I am not a number: I am a free variable

Conor McBride and James McKinna

June 8, 2004

Abstract

In this paper, we show how to manipulate syntax with binding using a mixed representation of names for free variables (with respect to the task in hand) and de Bruijn indices [dB72] for bound variables. By doing so, we retain the advantages of both representations: naming supports easy, arithmetic-free manipulation of terms; de Bruijn indices eliminate the need for α-conversion. Further, we have ensure that not only the user but also the *implementation* need never deal with de Bruijn indices, except within key basic operations.

Moreover, we give a representation for names which readily supports a power structure naturally reflecting the structure of the implementation. Name choice is safe and straightforward. Our technology combines easily with an approach to syntax manipulation inspired by Huet's 'zippers'[Hue97].

Without the technology in this paper, we could not have implemented Epigram [McB04]. Our example—constructing inductive elimination operators for datatype families—is but one of many where it proves invaluable.

Prologue

In conversation, we like to have *names* for the people we're talking about. If we had to say things like 'the person three to the left of me' rather than 'Fred', things would get complicated whenever anyone went to the lavatory. You don't need to have formalized the strengthening property for Pure Type Systems [MP99] to appreciate this basic phenomenon of social interaction.

It is in the company of strangers that more primitive pointing-based modes of reference acquire a useful rôle as a way of indicating unambiguously an individual with no socially agreed name. Even so, once a stranger enters the context of the conversation, he or she typically acquires a name. What this name is and who chooses it depends on the power relationships between those involved, as we learned in the playground at school.

Moreover, if we are having a conversation about hypothetical individuals—say, Alice, Bob and Unscrupulous Charlie—we have a tendency to name them locally to the discussion. We do not worry about whether Unscrupulous Charlie might actually turn out to be called Shameless David whenever he turns up. That is, we exploit naming locally to assist the construction of explanations which apply to individuals regardless of what they are called.

1 Introduction

This paper is about our everyday craft. It concerns, in particular, *naming* in the implementation of systems which manipulate syntax-with-binding. The problems we address here are not so much concerned with computations *within* such syntaxes as constructions *over* them. For example, given the declaration of an inductive datatype (by declaring the types of its constructors), how might one construct its induction principle? We encounter such issues all the time in the implementation of Epigram [MM04]. But as we develop new technology to support programming and reasoning in advanced type systems, but we must handle them effectively with today's technology. We work in Haskell and so do our students. When they ask us *what to read* in order to learn how to write the kinds of programs they will need, we tend to look blank and feel guilty. We want to do something about that.

Let's look at the example of constructing an induction principle for a datatype. Suppose someone declares

$$\mathtt{data}\,\mathtt{Nat} \;=\; \mathtt{Zero} \mid \mathtt{Suc}\,\mathtt{Nat}$$

We should like to synthesize some statement corresponding (in a suitable logic) to

$$\forall \mathtt{P} \in \mathtt{Nat} \to \mathtt{Prop}.\;\; \mathtt{P}\,\mathtt{Zero} \;\to\; (\forall \mathtt{k} \in \mathtt{Nat}.\,\mathtt{P}\,\mathtt{k} \to \mathtt{P}\,(\mathtt{Suc}\,\mathtt{k})) \;\to\; \forall \mathtt{n} \in \mathtt{Nat}.\,\mathtt{P}\,\mathtt{n}$$

In a theoretical presentation, we need not concern ourselves too much about where these names come from, and we can always choose them so that the sense is clear. In a practical implementation, we have to be more cautious—the user (unscrupulous, shameless or entirely innocent) may decide to declare

$$\mathtt{data}\,\mathtt{Nat} \;=\; \mathtt{Zero} \mid \mathtt{P}\,\mathtt{Nat}$$

or even

$$\mathtt{data}\,\mathtt{P} \;=\; \mathtt{Zero} \mid \mathtt{Suc}\,\mathtt{P}$$

We shall have to be careful not to end up with such nonsense as

$$\forall \mathtt{P} \in \mathtt{Nat} \to \mathtt{Prop}.\;\; \mathtt{P}\,\mathtt{Zero} \;\to\; (\forall \mathtt{k} \in \mathtt{Nat}.\,\mathtt{P}\,\mathtt{k} \to \mathtt{P}\,(\mathtt{P}\,\mathtt{k})) \;\to\; \forall \mathtt{n} \in \mathtt{Nat}.\,\mathtt{P}\,\mathtt{n}$$

or

$$\forall \mathtt{P} \in \mathtt{P} \to \mathtt{Prop}.\;\; \mathtt{P}\,\mathtt{Zero} \;\to\; (\forall \mathtt{k} \in \mathtt{P}.\,\mathtt{P}\,\mathtt{k} \to \mathtt{P}\,(\mathtt{Suc}\,\mathtt{k})) \;\to\; \forall \mathtt{n} \in \mathtt{P}.\,\mathtt{P}\,\mathtt{n}$$

This may seem like scaremongering, but it is not a joke—there are plenty of systems out there which do not handle this issue safely, although we shall not be so cruel as to name names. Is yours one?

