way to convert a term of this polymorphic encoding to another safe representation of names. As Atkey shows, this conversion relies on the Kripke version of the parametricity result of this type. (At the moment, the attempts to integrate parametricity in a programming language only support non-Kripke versions (Keller and Lasson 2012; Bernardy and Moulin 2012, 2013).)

7.4 Parametric Higher-Order Abstract Syntax (PHOAS)

Chlipala (2008) describes a way to represent binders using polymorphism and functions. Using that technique, called Parametric Higher-Order Abstract Syntax (PHOAS), terms of the untyped λ-calculus are as represented follows:

```
data TmP a where
  VarP :: a → TmP a
  LamP :: (a → TmP a) → TmP a
  AppP :: TmP a → TmP a → TmP a

type TmP’ = ∀ a. TmP a
```

Only universally quantified terms (`TmP’`) are guaranteed to correspond to terms of the λ-calculus.

The representation of binders used by Chlipala can be seen as a special version of `UnivScope`, where all variables are assigned the same type. This specialization has pros and cons. On the plus side, substitution is easier to implement with PHOAS: fresh variables do not need special treatment. The corresponding implementation of the monadic `join` is as follows:

```
joinP (VarP x)   = x
joinP (LamP f)   = LamP (λ x → joinP (f (VarP x)))
joinP (AppP t u) = AppP (joinP t) (joinP u)
```

On the minus side, all the variables (bound and free) have the same representation. This means that they cannot be told apart within a term of type `∀ a. TmP a`. Additionally, once the type variable `a` is instantiated to a closed type, one cannot recover the polymorphic version. Furthermore while `Tm Zero` denotes a closed term, `TmP Zero` denotes a term *without* variables, hence no term at all. Therefore, whenever a user of PHOAS needs to perform some manipulation on terms, they must make an upfront choice of a particular instance for the parameter of `TmP` that supports all the required operations on free variables. This limitation is not good for modularity, and for code clarity in general. Another issue arises from the negative occurrence of the variable type. Indeed this makes the type `TmP` invariant: it cannot be made a `Functor` nor a `Traversable` and this not a proper `Monad` either.

The use-case of PHOAS presented by Chlipala is the representation of well-typed terms. That is, the parameter to `TmP` can be made a type-function, to capture the type associated with each variable. This is not our concern here, but we have no reason to believe that our technique cannot support this, beyond the lack of proper for type-level computation in HASKELL — Chlipala uses COQ for his development.

7.5 McBride's "Classy Hack"

McBride (2010) has devised a set of combinators to construct λ-terms in de Brujin representation, with the ability to refer to bound variables by name. Terms constructed using McBride's technique are textually identical to terms constructed using ours. Another point of similarity is the use of instance search to recover the indices from a host-language variable name. A difference is that McBride integrates the injection in the abstraction constructor rather than the variable constructor. The type of the `var` combinator then becomes simpler, at the expense of `lam`:

```
lam :: ((∀ n. (Leq (S m) n ⇒ Fin n)) → Tm (S m))
       → Tm m
var :: Fin n → Tm n
```

An advantage of McBride's interface is that it does not require the "incoherent instances" extension.

However, because McBride represents variables as `Fin`, the types of his combinators are less precise ours. Notably, the `Leq` class captures only one aspect of context inclusion (captured by the class `⊆` in our development), namely that one context should be smaller than another. This means, for example, that the class constraint `a ⊆ b` can be meaningfully resolved in more cases than `Leq m n`, in turn making functions such as `wk` more useful in practice. Additionally, our `unpack` and `pack` combinators extend the technique to term analysis and manipulation.

