Scrap your Nameplate

(Functional Pearl)

James Cheney

University of Edinburgh
Edinburgh, United Kingdom
jcheney@inf.ed.ac.uk

Abstract

Recent research has shown how *boilerplate* code, or repetitive code for traversing datatypes, can be eliminated using generic programming techniques already available within some implementations of Haskell. One particularly intractable kind of boilerplate is *nameplate*, or code having to do with names, name-binding, and fresh name generation. One reason for the difficulty is that operations on data structures involving names, as usually implemented, are not regular instances of standard map, $fold$, or zip operations. However, in *nominal abstract syntax*, an alternative treatment of names and binding based on swapping, operations such as α-equivalence, capture-avoiding substitution, and free variable set functions are much better-behaved.

In this paper, we show how nominal abstract syntax techniques similar to those of FreshML can be provided as a Haskell library called $FreshLib$. In addition, we show how existing generic programming techniques can be used to reduce the amount of nameplate code that needs to be written for new datatypes involving names and binding to almost nothing—in short, how to *scrap your nameplate*.

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

General Terms Languages

Keywords generic programming, names, binding, substitution

1. Introduction

Many programming tasks in a statically typed programming language such as Haskell are more complicated than they ought to be because of the need to write "boilerplate" code for traversing user-defined datatypes. *Generic programming* (the ability to write programs that work for any datatype) was once thought to require significant language extensions or external tools (for example, Generic Haskell [20]). However, over the last few years it has been shown by several authors that a great deal of generic programming can be performed safely using well-understood existing extensions to Haskell (using Hinze and Peyton Jones' *derivable type classes* [13], or Lämmel and Peyton Jones' *scrap your boilerplate* (SYB) approach [17, 18, 19]) or even entirely within Haskell 98 (using Hinze's *generics for the masses* [12]). Using these techniques it is possible to eliminate many forms of boilerplate code.

One form of boilerplate that is especially annoying is what we shall call *nameplate*: code that deals with names, fresh name generation, equality-up-to-safe-renaming, free variables, and capture-avoiding substitution. The code to accomplish these tasks usually seems straightforward, even trivial, but nevertheless apparently must be written on a per-datatype basis. The main reason for this is that capture-avoiding substitution, $FV(-)$, and α-equivalence are, as usually written, not uniform instances of map, $fold$, or zip. Although most cases are straightforward, cases involving variables or name-binding require special treatment. Despite the fact that it involves writing a lot of repetitive nameplate, the classical *first-order* approach to programming abstract syntax with names and binding is the most popular in practice.

One class of alternatives is name-free techniques such as *de Bruijn indices* [9] in which bound names are encoded using pointers or numerical indices. While often a very effective and practical implementation or compilation technique, these approaches are tricky to implement, hard for non-experts to understand, and do not provide any special assistance with open terms, fresh name generation or "exotic" forms of binding, such as pattern-matching constructs in functional languages. Also, for some tasks, such as inlining, name-free approaches seem to require more implementation effort while not being much more efficient than name-based approaches [15].

Another alternative is *higher-order abstract syntax* [24]: the technique of encoding object-language variables and binding forms using the variables and binding forms of the metalanguage. This has many advantages: efficient implementations of α-equivalence and capture-avoiding substitution are inherited from the metalanguage, and all low-level name-management details (including side-effects) are hidden, freeing the programmer to focus on high-level problems instead. While this is a very powerful approach, most interesting programming tasks involving higher-order abstract syntax require *higher-order unification*, which is common in higher-order logic programming languages such as λProlog [23] but not in functional languages, Haskell in particular. Therefore, using higher-order abstract syntax in Haskell would require significant language extensions. Also, like name-free approaches, higher-order abstract syntax does not provide any special support for programming with open terms, fresh name generation, or exotic forms of binding.

A third alternative, which we advocate, is *nominal abstract syntax*, the swapping-based approach to abstract syntax with bound names introduced by Gabbay and Pitts [10, 11, 25] and employed in the FreshML (or FreshOCaml) [26, 30] and αProlog [7] languages. This approach retains many of the advantages of first-order abstract

ICFP'05 September 26–28, 2005, Tallinn, Estonia.

syntax while providing systematic support for α-equivalence and fresh name generation. Moreover, as we shall show, nominal abstract syntax can be implemented directly in Haskell using type classes, and the definitions of nameplate functions such as capture-avoiding substitution and free variables can be generated automatically for user-defined types. Thus, nominal abstract syntax and generic programming techniques can be fruitfully combined to provide much of the convenience of higher-order abstract syntax without sacrificing the expressiveness of first-order abstract syntax and without any language extensions beyond those needed already for generic programming in Haskell.

The purpose of this paper is to show how to *scrap your nameplate* by combining nominal abstract syntax with existing generic programming techniques available in Haskell implementations such as `ghc`. As illustration, we develop a small library called $\mathit{FreshLib}$ for FreshML-style programming with nominal abstract syntax in Haskell. The main technical contribution of this paper over previous work on FreshML is showing how generic programming techniques already available in `ghc` can be used to eliminate most of the work in implementing capture-avoiding substitution and free-variables computations. Although our implementation uses advanced features currently present only in `ghc`, we believe our technique to be applicable in other situations as well.

The remainder of the paper is structured as follows. Section 2 provides a high-level overview and three examples of using $\mathit{FreshLib}$ from the user's point of view, emphasizing the fact that the library "just works" without the user needing to understand nominal abstract syntax or generic programming *a priori* or being obliged to write reams of boilerplate code. Section 3 introduces the key concepts of nominal abstract syntax and describes an initial, type class-based implementation of *FreshLib*. Section 4 shows how $\mathit{FreshLib}$ can be made completely generic using Hinze and Peyton Jones' *derivable type classes* [13] and Lämmel and Peyton Jones' *scrap your boilerplate with class* [19]; this section is very technical and relies heavily on familiarity with Lämmel and Peyton Jones' paper, so casual readers may prefer to skip it on first reading. Section 5 discusses extensions such as handling user-defined name types and alternative binding forms. Section 6 and Section 7 discuss related work and conclude.

2. $\mathit{FreshLib}$ overview and examples

2.1 $\mathit{FreshLib}$ basics

In nominal abstract syntax, it is assumed that one or more special data types of *names* is given. $\mathit{FreshLib}$ provides a data type Name of string-valued names with optional integer tags:

$\mathbf{data}\ \mathit{Name} = \mathit{Name}\ \mathit{String}\ (\mathit{Maybe}\ \mathit{Int})$

with instances for Eq, Show, and other standard classes. By convention, user-provided names (written a, b, c) have no tag, whereas names generated by $\mathit{FreshLib}$ (written a_0, b_1, etc.) are tagged.

The next ingredient of nominal abstract syntax is the assumption that all types involved in abstract syntax trees possess an α-equivalence ($==_\alpha$) relation (in addition to some other functions which the casual user doesn't need to know about):

$\mathbf{class}\ \mathit{Nom}\ a\ \mathbf{where}$
$\quad - ==_\alpha - :: a \to a \to \mathit{Bool}$
$\qquad$ -- other members discussed in Section 3

Any datatype involving Names and name-binding needs to be an instance of Nom; however, $\mathit{FreshLib}$ provides instances for Name, the abstraction constructor (see below), and all of Haskell's built-in types and constructors. Moreover, generic instances for user-defined datatypes can be derived automatically. As a result, the library user only needs to provide instances for Nom when the default behavior is not desired, e.g. when implementing a datatype with exotic binding structure (Section 5.3).

In addition, $\mathit{FreshLib}$ provides a type constructor $a \backslash\!\backslash\!\backslash\, b$ for name-abstractions, or data with binding structure:

$\mathbf{data}\ a \backslash\!\backslash\!\backslash\, b = a \backslash\!\backslash\!\backslash\, b$

Syntactically, this is just pairing. However, when a is Name and b is an instance of Nom, $\mathit{Name} \backslash\!\backslash\!\backslash\, b$ has special meaning: it represents elements of b with one bound Name. The provided instance declarations of Nom for $\mathit{Name} \backslash\!\backslash\!\backslash\, b$ define ($==_\alpha$) as α-equivalence, that is, equivalence up to safe renaming of bound names. For example, we have

$> a \backslash\!\backslash\!\backslash\, a ==_\alpha b \backslash\!\backslash\!\backslash\, b$
True
$> a \backslash\!\backslash\!\backslash\,(a, b) ==_\alpha b \backslash\!\backslash\!\backslash\,(a, b)$
False
$> a \backslash\!\backslash\!\backslash\, b ==_\alpha b \backslash\!\backslash\!\backslash\, a$
False
$> c \backslash\!\backslash\!\backslash\,(a, c) ==_\alpha b \backslash\!\backslash\!\backslash\,(a, b)$
True

Other types besides Name can also be treated as binders, but we will stick with Name-bindings only for now; we will discuss this further in Section 5.3.

The Name and $- \backslash\!\backslash\!\backslash\, -$ types are meant to be incorporated into user-defined datatypes for abstract syntax trees involving names and binding. We will give examples in Section 2.2 and Section 2.3.