Possible alternative strategies include the adoption of one of de Bruijn's systems of nameless dummies [dB72] for the local quantifiers, either counting binders (including $\to$, which we take to abbreviate $\forall$ where the bound variable is not used) from the reference outward—de Bruijn **indices**,

$$\forall - \in \mathtt{Nat} \to \mathtt{Prop}.\;\; 0\,\mathtt{Zero} \;\to\; (\forall - \in \mathtt{Nat}.\,2\,0 \to 3\,(\mathtt{Suc}\,1)) \;\to\; \forall - \in \mathtt{Nat}.\,3\,0$$

or from the outside inward—de Bruijn **levels**.

$$\forall 0 \in \mathtt{Nat} \to \mathtt{Prop}.\;\; 0\,\mathtt{Zero} \;\to\; (\forall 2 \in \mathtt{Nat}.\,0\,2 \to 0\,(\mathtt{Suc}\,2)) \;\to\; \forall 3 \in \mathtt{Nat}.\,0\,3$$

It is unfair to object that terms in de Bruijn syntax are unfit for human consumption—they are not intended to be. Their main benefits lie in their uniform delivery of capture-avoiding substitution and their systematic resolution of α-equivalence. Our foes cannot choose wicked names in order to make mischief.

However, we do recommend that anyone seriously contemplating the use of de Bruijn syntax for systematic constructions like the above should think again. Performing constructions in either of these systems requires a lot of arithmetic, which occludes the idea being implemented, results in unreadable, unreliable, unmaintainable code, and is besides hard work. *We*, or rather *our programs*, cannot choose good names in order to make sense.

A mixed representation of names provides a remedy. In this paper, we *name* free variables (ie, variables bound in the context) so that we can refer to them and rearrange them without the need to count; we give bound variables de Bruijn indices to ensure a canonical means of reference where the is no 'social agreement' on a name.

The distinction between established linguistic signs, connecting a *signal* with its *signification*, and local signs, where the particular choice of signal is arbitrary was observed in the context of natural language by Saussure [dS83]. In formal languages, the idea of distinguishing free and bound variables syntactically is also far from new. Gentzen [Gen69], Kleene [Kle52] and Prawitz [Pra65] certainly exploit it. The second author learned it from Randy Pollack who learned it in turn from Thierry Coquand [Coq91]; the first author learned it from the second.

The idea of using free *names* and bound *indices* is not new either—it is a common representation in interactive proof systems. Here 'free' means 'bound globally in the context' and 'bound' means 'bound locally in the goal'. The distinction is keyed to the human user's perspective—the user proves an implication by introducing the hypothesis to the context, naming it `H` for easy reference, although other names are, we hear, permitted. By doing so, the user shifts perspective to one which is locally more convenient, even though the resulting proof is intended to apply regardless of naming.

What is new in this paper is the use of similar perspective shifts to support the use of convenient naming in constructions where the 'user' is itself a *program*. These shifts are similar in character to those used by the second author (with Randy Pollack) when formalizing Pure Type Systems [MP93, MP99], although in that work, bound variables are distinguished from free variables but nonetheless named. We draw on the 'zipper' technique, brought to us by Huet [Hue97], to help us write programs which navigate and modify the structure of terms. Huet equips syntax with an auxiliary datatype representing a *structural* context. In our variation on his theme, we require naming as we navigate under binders to ensure that a structural context is also a *linguistic* context. In effect, whoever 'I' may be, if I am involved in the discourse, then *I am not a number*: I am a free variable.

We further choose a representation of names which readily supports the separation of namespaces between mechanical constructions which call each other and indeed themselves. The choice we make is unremarkable, in the light of how humans address similar issues in the design of large computer systems. Both the ends and the means of exploiting names in human discourse become no less pertinent when the discourse is mechanical.