7.6 NOMPA (nominal fragment)

Pouillard and Pottier (2012) describe an interface for names and binders which provides maximum safety. The library NOMPA is written in AGDA, using dependent types. The interface makes use of a notion of `World`s (intuitively a set of names), `Binder`s (name declaration), and Names (the occurrence of a name). A World can either be Empty (called ∅ in the library NomPa) in or result of the addition of a Binder to an existing World, using the operator (◁). The type Name is indexed by Worlds: this ties occurrences to the context where they make sense.

```
World :: *
Binder :: *
Empty :: World
(◁) :: Binder → World → World
Name :: World → *
```

On top of these abstract notions, one can construct the following representation of terms (we use a Haskell-style syntax for dependent types, similar to that of Idris):

```
data Tm α where
  Var :: Name α → Tm α
  App :: Tm α → Tm α → Tm α
  Lam :: (b :: Binder) → Tm (b ◁ α) → Tm α
```

The safety of the technique comes from the abstract character of the interface. If one were to give concrete definitions for Binder, World and their related operations, it would become possible for user code to cheat the system. A drawback of the interface being abstract is that some subterms do not evaluate. This point is of prime concern in the context of reasoning about programs involving binders. In contrast, our interfaces are concrete (code using it always evaluates), but it requires the user to choose the representation appropriate to the current use (SuccScope, UnivScope or ExistScope).

8. Discussion

8.1 Binding Many Variables

In SuccScope, there is exactly one more free variable available in the sub-term. However, it might be useful to bind multiple names at once in a binder. This can be done by using a type n of the appropriate cardinality instead of ( ). This technique has been used for example by Kmett (2012).

```
type NScope n tm a = tm (a ▷ n)
```

Adapting the idea to our framework would mean to quantify over a family of types, indexed by a type n of the appropriate cardinality:

```
type NUnivScope  n tm a = ∀v. (n → v) → tm (a ▷ v)
type NExistScope n tm a = ∃v. (n → v) ×   tm (a ▷ v)
```

8.2 Delayed Substitutions

The main performance issue with de Brujn indices comes from the cost of importing terms into scopes without capture, which requires to increment free-variables in the substituted term (see fmap Old in the definition of liftSubst). This transformation incurs not only a direct cost proportional to the size of terms, but also an indirect cost in the form of loss of sharing.

Bird and Paterson (1999) propose a solution to this issue, which can be expressed simply as another implementation of binders, where free variables of the inner term stand for whole terms with one less free variables:

```
type DelayedScope tm a = tm (tm a ▷ ())
```

This means that the parallel substitution for a term representation based on DelayedScope does not require lifting of substitutions.

```
data TmD a where
  VarD :: a → TmD a
  LamD :: DelayedScope TmD a  → TmD a
  AppD :: TmD a → TmD a → TmD a
```

```
instance Monad TmD where
  return = VarD
  VarD a >>= θ = θ a
  AppD a b >>= θ = AppD (a >>= θ) (b >>= θ)
  LamD t >>= θ = LamD (t >>= λ x → VarD $ case x of
                         New b → New b
                         Old a → Old (a >>= θ))
```

Because idea of delayed substitutions is concerned with free variables, and the concepts we present here is concerned with bound variables, one can one can easily define scopes which are both delayed and safe. Hence the performance gain can is compatible with our safe interface.

```
type UnivScope'  tm a = ∀v. (v → tm (tm a ▷ v))
type ExistScope' tm a = ∃v. (v ,  tm (tm a ▷ v))
```

8.3 Future Work: Improving Safety

As it stands our interface prevents mistakes in the manipulation of de Bruijn indices, but requires a collaboration from the user. Indeed, a malicious user can instantiate v to a monotype either in the analysis of ∀ v. v → tm (a ▷ v) or in the construction of ∃ v. (v, tm (a ▷ v)). This situation can be improved by providing a quantifier which allows to substitute for type variables only other type variables. This quantifier can be understood as being at the same time existential and universal, and hence is self dual. We the notation ∇ (pronounced nabla) for it, due to the similarity with the quantifier of the same name introduced by Miller and Tiu (2003). We would then have the following definitions, and safety could not be compromised.

```
type UnivScope  tm a = ∇ v.  v → tm (a ▷ v)
type ExistScope tm a = ∇ v. (v ,  tm (a ▷ v))
```

These definitions would preclude using SuccScope as an implementation, however this should not cause any issue: either of the above could be used directly as an implementation. Supporting our version of ∇ in a type-checker seems a rather modest extension, therefore we wish to investigate how some future version of GHC could support it.