Another important component of nominal abstract syntax is the ability to generate fresh names. In Haskell, one way of accomplishing this is to use a monad. Rather than fixing a (probably too specific) monad and forcing all users of $\mathit{FreshLib}$ to use it, $\mathit{FreshLib}$ provides a type class of *freshness monads* that can rename existing names to fresh ones:

$\mathbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{FreshM}\ m\ \mathbf{where}$
$\quad \mathit{renameFM} :: \mathit{Name} \to m\ \mathit{Name}$

One application of the freshness monad is to provide a monadic destructor for $\mathit{Name} \backslash\!\backslash\!\backslash\, a$ that freshens the bound name:

$\mathit{unAbs} :: \mathit{FreshM}\ m \Rightarrow \mathit{Name} \backslash\!\backslash\!\backslash\, a \to m\ (\mathit{Name}, a)$

Unlike in FreshML, pattern matching against the abstraction constructor $\backslash\!\backslash\!\backslash$ does not automatically freshen the name bound by the abstraction; instead, we need to use the unAbs destructor to explicitly freshen names.[1]

In addition to providing α-equivalence, $\mathit{FreshLib}$ also provides type classes Subst and $\mathit{FreeVars}\{\!|-|\!\}$ that perform capture-avoiding substitution and calculate sets of free names:

$\mathbf{class}\ \mathit{Subst}\ t\ u\ \mathbf{where}$
$\quad [- \mapsto -]\, -\ :: \mathit{FreshM}\ m \Rightarrow \mathit{Name} \to t \to u \to m\ u$

$\mathbf{class}\ \mathit{FreeVars}\{\!|t|\!\}\ u\ \mathbf{where}$
$\quad \mathit{FV}\{\!|t|\!\}(-) :: u \to [\mathit{Name}]$

Intuitively, $\mathit{Subst}\ t\ u$ provides a substitution function that replaces variables of type t in u; similarly, $\mathit{FreeVars}\{\!|t|\!\}\ u$ provides a function that calculates a list of the free variables of type t in u. Note that for Subst, we may need to generate fresh names (e.g. when substituting into an abstraction), so we need to work in some freshness monad m. For $\mathit{FreeVars}\{\!|t|\!\}$, fresh name generation is not needed; however, we do need to specify the type t whose free variables we seek.[2] Appropriate instances of Subst and $\mathit{FreeVars}\{\!|-|\!\}$ for Name, $\backslash\!\backslash\!\backslash$, and all built-in datatypes are provided.

---

[1] We could hide the constructor $\backslash\!\backslash\!\backslash$ using Haskell's module system and instead only export a constructor $\mathit{abs} :: a \to b \to a \backslash\!\backslash\!\backslash\, b$ and the destructor unAbs; this would legislate that abstractions can only be unpacked using unAbs. But, this would force freshening (and require computation to take place in a monad) even when unnecessary. For the same reason, the current version of FreshML also provides two ways of pattern matching abstractions, one that freshens and one that does not.

[2] Explicit type-passing $f\{\!|t|\!\}$ is not allowed in Haskell, but can be simulated by passing a *dummy argument* of type t (for example, $\mathit{undefined} :: t$)

```
module Lam where
import FreshLib
data Lam = Var Name
         | App Lam Lam
         | Lam (Name \\\ Lam)
           deriving (Nom, Eq, Show)
instance HasVar Lam where
  is_var (Var x) = Just x
  is_var y       = Nothing
```

Figure 1. Nameplate-free implementation of *Lam*

```
cbn_eval               :: FreshM m => Lam -> m Lam
cbn_eval (App t1 t2) = do w <- cbn_eval t1
                          case w
                          of Lam (a \\\ u) ->
                               do v <- [a |-> t2]u
                                  cbn_eval v
                             _ -> return (App w t2)
cbn_eval x             = return x
```

Figure 2. Call-by-name evaluation

```
module PolyLam where
import FreshLib
data Type  = VarTy Name
           | FnTy Type Type
           | AllTy (Name \\\ Type)
             deriving (Nom, Show, Eq)
data Term = Var Name
          | App Term Term
          | Lam Type (Name \\\ Term)
          | TyLam (Name \\\ Term)
          | TyApp Term Type
            deriving (Nom, Show, Eq)
instance HasVar Type where
  is_var (VarTy x) = Just x
  is_var _         = Nothing
instance HasVar Term where
  is_var (Var x)   = Just x
  is_var _         = Nothing
```

Figure 3. Nameplate-free polymorphic lambda-calculus in *FreshLib*

Note that substitution and free variable sets are not completely type-directed calculations: we need to know something about the structure of t in each case. Specifically, *we need to know how to extract a* $Name$ *from a variable of type* t. Therefore, *FreshLib* provides a class *HasVar* providing a function *is_var* that tests whether the t value is a variable, and if so, extracts its name:

```
class HasVar t where
  is_var :: t -> Maybe Name
```

Once *HasVar* t is instantiated, instances of *Subst* t u and *FreeVars*{|t|} u are derived automatically.

FreshLib provides an instance of *HasVar Name*; a name can be considered as a variable that could be replaced with another name. For example,

```
> FV{|Name|}(a \\\ (a, b))
[b]
> runFM ([b |-> a](a \\\ (a, b)))
a0 \\\ (a0, a)
```

where *runFM* is a function that evaluates a monadic expression in a particular *FreshM FM*. (Recall that names of the form $a_0, a_1, \ldots$ are names that have been freshly generated by the *FreshM*.)

2.2 The lambda-calculus

We first consider a well-worn example: implementing the syntax, α-equivalence, capture-avoiding substitution, and free variables functions of the untyped lambda-calculus. The idealized[3] Haskell code shown in Figure 1 is all that is needed to do this using *FreshLib*. First, we consider α-equivalence on *Lam*-terms:

```
>  Lam (a \\\ Var a) ==α Lam (b \\\ Var b)
True
>  Lam (a \\\ Lam (a \\\ Var a)) ==α
   Lam (b \\\ Lam (a \\\ Var b))
False
>  Lam (a \\\ Lam (a \\\ Var a)) ==α
   Lam (b \\\ Lam (a \\\ Var a))
True
```

Here are a few examples of substitution:

```
> runFM ([a |-> Var b](Lam (b \\\ Var a)))
Lam b0 \\\ (Var b)
> runFM ([a |-> Var b](Lam (b \\\ Lam (a \\\ Var a))))
Lam b0 \\\ (Lam a1 \\\ (Var a1))
```

Note that in the first example, capture is avoided by renaming b_0, while in the second, the substitution has no effect (up to α-equivalence) because a is not free in the term. Here are some examples of FV{|$-$|}$(-)$:

```
> FV{|Lam|}(Lam (a \\\ App (Var a) (Var b)))
[b]
> FV{|Lam|}(App (Var a) (Var b))
[a, b]
```

Finally, we show how call-by-name evaluation can be implemented using *FreshLib*'s built-in substitution operation in Figure 2. Here is a small example:

```
> runFM (cbn_eval (App (Lam (a \\\ Lam (b \\\
                                   App (Var a) (Var b))))
                       (Var b)))
Lam (b0 \\\ App (Var b) (Var b0))
```

2.3 The polymorphic lambda-calculus

While the above example illustrates correct handling of the simplest possible example involving one type and one kind of names, real languages often involve multiple types and different kinds of names. We now consider a more involved example: the *polymorphic lambda-calculus* (or *System F*), in which names may be used for either term variables or type variables. The *FreshLib* code for this is shown in Figure 3. Here are some examples:

```
> let t1 = AllTy (a \\\ FnTy (VarTy a) (VarTy b))
> let t2 = AllTy (b \\\ FnTy (VarTy a) (VarTy b))
> let t3 = AllTy (c \\\ FnTy (VarTy c) (VarTy b))
> t1 ==α t2
False
> t1 ==α t3
True
```

In addition, since we indicated (via *HasVar* instances) that *Type* has a variable constructor *VarTy* and *Term* has a variable constructor *Var*, appropriate implementations of $[- \mapsto -]-$ and FV{|$-$|}$(-)$ are provided also.

```
> let tm = Lam (VarTy c) (a \\\ App (Var a) (Var b))
> FV{|Term|}(tm)
[b]
> FV{|Type|}(tm)
```

[3] There are a few white lies, which we will discuss in Section 4.3.

```
class FreshM m ⇒ PolyTCM m where
  bindTV   :: Name → m a → m a
  bindV    :: Name → Type → m a → m a
  lookupTV :: Name → m Bool
  lookupV  :: Name → m (Maybe Type)
  errorTC  :: String → m a

wfTy :: PolyTCM m ⇒ Type → m ()
wfTy (VarTy n)   =
  do b ← lookupTV n
     if b then return ()
     else errorTC "Unbound variable"
wfTy (FnTy t1 t2) = do wfTy t1
                       wfTy t2
wfTy (AllTy abs)  = do (a, ty) ← unAbs abs
                       bindTV a (wfTy ty)

eqTy :: PolyTCM m ⇒ Type → Type → m ()
eqTy ty1 ty2 =
  if ty1 ==α ty2 then return ()
  else errorTC "Type expressions differ"

unFnTy :: PolyTCM m ⇒ Type → m (Type, Type)
unFnTy (FnTy ty1 ty2) = return (ty1, ty2)
unFnTy _ = errorTC "Expected function type"

unAllTy :: PolyTCM m ⇒ Type → m (Name \\\ Type)
unAllTy (AllTy abs) = return abs
unAllTy _ = errorTC "Expected forall type"
```