As the above example may suggest, we develop our techniques in this paper for a fragment of a relational logic, featuring variables, application, and universal quantification. It can also be seen as a non-computational fragment of a dependent type theory. We've deliberately avoided a computational language in order to keep the focus on *construction*, but you can—and every day we do—certainly apply the same ideas to λ-calculi.

Acknowledgements

We should like to thank all of our friends and colleagues who have encouraged us and fed us ideas through the years. The earliest version of the programs we present here dates back to 1995—our Edinburgh days—and can still be found in the source code for LEGO version 1.3, in a file named inscrutably `conor-voodoo.sml`.

The first author would also like to thank the Foundations of Programming group at the University of Nottingham who provided the opportunity and the highly interactive audience for the informal 'Life Under Binders' course in which this work acquired its present tutorial form.

Special thanks must go to Randy Pollack, from whose conversation and implementation we have both learned a great deal.

2 La syntaxe du jour

Today, let us have variables, application, and universal quantification. We choose an entirely first-order presentation:

```
infixl 9 :$
infixr 6 :→
data Expr = F Name            -- free variables
          | B Int             -- bound variables
          | Expr :$ Expr      -- application
          | Expr :→ Scope     -- universal quantification
          deriving (Show, Eq)

newtype Scope = Scope Expr   deriving (Show, Eq)
```

We shall define Name later—for now, let us at least presume that it supports the (==) test. Observe that expressions over a common context of free Names can meaningfully be compared with the ordinary (==) test—α-conversion is not an issue. Urban sophisticates will doubtless be aware of the tricks you can play with polymorphic recursion if you parametrize expressions by names [BP99, AR99], but they would serve here only to distract from the central ideas of the paper. We shall tend to maintain a rustic monomorphism.

Nonetheless, we do introduce a cosmetic type distinction to help us remember that the scope of a binder must be interpreted differently. The Scope type stands in lieu of the precise 'term over one more variable' construction. For the most part, we shall pretend that Expr is the type of *closed* expressions—those with no 'dangling' bound variables pointing out of scope, and that Scope has one dangling bound variable, called B 0 at the top level. In order to support this pretence, however, we must first develop the key utilities which trade between free and bound variables, providing a high level interface to Scope. We shall have

```
abstract    :: Name → Expr → Scope
instantiate :: Expr → Scope → Expr
```

The operation **abstract** *me* turns a closed expression into a scope by turning *me* into B 0.

```
abstract :: Name → Expr → Scope
abstract me expr = Scope (letmeB 0 expr)  where
  letmeB this (F you) | you == me = B this
                      | otherwise = F you
  letmeB this (B that)            = B that
  letmeB this (fun :$ arg)        = letmeB this fun :$ letmeB this arg
  letmeB this (domain :→ Scope body) =
    letmeB this domain :→ Scope (letmeB (this + 1) body)
```

Observe that the existing bound variables within *expr*'s Scopes remain untouched. Meanwhile, **instantiate** *image* turns a scope into an expression by turning B 0 (or its successors under successive Scopes) into *image*, which we presume is closed. Of course, F *me* is closed, so we can use **instantiate** (F *me*) to invert **abstract** *me*.

```
instantiate :: Expr → Scope → Expr
instantiate what (Scope body) = what'sB 0 body  where
  what'sB this (B that) | this == that = what
                        | otherwise    = B that
  what'sB this (F you)                 = F you
  what'sB this (fun :$ arg)            = what'sB this fun :$ what'sB this arg
  what'sB this (domain :→ Scope body) =
    what'sB this domain :→ Scope (what'sB (this + 1) body)
```

Note that the choice of an unsophisticated de Bruijn indexed representation allows us to re-use the closed expression *what*, however many bound variables have become available when it is being referenced.

Those with an eye for abstraction will have spotted that both of these operations can be expressed as instances of a single general-purpose and rather dangerous higher-order substitution operation, parametrized by arbitrary operations on free and bound variables which in turn take a count of the Scopes we have entered—perhaps

```
varsplatter :: (Int → Name → Expr) → (Int → Int → Expr) → Expr → Expr
```

We might well do this in practice, to reduce the 'boilerplate' code required by the separate first-order definitions. Here again, we choose the direct presentation for pedagogical purposes.

Another potential optimization, given that we often iterate these operations, is to generalize **abstract**, so that it turns a *sequence* of names into dangling indices, and correspondingly **instantiate**, replacing dangling indices with a *sequence* of closed expressions. We leave this as an exercise for the reader.

From now on, let's pretend that Expr is the type of closed expressions and that Scopes have just one dangling index. The data constructors B and Scope have served their purpose—we prefer to speak of them no longer.