8.4 Future Work: Improve Performance

An apparent issue with our conversion functions between ExistScope or UnivScope on one side and SuccScope on the other side is that all but succToExist cost a time proportional to the size of the term converted. In the current state of affairs, we might be able to use a system of rewrite rules, such as that implemented in GHC, to eliminate the conversions to and from the safe interfaces. However, within a system which supports ∇-quantification, a better option offers itself: the machine-representation of the type v should be nil (nothing at all) if v is a ∇-bound variable; therefore the machine-implementation of the conversions can be the identity.

8.5 Future Work: No Injections

We use the instance search of GHC in a very specific way: only to discover in injections. This suggests that a special-purpose type-system (featuring a form of subtyping) could be built to take care of those injections automatically. An obvious benefit would be some additional shortening of programs manipulating terms. Additionally, this simplification of programs would imply an even greater simplification of the proofs about them; indeed, a variation in complexity in an object usually yields a greater variation in complexity in proofs about it.

8.6 Conclusion

We have shown how to make de Bruijn indices safe, by typing them precisely with the context where they make sense. Such precise contexts are obtained is by using (appropriately) either of the interfaces `UnivScope` or `ExistScope`. These two interfaces can be seen as the both sides of the ∇ quantifier of Miller and Tiu (2003). Essentially, we have deconstructed that flavor of quantification over names, and implemented it in HASKELL. The result is a safe method to manipulate names and binders, which is supported by today's Glasgow Haskell Compiler.

The method preserves the good properties of de Bruijn indices, while providing a convenient interface to program with multiple open binders. We have illustrated these properties by exhibiting the implementation of a number of examples.

Acknowledgments

We thank Emil Axelsson, Koen Claessen, Daniel Gustafsson and Patrik Jansson for useful feedback.

References

T. Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In J. G. M. Bezem, editor, *Typed Lambda Calculi and Applications*, LNCS 664, pages 13 – 28, 1993. URL `http://www.cs.nott.ac.uk/ txa/publ/tlca93.pdf`.

T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 1999. URL `http://www.cs.nott.ac.uk/ txa/publ/csl99.pdf`.

R. Atkey. Syntax for free: representing syntax with binding using parametricity. In *International Conference on Typed Lambda Calculi and Applications (TLCA)*, volume 5608 of *Lecture Notes in Computer Science*, pages 35–49. Springer, July 2009.

F. Bellegarde and J. Hook. Substitution: A formal methods case study using monads and transformations. *Sci. Comput. Program.*, 23(2-3):287–311, 1994. doi: http://dx.doi.org/10.1016/0167-6423(94)00022-0.

J.-P. Bernardy and G. Moulin. A computational interpretation of parametricity. In *Proceedings of the Symposium on Logic in Computer Science*. IEEE Computer Society, 2012.

J.-P. Bernardy and G. Moulin. Type-theory in color. In *Proceeding of the 18th ACM SIGPLAN international conference on Functional Programming*, 2013. To appear.

J.-P. Bernardy, P. Jansson, and R. Paterson. Proofs for free — parametricity for dependent types. *Journal of Functional Programming*, 22(02):107–152, 2012. doi: 10.1017/S0956796812000056.

R. Bird and R. Paterson. de Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, Jan. 1999. URL `http://dx.doi.org/10.1017/S0956796899003366`.

A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, pages 143–156, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7. doi: http://doi.acm.org/10.1145/1411204.1411226.

N. G. de Bruijn. Lambda calculus notation with nameless dummies. In *Indagationes Mathematicae*, volume 34. Elsevier, 1972.

L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 284–294, St. Petersburg Beach, Florida, United States, 1996. ACM. ISBN 0-89791-769-3. doi: 10.1145/237721.237792. URL `http://portal.acm.org/citation.cfm?id=237792`.

L.-J. Guillemette and S. Monnier. A type-preserving closure conversion in haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 83–92. ACM, 2007.

L.-J. Guillemette and S. Monnier. A type-preserving compiler in haskell. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional Programming*, volume 43, pages 75–86. ACM, 2008.