Figure 4. Type well-formedness and utility functions

```
inferTm :: PolyTCM m ⇒ Term → m Type
inferTm (Var x)        =
  do ty ← lookupV x
     case ty
     of Just ty′ → return ty′
        Nothing →
          errorTC "Unbound variable"
inferTm (App t1 t2)    =
  do ty1 ← inferTm t1
     (argty, resty) ← unFnTy ty1
     ty2 ← inferTm t2
     eqTy argty ty2
     return resty
inferTm (Lam ty abs)   =
  do (a, t) ← unAbs abs
     ty′ ← bindV a ty (inferTm t)
     return (FnTy ty ty′)
inferTm (TyApp tm ty) =
  do ty′ ← inferTm tm
     wfTy ty
     abs ← unAllTy ty′
     (a, ty″) ← unAbs abs
     [a ↦ ty]ty″
inferTm (TyLam abs)    =
  do (a, tm) ← unAbs abs
     ty ← bindTV a (inferTm tm)
     return (AllTy (a \\\ ty))
```

Figure 5. Type checking for the polymorphic lambda-calculus

```
[c]
> FV{|Name|}(tm)
[c, b]
> runFM ([c ↦ AllTy (c \\\ VarTy c)]tm)
Lam (AllTy (c \\\ Var c)) (a0 \\\ App (Var a0) (Var b))
> [b ↦ Var a]tm
Lam (VarTy c) (a0 \\\ App (Var a0) (Var a))
```

Finally, we give a complete implementation of type checking for $PolyLam$. Since $PolyLam$ terms are type-annotated, type checking is wholly syntax directed. Figure 4 shows the monadic interface to the typechecker ($PolyTCM$) and the type well-formedness and utility functions, and Figure 5 shows the type checker proper. The only thing that is missing is an instance of $PolyTCM$; the details of an implementation, say, TCM, are not particularly enlightening so are omitted. Here is a quick example: inferring the type of $\Lambda\alpha.\lambda x{:}\alpha{\to}\alpha.\lambda y{:}\alpha.x\ y$:

```
> runTCM (inferTm
    (TyLam (t \\\ Lam (FnTy (VarTy t) (VarTy t))
                (a \\\ Lam (VarTy t)
                     (b \\\ App (Var a) (Var b)))))))
AllTy (t0 \\\ FnTy (FnTy (VarTy t0) (VarTy t0))
                   (FnTy (VarTy t0) (VarTy t0)))
```

We stress that the code in Figure 3, Figure 4, and Figure 5 is a complete $FreshLib$ program. No boilerplate code whatsoever needs to be written to make the above program work (unless you count instantiating $PolyTCM$).

On the other hand, since there is just one type $Name$ of names, this implementation allows some nonsensical expressions to be formed that blur the distinction between type variables and term variables. This can be fixed by allowing multiple name-types. We return to this issue in Section 5.2.

2.4 A record calculus

As a final example, we sketch how the abstract syntax of a simple record calculus (an untyped fragment of part 2B of the POPLMark Challenge [5]) can be implemented in $FreshLib$. This calculus provides record constructors $\{l_1 = e_1, \ldots, l_n = e_n\}$, field lookups $e.l$, and pattern matching $let\ p = e\ in\ e'$, where patterns p consist of either pattern variables x or record patterns $\{l_1 : p_1, \ldots, l_n : p_n\}$. In both record expressions and record patterns, labels must be distinct; in patterns, variables must be distinct. The pattern variables in $let\ p = e\ in\ e'$ are considered bound in e'.

To represent this abstract syntax, we augment the Lam type as follows:

```
data Lam = ...
         | Rec [(Label, Lam)]
         | Deref Lam Label
         | Let Lam (Pat \\\ Lam)
data Pat = PVar Name
         | PRec [(Label, Pat)]
```

The Let constructor encodes the syntax $let\ p = e\ in\ e'$ as $Let\ e\ (p \mathbin{\backslash\!\backslash\!\backslash} e')$. So far, we have not given any special meaning to $t \mathbin{\backslash\!\backslash\!\backslash} u$ except when t is $Name$. In fact, $FreshLib$ provides a type class $BType$ for those types that can be bound on the left-hand size of an abstraction. So, to provide the desired behavior for pattern binding, we only need to instantiate $BType\ Pat$. The internal workings of the $BType$ class and implementation of the instance $BType\ Pat$ are deferred to Section 5.3.

This technique does not automatically equate expressions (or patterns) up to reordering of labels in record expressions, but this behavior can be provided by suitable specializations of Nom, $BType$, and Eq.

3. Implementation using type classes

In this section, we will show how a first approximation of *FreshLib* can be implemented using type classes in Haskell. The implementation in this section requires liberal amounts of boilerplate per user-defined datatype; however, this boilerplate can be eliminated using advanced generic programming techniques, as shall be shown in Section 4.

3.1 Names and nominal types

As described earlier, *Name* consists of strings with optional integer tags:

```
data Name = Name String (Maybe Int)
```

The aforementioned convention that user-provided names are untagged helps avoid collisions with names generated by *FreshLib*. This could be enforced by making *Name* abstract.

A key ingredient of nominal abstract syntax (which we glossed over earlier) is the assumption that all types of interest possess a *name-swapping* operation (•), which exchanges two names within a value, and a *freshness* operation (#), which tests that a name does not appear "free" in a value. These two operations can be used as building blocks to formalize α-equivalence (==$_\alpha$) in a particularly convenient way: in particular, it is not necessary to define α-equivalence in terms of capture-avoiding renaming and fresh name generation. The *Nom* type class includes the four functions:

```
class Nom a where
  − • −    :: Trans → a → a
  − ⊙ −    :: Perm → a → a
  π ⊙ x    = foldr (− • −) x π
  − # −    :: Name → a → Bool
  − ==α −  :: a → a → Bool
```

where the types

```
data Trans = (Name↔Name)
type Perm = [Trans]
```

indicate pairs or lists of pairs of names considered as transpositions or permutations respectively. The notation $(a{\leftrightarrow}b)$ indicates a transposition (swapping) of two names a and b. Note that the permutation-application function ($\odot$) just applies each of the transpositions in a list from right to left; it is convenient in the *BType* class in Section 5.3.

Obviously, the instance *Nom Name* needs to spell out how name-swapping, freshness and α-equivalence behave for names:

```
instance Nom Name where
  (a↔b) • c | a == c     = b
            | b == c     = a
            | otherwise  = c
  a # b                  = a /= b
  a ==α b                = a == b
```

We also provide a number of instance declarations for built-in datatypes and type constructors. For base types, these functions are trivial; for built-in type constructors such as lists and pairs, we just proceed recursively:

```
instance Nom Int where
  τ • i    = i
  a # i    = True
  i ==α j  = i == j
instance Nom a ⇒ Nom [a] where
  τ • l    = map (τ • −) l
  a # l    = all (a # −) l
  l ==α l′ = all (map (λ(x, y) → x ==α y) (zip l l′)
instance (Nom a, Nom b) ⇒ Nom (a, b) where
  τ • (x, y)           = (τ • x, τ • y)
  a # (x, y)           = a # x ∧ a # y
  (x, y) ==α (x′, y′)  = x ==α x′ ∧ y ==α y′
  -- etc...
```

3.2 Abstraction types

So far none of the types discussed binds any names. We now consider the type constructor ⑊ for *name-abstractions*, i.e. values with one bound name. Recall that the abstraction type was defined as:

```
data a ⑊ t = a ⑊ t
```

Structurally, this is just a pair of an a and a t. However, we provide an instance declaration for *Nom* (*Name* ⑊ t) that gives it a special meaning:

```
instance Nom t ⇒ Nom (Name ⑊ t) where
  τ • (a ⑊ x)           = (τ • a) ⑊(τ • x)
  a # (b ⑊ t)           = a == b ∨ a # t
  (a ⑊ x) ==α (b ⑊ y)   = (a == b ∧ x ==α y) ∨
                          (a # y ∧ x ==α (a↔b) • y)
```

Swapping is purely structural, but freshness and α-equivalence are not. In particular, a name is fresh for an abstraction if it is bound immediately or if it is fresh for the body of the abstraction. Similarly, two abstractions are α-equivalent if they are literally equal or if *the name bound on one side is fresh for the body on the other side*, and *the bodies are equal modulo swapping the bound names*.

This definition of α-equivalence has been studied by Gabbay and Pitts [10, 11, 25] and shown to be equivalent to the classical definition; earlier, a swapping-based definition was used by McKinna and Pollack [22] in a formal verification of properties of the λ-calculus. A key advantage (from the point of view of Haskell programming) is that unlike the classical definition, our definition does not require performing fresh name generation and capture-avoiding renaming in tandem with α-equivalence testing. As a result, (==$_\alpha$) can be given the same type as (==), and can be used as an equality function for nominal abstract syntax trees.

3.3 Freshness monads

The ability to swap names and test for freshness and α-equivalence is not enough for most applications. For example, to define capture-avoiding substitution, we need to be able to choose fresh names so that substitutions can be safely pushed inside abstractions. In Haskell, name-generation is usually performed using a monad [4].

In fact, different applications (e.g., parsing, typechecking, code generation) typically employ different monads. For example, it is not unusual to use a single monad for both maintaining a typechecking or evaluation environment and generating fresh names. For our purposes, we only need to know how to generate fresh names. Therefore, we define a type class of *freshness monads* (cf. Section 2) in which any computation involving a choice of fresh names can take place.

```
class Monad m ⇒ FreshM m where
  renameFM :: Name → m Name
```

Functions such as capture-avoiding substitution can then be parameterized over all freshness monads, rather than needing to be specialized to a particular one.

We also define the monadic destructor *unAbs* for unpacking an abstraction and freshening the bound name:

```
unAbs         :: FreshM m ⇒ Name ⑊ a → m (Name, a)
unAbs (a ⑊ x) = do b ← renameFM a
                   return (b, (a↔b) • x)
```

Finally, we provide a default freshness monad *FM* that simply maintains an integer counter:

```
data FM a = FM (Int → (a, Int))
instance Monad FM where
    -- omitted
```

```
instance FreshM FM where
  gensymFM s = FM (λn → (Name s (Just n), n + 1))
runFM     :: FM a → a
runFM m = let FM (a, _) = m 0 in a
```

3.4 Capture-avoiding substitution and free variables

We now show how to implement the type classes for capture-avoiding substitution and calculating sets of free variables. For *Subst*, recall that the class definition was:

```
class Subst t u where
  [− ↦ −] − :: FreshM m ⇒ Name → t → u → m u
```

We first provide instances for built-in types. In all cases, capture-avoiding substitution commutes with the existing structure. Note that no renaming needs to be performed in any of these cases.

```
instance Subst t Int where
  [n ↦ t]i = return i
instance Subst t a ⇒ Subst t [a] where
  [n ↦ t]l = mapM ([n ↦ t]−) l
instance (Subst t a, Subst t b) ⇒ Subst t (a, b) where
  [n ↦ t](a, b) = do a′ ← [n ↦ t]a
                     b′ ← [n ↦ t]b
                     return (a′, b′)
  -- etc...
```

Next, we provide an instance of *Subst Name Name*: that is, a name can be substituted for another name.

```
instance Subst Name Name where
  [a ↦ b]c = if a == c then b else c
```

Finally, we provide an instance for abstractions: if we know how to substitute for t in a, then we can also substitute for t in *Name* ⑊ a, first using *unAbs* to freshen the bound name.

```
instance Subst t a ⇒ Subst t (Name ⑊ a) where
  [n ↦ t]abs = do (a, x) ← unAbs abs
                  x′ ← [n ↦ t]x
                  return (a ⑊ x′)
```

The class *FreeVars*{|−|} is defined as follows:

```
class FreeVars{|t|} u where
  FV{|t|}(−) :: u → [Name]
```

As explained in Section 2, the type parameter t is realized as a dummy argument *undefined* :: t needed only as a typechecking hint. We can now implement the basic cases for built-in types:

```
instance FreeVars{|t|} Int where
  FV{|t|}(i)     = []
instance FreeVars{|t|} a ⇒ FreeVars{|t|} [a] where
  FV{|t|}(l)     = foldl union [] (map (FV{|t|}(−)) l)
instance
  (FreeVars{|t|} a, FreeVars{|t|} b) ⇒ FreeVars{|t|} (a, b)
where
  FV{|t|}(x, y) = FV{|t|}(x) ∪ FV{|t|}(y)
```

Next, we provide an instance of *FreeVars*{|*Name*|} *Name*:

```
instance FreeVars{|Name|} Name where
  FV{|Name|}(x) = [x]
```

Finally, for abstractions, we compute the free variables of the body and then filter out the bound name:

```
instance
  (Nom a, FreeVars{|t|} a) ⇒ FreeVars{|t|} (Name ⑊ a)
where
  FV{|t|}(a ⑊ x) = FV{|t|}(x) \\ [a]
```

Note that in this approach, the *HasVar* class is not used. As a result, instances of *Subst* and *FreeVars*{|−|} for user-defined datatypes must be provided instead. Such instances have special behavior only for cases involving variables of type t; all other cases are straightforward recursion steps (see Figure 6).

```
instance Nom Lam where
  τ • (Var c)                   = Var (τ • c)
  τ • (App t u)                 = App (τ • t) (τ • u)
  τ • (Lam abs)                 = Lam (τ • abs)
  a # (Var c)                   = a # c
  a # (App t u)                 = a # t ∧ a # u
  a # (Lam abs)                 = a # abs
  (Var n)      ==α (Var m)      = n == m
  (App t₁ t₂) ==α (App u₁ u₂)   = t₁ ==α u₁ ∧ t₂ ==α u₂
  (Lam abs₁)  ==α (Lam abs₂)    = abs₁ ==α abs₂
instance Subst Lam Lam where
  [n ↦ t](Var m)       = if n == m
                         then return t
                         else return (Var m)
  [n ↦ t](App u₁ u₂) = do t′₁ ← [n ↦ t]u₁
                          t′₂ ← [n ↦ t]u₂
                          return (App t′₁ t′₂)
  [n ↦ t](Lam abs)     = do abs′ ← [n ↦ t]abs
                            return (Lam abs′)
instance FreeVars{|Lam|} Lam where
  FV{|Lam|}(Var m)       = [m]
  FV{|Lam|}(App u₁ u₂) = FV{|Lam|}(u₁) ∪ FV{|Lam|}(u₂)
  FV{|Lam|}(Lam abs)     = FV{|Lam|}(abs)
```

Figure 6. The “nameplate” code for *Lam*

3.5 Limitations of this approach

We have now described a working type class-based implementation of *FreshLib*, culminating in definitions of capture-avoiding substitution and free variable sets for which many cases are automatically provided.

However, so far this approach has simply *reorganized* the nameplate that must be written for a new user-defined datatype involving names and binding. This reorganization has some code reuse and convenience benefits: for example, we can override and reuse the − $==_\alpha$ −, [− ↦ −]− and *FV*{|−|}(−) notations; we don’t have to write “trivial” cases for pushing substitutions inside lists, pairs, etc.; and for many datatypes, the remaining cases that need to be written down are very uniform because the tricky case for − ⑊ − is provided by *FreshLib*. Nevertheless, although the nameplate code is simpler, *we still have to write just as much boilerplate for a new datatype*. In fact, we may have to write *more* code because *Nom* needs to be instantiated for user-defined datatypes.

For example, Figure 6 shows the additional code one would have to write to implement α-equivalence, substitution, and free variables for the *Lam* type using the type class-based version of *FreshLib*. Fortunately, existing techniques for boilerplate-scrapping now can be applied, because *Nom* turns out to be a perfect example of a *derivable type class*, and [− ↦ −]− and *FV*{|−|}(−) are examples of *generic (monadic) traversals* or *generic queries* of the SYB approach. In the next section we describe how to make *FreshLib* completely generic, so that suitable instances of *Nom*, *Subst*, and *FreeVars*{|−|} are derived automatically for datatypes built up using standard types and constructors or using *Name* and − ⑊ −.

4. Implementation using generic programming

We will employ two different approaches to scrap the remaining nameplate in *FreshLib*. First, we use *derivable type classes* [13] to provide generic default definitions of the methods of *Nom* that

$$
\begin{array}{l}
\textbf{class}\ \mathit{Nom}\ a\ \textbf{where} \\
\quad \begin{array}{lll}
- \bullet - & & :: \mathit{Trans} \to a \to a \\
\tau \bullet^{\{\!|\mathit{Unit}|\!\}}\ \mathit{Unit} & & = \mathit{Unit} \\
\tau \bullet^{\{\!|a \oplus b|\!\}}\ (\mathit{Inl}\ x) & & = \mathit{Inl}\ (\tau \bullet x) \\
\tau \bullet^{\{\!|a \oplus b|\!\}}\ (\mathit{Inr}\ x) & & = \mathit{Inr}\ (\tau \bullet x) \\
\tau \bullet^{\{\!|a \otimes b|\!\}}\ (x \otimes y) & & = (\tau \bullet x) \otimes (\tau \bullet y) \\
- \# - & & :: \mathit{Name} \to a \to \mathit{Bool} \\
a \ \#^{\{\!|\mathit{Unit}|\!\}}\ \mathit{Unit} & & = \mathit{True} \\
a \ \#^{\{\!|a \oplus b|\!\}}\ (\mathit{Inl}\ x) & & = a \# x \\
a \ \#^{\{\!|a \oplus b|\!\}}\ (\mathit{Inr}\ y) & & = a \# y \\
a \ \#^{\{\!|a \otimes b|\!\}}\ (x \otimes y) & & = a \# x \wedge a \# y \\
- =\!=_\alpha - & & :: a \to a \to \mathit{Bool} \\
\mathit{Unit} & =\!=_\alpha^{\{\!|\mathit{Unit}|\!\}}\ \mathit{Unit} & = \mathit{True} \\
(\mathit{Inl}\ x) & =\!=_\alpha^{\{\!|a \oplus b|\!\}}\ (\mathit{Inl}\ x') & = x =\!=_\alpha x' \\
(\mathit{Inr}\ y) & =\!=_\alpha^{\{\!|a \oplus b|\!\}}\ (\mathit{Inr}\ y') & = y =\!=_\alpha y' \\
\_ & =\!=_\alpha^{\{\!|a \oplus b|\!\}}\ \_ & = \mathit{False} \\
(x \otimes y) & =\!=_\alpha^{\{\!|a \otimes b|\!\}}\ (x' \otimes y') & = x =\!=_\alpha x' \wedge y =\!=_\alpha y'
\end{array}
\end{array}
$$

Figure 7. Nom as a derivable type class

are suitable for most user-defined datatypes. Unfortunately, this approach does not work for Subst and $\mathit{FreeVars}\{\!|-|\!\}$, so instead we employ the latest version of Lämmel and Peyton Jones' "scrap your boilerplate" (SYB) library [19]. In particular, we make essential use of a recent innovation that supports *modular generic traversals* (i.e., traversals for which special cases can be provided using type class instances). This was not possible in previous versions of SYB.