C. Keller and M. Lasson. Parametricity in an impredicative sort. In *CSL*, pages 381–395, 2012.

E. Kmett. Bound: Making de Bruijn Succ less, 2012. URL `http://hackage.haskell.org/package/bound`. Haskell package.

C. McBride. I am not a number, i am a classy hack, 2010. URL `http://www.e-pig.org/epilogue/?p=773`. Weblog entry.

C. McBride and J. McKinna. The view from the left. *J. Funct. Program.*, 14:69–111, January 2004. ISSN 0956-7968. doi: 10.1017/S0956796803004829. URL `http://www.cs.ru.nl/ james/RESEARCH/view-final2004.pdf`.

C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(01): 1–13, 2007. doi: 10.1017/S0956796807006326. URL `http://www.journals.cambridge.org/abstract_S0956796807006326`.

D. A. Miller and A. F. Tiu. A proof theory for generic judgments: An extended abstract. In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, pages 118–127. IEEE Computer Society, June 2003.

A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Trans. Comput. Logic*, 9(3):23:1–23:49, June 2008. ISSN 1529-3785. doi: 10.1145/1352582.1352591. URL `http://doi.acm.org/10.1145/1352582.1352591`.

N. Pouillard and F. Pottier. A unified treatment of syntax with binders. *Journal of Functional Programming*, 22(4–5):614–704, 2012.

J. C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier Science, 1983. URL `ftp://ftp.cs.cmu.edu/user/jcr/typesabpara.pdf`.

G. Washburn and S. Weirich. Boxes go bananas: encoding higher-order abstract syntax with parametric polymorphism. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 249–262, Uppsala, Sweden, 2003. ACM. ISBN 1-58113-756-7. doi: 10.1145/944705.944728. URL `http://portal.acm.org/citation.cfm?id=944728`.

A. Implementation details

A.1 Traversable

```
instance Foldable Tm where
  foldMap = foldMapDefault
```

A.2 NbE

```
instance Functor No where
  fmap f (LamNo x t)  =
     LamNo x (fmap (bimap f id) t)
  fmap f (Neutr x ts) =
     Neutr (f x) (fmap (fmap f) ts)
```

A.3 CPS

```
instance Functor Value where
  fmap f (VarC x)      = VarC (f x)
  fmap f (FstC x)      = FstC (f x)
  fmap f (SndC x)      = SndC (f x)
  fmap f (PairC v1 v2) =
     PairC (fmap f v1) (fmap f v2)
  fmap f (LamC t)      =
     LamC (fmap (bimap f id) t)

instance Functor TmC where
  fmap f (HaltC v)     = HaltC (fmap f v)
  fmap f (AppC v1 v2) =
     AppC  (fmap f v1) (fmap f v2)
  fmap f (LetC p t)    =
     LetC (fmap f p) (fmap (bimap f id) t)

letC p f = LetC p (f ())
varC = VarC . inj
lamC f = LamC (f ())
fstC = FstC . inj
sndC = SndC . inj
```

A.4 Closure Conversion

```
instance Functor LC where
  fmap f t = t >>= return . f

instance Monad LC where
  return = VarLC
  VarLC x >>= θ = θ x
  Closure c env >>= θ = Closure c (env >>= θ)
  LetOpen t g >>= θ = LetOpen (t >>= θ)
    (λ f env → g f env >>=
        liftSubst env (liftSubst f θ))
  Tuple ts >>= θ = Tuple (map (>>= θ) ts)
  Index t i >>= θ = Index (t >>= θ) i
  AppLC t u >>= θ = AppLC (t >>= θ) (u >>= θ)
```

B. Bind and substitute an arbitrary name

```
packGen _ t x = fmap (shuffle cx) t
  where cx :: v → w
        cx _ = x

class (v ∈ b) ⇒ Insert v a b where
  -- inserting 'v' in 'a' yields 'b'.
  shuffle :: (v → w) → b → a ▷ w

instance Insert v a (a ▷ v) where
  shuffle f (New x) = New (f x)
  shuffle f (Old x) = Old x

instance Insert v a b ⇒
         Insert v (a ▷ v') (b ▷ v') where
  shuffle f (New x) = Old (New x)
  shuffle f (Old x) = case shuffle f x of
    New y → New y
    Old y → Old (Old y)

substituteGen ::
   (Insert v a b, Functor tm, Monad tm) ⇒
   v → tm a → tm b → tm a
substituteGen x t u =
   substituteOut x t (fmap (shuffle id) u)
```