Warning. This section (especially Section 4.2) depends rather heavily on derivable type classes and the new version of the SYB library. The papers [13] and [19] are probably prerequisite to understanding this section. However, these details do *not* have to be mastered by casual users of $\mathit{FreshLib}$.

4.1 Nom as a derivable type class

In a *derivable type class* [13] (also called *generic class* in the `ghc` documentation), we may specify the default behavior of a class method by induction on the structure of a type, expressed in terms of generic unit types Unit, sum types $a \oplus b$, and product types $a \otimes b$. To instantiate a derivable type class to a particular type (constructor), we write a structural description of the type using existing type constructors, Unit for units, $\oplus$ for sums, $\otimes$ for products, Λ for type-level abstraction and μ for recursion. For example, the structure of the Lam type is $\mu\alpha.\mathit{Name} \oplus (\alpha \otimes \alpha) \oplus \mathit{Name} \lll \alpha$, whereas the structure of the list type constructor $[\,]$ is $\Lambda\beta.\mu\alpha.\mathit{Unit} \oplus \beta \otimes \alpha$. A derivable type class declaration is specialized to a type by following the structural type description. The provided cases for Unit, $\oplus$, and $\otimes$ in the declaration are used for the corresponding cases in the type; type-level recursion is translated to term-level recursion; and type-level abstraction is translated to class dependences in instance declarations. Few generic functions are purely structure-driven, so specialized behavior can also be provided as usual by providing appropriate type class instances. These instances take precedence over the default instance provided by the derivable type class declaration. If an empty instance is provided, the default behavior is inherited.

Nom turns out to be a prime example of a derivable type class. Figure 7 shows how to define Nom as a derivable type class whose methods can be derived automatically for user-defined datatypes simply by providing an empty instance of Nom. For example, for Lam, the declaration specializes to exactly the instance of $\mathit{Nom}\ \ \mathit{Lam}$ in Figure 6. For the list type constructor, the default instance declaration for $\mathit{Nom}\ a \Rightarrow \mathit{Nom}\ [a]$ is essentially the same as the one shown in Section 3.1.

The behavior of Nom for built-in types such as Int, Char, etc. and for special $\mathit{FreshLib}$ types $- \lll -$ and Name is provided by the instances given in Section 3; no changes are needed.

4.2 Subst and $\mathit{FreeVars}\{\!|-|\!\}$ as modular generic traversals

While derivable type classes work very well for Nom, they do not help scrap the remaining boilerplate involved in Subst and $\mathit{FreeVars}\{\!|-|\!\}$. One reason is that these classes take multiple parameters, and multiple-parameter derivable type classes are not supported by `ghc`. Also, these classes provide behavior that is constructor-dependent, not just type-dependent. Derivable type classes work well when a function's behavior is dependent only on the structure of its argument type, but they are not suitable for writing functions with different behavior for different constructors of the same type. One possible solution would be to use a more powerful generic programming system such as Generic Haskell that *does* allow generic functions to display constructor-dependent behavior. This would work, but users of $\mathit{FreshLib}$ would then also need to become familiar with Generic Haskell.

Another approach that supports constructor-dependent generic functions is Lämmel and Peyton Jones' SYB library [17, 18]. This approach provides powerful facilities for "almost generic" functions which traverse the data structure generically *except for a few special cases*. We assume familiarity with this approach in the rest of this section.

Capture-avoiding substitution is *almost* an example of a *generic traversal* in the original SYB library. A naïve approach would be to implement a Lam-specific substitution function $\mathit{substLam}$ as a generic (monadic) traversal by lifting the following $\mathit{substVar}$ function to one that works for any datatype:

$$
\begin{array}{ll}
\mathit{substVar} & :: \mathit{Name} \to \mathit{Lam} \to \mathit{Lam} \to \mathit{Lam} \\
\mathit{substVar}\ a\ t\ (\mathit{Var}\ b) & = \textbf{if}\ a =\!= b \\
 & \quad\ \textbf{then}\ \mathit{return}\ t \\
 & \quad\ \textbf{else}\ \mathit{return}\ (\mathit{Var}\ b) \\
\mathit{substVar}\ a\ t\ x & = \mathit{return}\ x
\end{array}
$$

$$
\begin{array}{l}
\mathit{substLam} \quad :: \mathit{Name} \to \mathit{Lam} \to a \to a \\
\mathit{substLam}\ a\ t = \mathit{everywhereT}\ (\mathit{mkT}\ (\mathit{substVar}\ a\ t))
\end{array}
$$

Of course, this implements *capturing substitution*, which is not what we want. The natural next thing to try is to make $\mathit{substVar}$ and $\mathit{substLam}$ monadic, define a function $\mathit{substAbs}$ that gives the behavior of substitution for abstractions (performing freshening using a FreshM), and then use the extension function $\mathit{ext1M}$ of the "Scrap More Boilerplate" paper [18] to extend $\mathit{substLam}$ so that it freshens bound names appropriately.

Unfortunately, this approach does not quite work. The reason is that the function $\mathit{substAbs}$ needs to know that the type of the body is in Nom, not just Data; thus, $\mathit{substAbs}$ is *not polymorphic enough* to be used in a generic traversal. One way to solve this would be to make Nom a superclass of Data, but this is very unsatisfactory because Data is part of a library. Moreover, even if this approach *did* work, it would still have disadvantages: for example, we would have to repeat the tricky (though admittedly shorter) definition of substitution for each user-defined type, and even worse, these definitions would have to be modified if we ever added new binding types.

In fact, these are examples of more general limitations of the SYB library. As observed by Lämmel and Peyton Jones [19], the original SYB approach has two related disadvantages relative to type classes. First, generic functions are "closed" (cannot be extended) once they are defined, whereas type classes are "open" and can be extended with interesting behavior for new datatypes by providing instances. Second, SYB can only generalize *completely* *polymorphic* functions of the form $\forall a. Data\ \ a \Rightarrow a \rightarrow a$; although type-specific behavior is made possible using $cast$, *class-specific* behavior is not, and in particular, we cannot generalize functions that rely on knowing that a is an instance of some class other than $Data$.

As a result, though SYB-style generics are very powerful, they lack some of the *modularity* advantages of type classes and cannot be integrated with existing type class libraries very easily. Lämmel and Peyton Jones [19] have developed a new version of SYB that addresses both problems by, in essence, parameterizing the $Data$ type class by another type class C, so that elements of $Data\{\![C]\!\}$ can be assumed to belong to C. This form of parameterization is not allowed in Haskell proper, but may be simulated in ghc using other extensions, based on a technique due to Hughes [14]. We refer to the current SYB library as SYB3.

Using SYB3, we can implement $[- \mapsto -]-$ and $FV\{\![-]\!\}(-)$ "once and for all", rather than on a per-datatype basis. Each case in the definition of $[- \mapsto -]-$ and $FV\{\![-]\!\}(-)$ is essentially the same except for the variable constructor. Ideally, we would like to be able to parameterize the definitions of $[- \mapsto -]-$ and $FV\{\![-]\!\}(-)$ by this constructor. Haskell does not, of course, allow this kind of parameterization either, but we can simulate it using the $HasVar$ type class:

$$
\begin{array}{l}
\textbf{class}\ HasVar\ a\ \textbf{where} \\
\quad is\_var :: a \rightarrow Maybe\ Name
\end{array}
$$

Now, using SYB3, we can implement $Subst$ and $FreeVars\{\![-]\!\}$ as shown in Figure 8 and Figure 9. Following Lämmel and Peyton Jones [19], this code contains some more white lies (namely, the use of class parameters to $Data\{\![-]\!\}$ and explicit type arguments to $gfoldl\{\![-]\!\}$) that hide details of the actual encoding in Haskell. The real version is available online;[4] however, this code is likely to change to match modifications in the SYB3 library as it evolves.

The first instance declaration for $Subst$ specifies the default behavior. For most types, substitution just proceeds structurally, so we use the monadic traversal combinator $gmapM$ from SYB.

$$
\begin{array}{l}
\textbf{instance}\ Data\{\![Subst\ a]\!\}\ t \Rightarrow Subst\ a\ t\ \textbf{where} \\
\quad [a \mapsto t]x = gmapM\{\![Subst\ a]\!\}\ ([a \mapsto t]-)\ x \\
\textbf{instance} \\
\quad (HasVar\ a, Data\{\![Subst\ a]\!\}\ a) \Rightarrow Subst\ a\ a \\
\textbf{where} \\
\quad [n \mapsto t]x = \textbf{if}\ is\_var\ x == Just\ n \\
\qquad\qquad\quad \textbf{then}\ return\ t \\
\qquad\qquad\quad \textbf{else}\ gmapM\{\![Subst\ a]\!\}\ ([n \mapsto t]-)\ x
\end{array}
$$

Figure 8. Substitution using modular generics

$$
\begin{array}{l}
\textbf{instance} \\
\quad Data\{\![FreeVars\{\![a]\!\}]\!\}\ t \Rightarrow FreeVars\{\![a]\!\}\ t \\
\textbf{where} \\
\quad FV\{\![a]\!\}(x) = gfoldl\{\![FreeVars\{\![a]\!\}]\!\} \\
\qquad\qquad\qquad (\lambda fvs\_f\ y \rightarrow fvs\_f \cup FV\{\![a]\!\}(y)) \\
\qquad\qquad\qquad (\lambda\_ \rightarrow [\,])\ x \\
\textbf{instance} \\
\quad (HasVar\ a, Data\{\![FreeVars\{\![a]\!\}]\!\}\ a) \Rightarrow FreeVars\{\![a]\!\}\ a \\
\textbf{where} \\
\quad FV\{\![a]\!\}(x) = \\
\qquad \textbf{case}\ is\_var\ x \\
\qquad \textbf{of}\ Just\ n \rightarrow\ [n] \\
\qquad\quad Nothing \rightarrow gfoldl\{\![FreeVars\{\![a]\!\}]\!\} \\
\qquad\qquad\qquad (\lambda fvs\_f\ y \rightarrow fvs\_f \cup FV\{\![a]\!\}(y)) \\
\qquad\qquad\qquad (\lambda\_ \rightarrow [\,])\ x
\end{array}
$$

Figure 9. Free names using modular generics

4.3 White lies

We mentioned earlier that the picture painted of $FreshLib$ in Section 2 was a little unrealistic. This is mostly because the underlying generic programming techniques used by $FreshLib$ are still work in progress. We now describe the (mostly cosmetic) differences between the idealized code in Section 2 and what one actually has to do in the current implementation to use $FreshLib$ for a user-defined datatype T.

First off, $FreshLib$ depends on several extensions to Haskell present in ghc. The following declarations therefore need to be added to the beginning of any ghc source file making use of $FreshLib$:

```
{-# OPTIONS -fglasgow-exts #-}
{-# OPTIONS -fallow-undecidable-instances #-}
{-# OPTIONS -fallow-overlapping-instances #-}
{-# OPTIONS -fgenerics #-}
{-# OPTIONS -fth #-}
```

We also need to import parts of the $SYBnew$ library:[5]

$$
\begin{array}{l}
\textbf{import}\ SYBnew \\
\textbf{import}\ Basics \\
\textbf{import}\ Derive
\end{array}
$$

Next, even though Nom is a "derivable" type class, it is not one of Haskell 98's *built-in derivable type classes*, that is, one of the built-in classes (Eq, Ord, etc.) permitted in a **deriving** clause. So, we cannot actually write

$$\textbf{data}\ T = ...\textbf{deriving}\ (Nom, ...)$$

to automatically derive $Nom\ \ T$, but instead we need to write an empty instance

$$
\begin{array}{l}
\textbf{instance}\ Nom\ T\ \textbf{where} \\
\qquad \text{-- generic}
\end{array}
$$

in order to instantiate the "derivable" type class Nom to T. Another cosmetic difference is that as noted earlier, Haskell does not support explicit type parameters, which we have been writing as $f\{\![t]\!\}$. However, type parameter passing can be coded in Haskell using dummy arguments and ascription (e.g. writing $f\ \ (undefined :: t)$). Finally, because the latest version of the SYB library [19] relies on Template Haskell [29] to derive instances of the SYB library's $Data$ and $Typeable$ classes, we need to write a Template Haskell directive:

```
$(derive [''T])
```

However, these changes introduce at most a fixed overhead per file and user-defined datatype. All of the changes are minor and most can be expected to disappear in future versions of ghc as support is added for the modular version of the SYB library.

5. Extensions

5.1 Integrating with other type classes

One subtle problem arises if one wishes to define (==) directly as α-equivalence without having to write additional boilerplate code. In an early version of $FreshLib$, Nom only contained $(- \bullet -)$ and $(- \# -)$. We defined (==) as α-equivalence for $\lll$ and let nature take its course for other instances of (==), by defining:

$$
\begin{array}{l}
\textbf{instance}\ (Eq\ a, Nom\ a) \Rightarrow Eq\ (Name \lll a)\ \textbf{where} \\
\quad a \lll x == b \lll y = (a == b \wedge x == y) \vee \\
\qquad\qquad\qquad\qquad (a \# y \wedge x == (a \leftrightarrow b) \bullet y)
\end{array}
$$

This was unsatisfactory because (as discussed earlier) Nom cannot be mentioned in a **deriving** clause, so Eq cannot be mentioned

[4] http://homepages.inf.ed.ac.uk/jcheney/FreshLib.html

[5] available from http://www.cwi.nl/~ralf/syb3/

either (because it is dependent on *Nom* for any type containing $- \backslash\!\!\backslash\!\!\backslash -$). Thus an explicit boilerplate instance of *Eq Lam* had to be provided after *Nom Lam* was instantiated:

```
instance Nom Lam where
    -- generic
instance Eq Lam where
  (Var n) == (Var m) = n == m
    -- more boilerplate cases
```

To get rid of this boilerplate, we put a *Nom*-specific version of equality (namely, $(==_\alpha)$) into *Nom*, that can be used to provide a two-line instantiation of *Eq* whenever desired. However, to integrate *Nom* with other existing type classes (for example, to provide an instance of *Ord* compatible with α-equivalence), we would have to put additional *Nom*-specific versions of their members into *Nom*. We would prefer to be able to use our original, more modular approach; this would be possible if "derivable" type classes could be used in **deriving** clauses.

5.2 User-defined name-types

FreshLib provides a "one size fits all" type of string-valued *Name*s that is used for all name types. Often we wish to have names that carry more (or less) information than a *String*; for example, a symbol table reference, location information, namespace information, or a pointer to a variable's value.

In addition, the use of a single *Name* type for all names can lead to subtle bugs due to *Name*s of one kind "shadowing" or "capturing" *Name*s of another kind. For example, in Haskell, ordinary variables and type variables are separate, so there is no confusion resulting from using a as both a type and as a term variable. However, doing this in *FreshLib* leads to disaster:

```
>  Lam (a \\\ TyApp (Var b) (VarTy a)) ==α
   Lam (a \\\ TyApp (Var a) (VarTy a))
False
```

that is, the term-level binding of a in *Lam* *captures* the type variable a. This is not desired behavior, and to avoid this, we have to take care to ensure that term and type variable names are always distinct. Using different name types for type and term variables would rule out this kind of bug.

One way to support names of arbitrary types n is to parameterize *Name* and other types by the type of data n carried by *Name*s:

```
data Name n = Name n (Maybe Int)
type Trans n = (Name n, Name n)
type Perm n = [Trans n]
class Nom a where
  − • − :: Trans n → a → a
  − # − :: Name n → a → Bool
    -- etc...
```

An immediate difficulty in doing this is that the old instance of *Nom Name* does not work as an instance of *Name String*, or for any other type t. The reason is that we would need to provide functions

```
 − • − :: Trans n → Name t → Name t
 − # − :: Name n → Name t → Bool
```

However, in each case the behavior we want is non-parametric: if n and t are the same type, we swap names or test for inequality, otherwise swapping has no effect and freshness holds. One adequate (but probably inefficient) solution is to require n and t to be *Typeable*, so that we can test whether n and t are the same type dynamically using *cast*:

```
class Nom a where
  − • −    :: Typeable n ⇒ Trans n → a → a
  − # −    :: Typeable n ⇒ Name n → a → Bool
  − ==α − :: a → a → Bool
instance (Typeable n, Eq n) ⇒ Nom (Name n) where
  τ • n = case cast t
          of Just (a↔b) → if a == n then b
                          else if b == n then a
                          else n
             Nothing → n
  a # n = case cast a
          of Just a′ → a′ /= n
             Nothing → True
  a ==α b = a == b
```

The instances for *Nom* for basic datatypes are unchanged. For $- \backslash\!\!\backslash\!\!\backslash -$, it is necessary to use *cast* when testing for freshness:

```
instance (Typeable n, Eq n, Nom a) ⇒
         Nom ((Name n) \\\ a) where
  a # (b \\\ t) = (case cast a
                   of Just a′ → a′ == b
                      Nothing → False) ∨ a # t
```

The *FreshM*, *HasVar*, *Subst*, and *FreeVars*$\{\!|-|\!\}$ classes also need to be modified slightly but are essentially unchanged.

Another possibility would be to abstract out the type *Name* itself, and parameterize *Nom*, *FreshM*, and the other classes over n. There are two problems with this. First, `ghc` does not support multi-parameter generic type classes; and second, to avoid variable capture it is important that a *FreshM* knows how to freshen *all* kinds of names, not just a particular kind. In the approach suggested above, this is not a problem because *renameFM* :: *FreshM m* ⇒ *Name n* → *m* (*Name n*) is parametric in n.

5.3 User-defined binding forms

The name-abstraction type $Name \backslash\!\!\backslash\!\!\backslash a$ can be used for a wide variety of binding situations, but for some situations it is awkward. For example, *let*-bindings **let** $x = e_1$ **in** e_2, typed $\forall$-quantifiers $\forall x : \tau.\phi$, and binding transitions $p \stackrel{x(y)}{\rightarrow} q$ in the π-calculus can be represented using $Name \backslash\!\!\backslash\!\!\backslash a$, but the representation requires rearranging the "natural" syntax, for example as $Let\ e_1\ (x \backslash\!\!\backslash\!\!\backslash e_2)$, $Forall\ \tau\ (x \backslash\!\!\backslash\!\!\backslash \phi)$, or $BndOutTrans\ p\ x\ (y \backslash\!\!\backslash\!\!\backslash q)$.

To provide better support for the first two forms of binding, we can provide instances of $- \backslash\!\!\backslash\!\!\backslash -$ that allow binding types other than *Name*. The following code permits binding a name-value pair:

```
data a ▷ b = a ▷ b
instance
  (Nom a, Nom b) ⇒ Nom ((Name ▷ a) \\\ b)
where
  a # ((b ▷ x) \\\ y)                =
    a # x ∧ (a == b ∨ a # y)
  ((a ▷ x) \\\ y) ==α ((b ▷ x′) \\\ y′) =
    x == x′ ∧
    (a == b ∧ y == y′ ∨
      a # y′ ∧ y == (a↔b) • y′)
```

Then we can encode **let**-binding as $Let\ ((x \triangleright e_1) \backslash\!\!\backslash\!\!\backslash e_2)$ and typed quantifiers as $Forall\ ((x \triangleright \tau) \backslash\!\!\backslash\!\!\backslash \phi)$. In addition, custom instances of *Subst* and *FreeVars*$\{\!|-|\!\}$ are needed, but not difficult to derive. More exotic binding forms such as the π-calculus binding transitions can be handled in a similar fashion by defining customized instances of *Nom*, *Subst*, and *FreeVars*$\{\!|-|\!\}$.

There are other common forms of binding that cannot be handled at all using $Name \backslash\!\!\backslash\!\!\backslash a$. Some examples include

- binding a list of names, e.g. the list of parameters in a C function;
- binding the names in the domain of a typing context, e.g. $\Gamma \vdash e : \tau$ is considered equal up to renaming variables bound in Γ within e and τ;

- binding the names in a pattern-matching case, e.g. $p \to e$ is considered equal up to renaming of bound variables in p within e; and
- binding several mutually recursive names in a recursive **let**.

In each case we wish to *simultaneously bind all of an unknown number of names appearing in a value*.

We sketch a general mechanism for making a type bindable (that is, allowing it on the left side of $- \lll -$). For a type a to be bindable, we need to be able to tell *which names are bound by a a-value* and *whether two a-values are equal up to a permutation of names*. Thus, we introduce a type class for *bindable types*:

$$
\begin{array}{l}
\textbf{class}\ Nom\ a \Rightarrow BType\ a\ \textbf{where} \\
\quad BV(-) :: a \to [Name] \\
\quad - \oslash - :: a \to a \to Maybe\ Perm
\end{array}
$$

The first member, $BV(-)$, computes the set of names bound by a $BType$, whereas the second, $- \oslash -$, tests whether two values are equal up to a permutation, and returns such a permutation, if it exists. Now we can provide a very general instance for $Nom\ (a \lll b)$:

$$
\begin{array}{l}
\textbf{instance}\ (BType\ a, Nom\ b) \Rightarrow Nom\ (a \lll b)\ \textbf{where} \\
\quad a \# (x \lll y) \qquad = a \in BV(x) \lor a \# y \\
\quad (x \lll y) ==_\alpha (x' \lll y') = \\
\qquad (x ==_\alpha x' \land y ==_\alpha y') \lor \\
\qquad\quad (\textbf{case}\ x \oslash y \\
\qquad\quad\quad \textbf{of}\ Just\ \pi \to \\
\qquad\qquad\qquad (all\ (\lambda a \to a \# y')\ (BV(x) \setminus\setminus BV(y))) \land \\
\qquad\qquad\qquad (x' ==_\alpha \pi \odot y') \\
\qquad\qquad\quad Nothing \to False
\end{array}
$$

The α-equivalence test checks whether the bound data structures are equal up to a permutation, then checks that all names bound on the left-hand side but not on the right-hand side are fresh for the body on the right-hand side, and finally checks that the permutation that synchronizes the bound names also synchronizes the bodies. This is a natural, if complicated, generalization of α-equivalence for a single bound name.

In the class instance for substitution, we calculate the names bound by the left-hand side, generate fresh names, and rename the bound names to the fresh names. In the class instance for free variables, instead of subtracting the singleton list $[a]$, we subtract $BV(x)$. The details are omitted.

Then, for example, we can make contexts

$$\textbf{newtype}\ Ctx = Ctx\ [(Name, Type)]$$

bindable by implementing $BV(-)$ as $map\ fst$ and $-\oslash-$ as a function that constructs the simplest permutation π such that $ctx_1 == \pi \odot ctx_2$, if it exists. Similarly, pattern-based binding can be implemented by providing the corresponding functions for patterns. Note that we can replace the earlier instances of $Nom\ (Name \lll a)$ and $Nom\ ((Name \triangleright a) \lll b)$ by providing the following instance declarations:

$$
\begin{array}{l}
\textbf{instance}\ BType\ Name\ \textbf{where} \\
\quad BV(a) = [a] \\
\quad a \oslash b \quad = Just\ [(a \leftrightarrow b)] \\
\textbf{instance}\ (BType\ a, Nom\ b) \Rightarrow BType\ (a \triangleright b)\ \textbf{where} \\
\quad BV(a \triangleright b) \qquad = BV(a) \\
\quad (a \triangleright b) \oslash (a' \triangleright b') = \textbf{if}\ b ==_\alpha b'\ \textbf{then}\ a \oslash a'\ \textbf{else}\ Nothing
\end{array}
$$

As promised, we show how to implement the abstract syntax of pattern matching sketched in Section 2.4 as follows:

$$
\begin{array}{l}
\textbf{instance}\ BType\ Pat\ \textbf{where} \\
\quad BV(PVar\ n) \qquad\qquad = [n] \\
\quad BV(PRec\ []) \qquad\qquad = [] \\
\quad BV(PRec\ ((\_ : x) : xs)) = BV(x) +\!\!+ BV(xs) \\
\quad (PVar\ n) \oslash (PVar\ m) = Just\ [(n \leftrightarrow m)] \\
\quad (PRec\ []) \oslash (PRec\ [])\ = Just\ [] \\
\quad (PRec\ ((l, p) : r1)) \oslash (PRec\ ((l' : q) : r2)) \\
\qquad |\ l == l' \\
\qquad = \textbf{do}\ \pi \leftarrow p \oslash q \\
\qquad\qquad\quad \tau \leftarrow (PRec\ r1) \oslash (PRec\ (\pi \odot r2)) \\
\qquad\qquad\quad return\ (\tau +\!\!+ \pi) \\
\quad \_ \oslash \_ \qquad\qquad\qquad = Nothing
\end{array}
$$

Note that this implementation assumes, but does not enforce, that labels and pattern variables are distinct; thus, expressions like $\{l : e_1, l : e_2\}$ and patterns like $\{l_1 : x, l_2 : x\}$ need to be excluded manually.

Unfortunately, combining user-defined name types with user-defined binding forms appears to be nontrivial. We are currently working on combining these extensions.

5.4 Other nominal generic functions

Capture-avoiding substitution and free variables sets are just two among many possible interesting generic operations on abstract syntax with names. A few other examples include α-equivalence-respecting linear and subterm orderings; conversion to and from name-free encodings like de Bruijn indices or binary formats; syntactic unification [21, 33]; and randomized test generation as in QuickCheck [8].

Using the SYB3 library, it appears possible to define "nominal" versions of the *gfoldl*, *gmap*, *gzip*, and other combinators of $Data$, such that names are freshened by default when passing through a name-abstraction. In this approach, many interesting generic functions besides the ones we have considered would be expressible as nominal generic traversals or queries. We leave exploration of this possibility for future work.

5.5 Optimizations

Substitution and free variable computations are basic operations that need to be efficient. Currently $FreshLib$ is written for clarity, not efficiency; in particular, it follows a "sledge hammer" approach [15] in which all bound names are renamed and all subterms visited during capture-avoiding substitution. While Haskell's built-in sharing, laziness, and other optimizations offer some assistance, faster techniques for dealing with substitution are well-known, and we plan to investigate whether they can be supported in $FreshLib$.

Some minor optimizations are easy to incorporate. For example, our implementation of substitution always traverses the whole term, but we can easily modify the instance declaration for $Subst\ t\ (Name \lll a)$ to stop substitution early if we detect that the name for which we are substituting becomes bound. Similarly, we can improve the efficiency of simultaneous substitution and $FV\{\!|-|\!\}(-)$ using efficient $FiniteMap$ or Set data structures.

Another possible optimization would be to use the "rapier" approach to capture-avoiding substitution used in the `ghc` inliner and described by Peyton Jones and Marlow [15, Section 4.2]. In this approach, the set of all variables in scope is computed simultaneously with capture-avoiding substitution, and fresh names are not generated using a monad, but by hashing the set of names to guess a name that is (with high probability) not already in scope. In this approach, substitution is a pure function, so the use of monads for name-generation can be avoided. On the other hand, the hashing step may need to be repeated until a fresh name is found.

5.6 Parallelization

The order in which fresh names are generated usually has no effect on the results of computation, so theoretically, substitution operations could be reordered or even be performed in parallel. (We have in mind a fine-grained approach to parallel programming such as GPH [1]). However, the classical approach based on side-effects hides these optimization opportunities because fresh names are generated sequentially. In our approach, substitution can be performed in parallel as long as *separate threads generate distinct* *fresh names*. One way to do this is to replace the "single-threaded" freshness monad with one that can always "split" the source of fresh names into two disjoint parts. For example, fresh names could be generated using the technique of Augustsson et al. [4], in which the fresh name source is an infinite lazy tree which can be split into two disjoint fresh name sources as needed.

6. Related and Future Work

FreshML [26, 30] was an important source of inspiration for this work. Another source was logic programming languages such as λProlog [23] and Qu-Prolog [32], which provide capture-avoiding substitution as a built-in operation defined on the structure of terms.

We are aware of at least two other implementations of FreshML-like functionality as a Haskell library [35, 28], all based on essentially the same idea as ours: use type classes to provide swapping, freshness, and α-equivalence. The alternative attempts of which we are aware seem to include roughly the same functionality as discussed in the first half of Section 3, but not to use generic programming, or to consider substitution or free variable set computations at all. Sheard's library in particular inspired our treatment of freshness monads and user-defined binding forms.

Urban and Tasson [34] have used Isabelle/HOL's *axiomatic type classes* to develop a formalization of the lambda-calculus. Our techniques for generic programming with nominal abstract syntax may be relevant in this setting.

Recently, Pottier [27] has developed Cαml, a source-to-source translation tool for OCaml that converts a high-level type specification including a generalization of FreshML-like name and abstraction types. Interestingly, this approach also provides more advanced declarative support for exotic binding forms, including `letrec`. In Cαml, although capture-avoiding substitution is not built-in, it is easy to implement by overriding a *visitor* operation on syntax trees that is provided automatically. This is further evidence that nominal abstract syntax is compatible with a variety of generic programming techniques, not just those provided by `ghc`.

One advantage of implementing nominal abstract syntax as a language extension (as in FreshML and αProlog) rather than as a library is that built-in equality *is* α-equivalence, so even though name-generation is treated using side-effects or nondeterminism in these languages, capture-avoiding substitution is a pure function (i.e., has no observable side-effects up to α-equivalence). Such language extensions also have the advantage that providing user-defined name-types is straightforward; the lack of good support for the latter is probably the biggest gap in $\mathit{FreshLib}$. Although $\mathit{FreshLib}$ provides fewer static guarantees, it is more flexible in other important respects: for example, it is possible for users to define their own binding forms (Section 5.3). Another advantage of $\mathit{FreshLib}$ is that the underlying representations of names are accessible; for example, names can be ordered, and so can be used as keys in efficient data structures, whereas in FreshML and αProlog this is not allowed because there is no swapping-invariant ordering on names.

There is a large literature on efficient representations of λ-terms and implementations of capture-avoiding substitution in a variety of settings; for example, explicit substitutions [2], optimal reduction [3], and λ-DAGs [31]. We plan to attempt to integrate some such techniques into $\mathit{FreshLib}$.

Lämmel [16] proposed using generic programming, and in particular, generic traversals, as the basis for refactoring tools (that is, tools for automatic user-controlled program transformation). In this technique, refactorings can be described at a high level of generality and then instantiated to particular languages by describing the syntax and binding structure. This approach has much in common with the use of the HasVar and BType classes, and we are interested in exploring this connection further. An important difference is that in refactoring, renaming and fresh name generation is expected to be performed by the user. Thus, refactorings simply fail if a name clash is detected, whereas $\mathit{FreshLib}$ needs to be able to generate fresh names automatically in such situations.

The $\mathit{FreshLib}$ approach is a lightweight but powerful way to incorporate the novel features of FreshML inside Haskell. It seems particularly suitable for prototyping, rapid development, or educational purposes. But is it suitable for use in real Haskell programs? We are optimistic that there is some way of reconciling efficiency, modularity, and transparency, but this is an important direction for future work. One recent development that may help in this respect is Chakravarty et al.'s extension of Haskell type classes to support *associated types* [6]. We speculate that associated types may be useful for providing better support for user-defined name and binding types in $\mathit{FreshLib}$.

7. Conclusion

This paper shows that recent developments in two active research areas, *generic programming* and *nominal abstract syntax*, can be fruitfully combined to provide advanced capabilities for programming abstract syntax with names and binding in Haskell. In nominal abstract syntax, functions for comparing two terms up to renaming, calculating the set of free variables of a term, and safely substituting a term for a variable have very regular definitions—so regular, in fact, that they can be expressed using generic programming techniques already supported by extensions to Haskell such as derivable type classes and the SYB library. Moreover, these definitions can be provided *once and for all* by a library; we have developed a "proof of concept" library called $\mathit{FreshLib}$. All of the code for chores such as α-equivalence, substitution, and free variables are provided by $\mathit{FreshLib}$ and can be used without having to first learn nominal abstract syntax or generic programming, or master some external generic programming tool.

The ability to provide capture-avoiding substitution as a built-in operation is often cited as one of the main advantages of higher-order abstract syntax over other approaches. We have shown that, in the presence of generic programming techniques, this advantage is shared by nominal abstract syntax. In addition, our approach provides for more exotic forms of user-defined binding, including pattern-matching binding forms. In contrast, name-free or higher-order abstract syntax techniques provide no special assistance for this kind of binding.

On the other hand, this paper has focused on clarity over efficiency. There are many optimization techniques that we hope can be incorporated into $\mathit{FreshLib}$. The fact that $\mathit{FreshLib}$ works *at all* is encouraging, however, because it suggests that nominal abstract syntax, like higher-order abstract syntax, is a sensible high-level programming interface for names and binding. It remains to be determined whether this interface can, like higher-order abstract syntax, be implemented efficiently. We believe that $\mathit{FreshLib}$ is a promising first step towards an efficient generic library for *scrapping your nameplate*.

Acknowledgments

I wish to thank Ralf Lämmel and Simon Peyton Jones for answering questions about the new Scrap your Boilerplate library and associated paper. I also wish to thank Tim Sheard for sharing his FreshML-like Haskell library, some of whose ideas have been incorporated into $\mathit{FreshLib}$. This work was supported by EPSRC grant R37476.

References

[1] Glasgow Parallel Haskell, June 2005. `http://www.macs.hw.ac.uk/~dsg/gph/`.

[2] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[3] Andrea Asperti and Stefano Guerrini. *The Optimal Implementation of Functional Programming Languages*. Cambridge University Press, 1999.

[4] Lennart Augustsson, Mikael Rittri, and Dan Synek. On generating unique names. *J. Funct. Program.*, 4(1):117–123, 1994.

[5] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark Challenge. In *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, 2005. To appear.

[6] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 1–13, New York, NY, USA, 2005. ACM Press.

[7] J. Cheney and C. Urban. Alpha-Prolog: A logic programming language with names, binding and alpha-equivalence. In *Proc. 20th Int. Conf. on Logic Programming (ICLP 2004)*, number 3132 in LNCS, pages 269–283, 2004.

[8] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, pages 268–279. ACM, 2000.

[9] N. G. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indagationes Mathematicae*, 34(5):381–392, 1972.

[10] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In Giuseppe Longo, editor, *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, pages 193–202, Washington, DC, 1999. IEEE, IEEE Press.

[11] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.

[12] Ralf Hinze. Generics for the masses. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN International Conference on Functional Programming*, pages 236–243, New York, NY, USA, 2004. ACM Press.

[13] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.

[14] J. Hughes. Restricted data types in Haskell. In E. Meijer, editor, *Proceedings of the 1999 Haskell Workshop*, number UU-CS-1999-28 in Technical report, Utrecht University, Department of Computer Science, 1999.

[15] Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4):393–434, July 2002.

[16] Ralf Lämmel. Towards generic refactoring. In *Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*, Pittsburgh, USA, 2002. ACM Press. 14 pages.

[17] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in Language Design and Implementation*, pages 26–37, New York, NY, USA, 2003. ACM Press.

[18] Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN International Conference on Functional Programming*, pages 244–255, New York, NY, USA, 2004. ACM Press.

[19] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class. In Benjamin Pierce, editor, *Proceedings of the 10th International Conference on Functional Programming (ICFP 2005)*, Tallinn, Estonia, 2005.

[20] Andres Löh and Johan Jeuring (editors). The Generic Haskell user's guide, version 1.42 - coral release. Technical Report UU-CS-2005-004, Utrecht University, 2005.

[21] Conor McBride. First-Order Unification by Structural Recursion. *Journal of Functional Programming*, 13(6):1061–1075, 2003.

[22] James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *J. Autom. Reason.*, 23(3):373–409, 1999.

[23] G. Nadathur and D. Miller. Higher-order logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, chapter 8, pages 499–590. Oxford University Press, 1998.

[24] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '89)*, pages 199–208. ACM Press, 1989.

[25] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 183:165–193, 2003.

[26] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Proc. 5th Int. Conf. on Mathematics of Programme Construction (MPC2000)*, number 1837 in Lecture Notes in Computer Science, pages 230–255, Ponte de Lima, Portugal, July 2000. Springer-Verlag.

[27] François Pottier. An overview of Cαml, June 2005. Available at `http://cristal.inria.fr/~fpottier/publis/fpottier-alphacaml.pdf`.

[28] Tim Sheard, March 2005. Personal communication.

[29] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Haskell '02: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 1–16, New York, NY, USA, 2002. ACM Press.

[30] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programmming with binders made simple. In *Proc. 8th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP 2003)*, pages 263–274, Uppsala, Sweden, 2003. ACM Press.

[31] Olin Shivers and Mitchell Wand. Bottom-up β-reduction: Uplinks and λ-DAGs. In M. Sagiv, editor, *Proceedings of the 14th European Symposium on Programming (ESOP 2005)*, number 3444 in LNCS, pages 217–232, 2005.

[32] J. Staples, P. J. Robinson, R. A. Paterson, R. A. Hagen, A. J. Craddock, and P. C. Wallis. Qu-Prolog: An extended Prolog for meta level programming. In Harvey Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, chapter 23. MIT Press, 1996.

[33] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1–3):473–497, 2004.

[34] C. Urban and C. Tasson. Nominal techniques in Isabelle/HOL. In *Proceedings of the 20th International Conference on Automated Deduction (CADE 2005)*, 2005. To appear.

[35] Phil Wadler, Andrew Pitts, and Koen Claessen, September 2003. Personal communication.