It's trivial to define substitution for closed expressions using **abstract** and **instantiate** (naturally, this also admits a less succinct, more efficient implementation):

```
substitute :: Expr → Name → Expr → Expr
substitute image me = instantiate image · abstract me
```

Next, let us see how **instantiate** and **abstract** enable us to navigate under binders and back out again, without ever directly encountering a de Bruijn index.

3 One-Step Navigation

We may readily define operations which attempt to decompose expressions, safely combining selection (testing which constructor is at the head) with projection (extracting subexpressions). Haskell's support for monads gives us a convenient means to handle failure when the 'wrong' constructor is present. Inverting $(:\$)$ is straightforward:

$$
\begin{array}{l}
\textbf{unapply} :: \text{MonadPlus}\, m \Rightarrow \mathsf{Expr} \rightarrow m\,(\mathsf{Expr}, \mathsf{Expr}) \\
\textbf{unapply}\ (\mathit{fun} :\$\ \mathit{arg}) = \textbf{return}\ (\mathit{fun}, \mathit{arg}) \\
\textbf{unapply}\ \_ \qquad\qquad\quad = \textbf{mzero}
\end{array}
$$

For our quantifier, however, we combine structural decomposition with the naming of the bound variable. Rather than splitting a quantified expression into a domain and a Scope, we shall extract a *binding* and the closed Expr representing the *range*. We introduce a special type of pairs which happen to be bindings, rather than using ordinary tuples, just to make the appearance of programs suitably suggestive. We equip Binding with some useful coercions.

$$
\begin{array}{l}
\text{infix } 5\ \ :\in \\
\text{data } \mathsf{Binding} = \mathsf{Name} :\in \mathsf{Expr} \\
\\
\textbf{name} :: \mathsf{Binding} \rightarrow \mathsf{Name} \\
\textbf{name}\ (\mathit{me} :\in \_) = \mathit{me} \\
\textbf{var} :: \mathsf{Binding} \rightarrow \mathsf{Expr} \\
\textbf{var} = \mathsf{F} \cdot \textbf{name}
\end{array}
$$

Now we can develop a 'smart constructor' which introduces a universal quantifier by discharging a binding, and its monadically lifted inverter:

$$
\begin{array}{l}
\text{infixr } 6\ \longrightarrow \\
(\longrightarrow) :: \mathsf{Binding} \rightarrow \mathsf{Expr} \rightarrow \mathsf{Expr} \\
(\mathit{me} :\in \mathit{domain}) \longrightarrow \mathit{range} = \mathit{domain} :\rightarrow \textbf{abstract}\ \mathit{me}\ \mathit{range} \\
\\
\text{infix } \longleftarrow \\
(\longleftarrow) :: \text{MonadPlus}\, m \Rightarrow \mathsf{Name} \rightarrow \mathsf{Expr} \rightarrow m\,(\mathsf{Binding}, \mathsf{Expr}) \\
\mathit{me} \longleftarrow (\mathit{domain} :\rightarrow \mathit{scope}) = \textbf{return}\ (\mathit{me} :\in \mathit{domain}, \textbf{instantiate}\ (\mathsf{F}\ \mathit{me})\ \mathit{scope}) \\
\mathit{me} \longleftarrow \_ \qquad\qquad\qquad\quad = \textbf{mzero}
\end{array}
$$

3.1 Inspiration—the 'zipper'

We can give an account of one-hole contexts in the style of Huet's 'zippers' [Hue97]. A Zipper is a stack, storing the information required to reconstruct an expression tree from a particular subexpression at each step on the path back to the root. The operations defined above allow us to develop the corresponding one-step manoeuvres uniformly over the type (Zipper, Expr).

$$
\begin{array}{l}
\text{infixl } 4\ \ :< \\
\text{data } \mathsf{Stack}\ x = \mathsf{Empty} \mid \mathsf{Empty}\ x :< x \quad \text{deriving (Show, Eq)}
\end{array}
$$

```
type Zipper  =  Stack Step

data Step = Fun () Expr
          | Arg Expr ()
          | Domain () Scope
          | Range Binding ()
```

This zipper structure combines the notions of *structural* and *linguistic* context—a Zipper contains the bindings for the names which may appear in any Expr which sits in the 'hole'. Note that we do not bind the variable when we edit a domain, because it is not in scope. We can easily edit these zippers, inserting new bindings (e.g., for inductive hypotheses) or permuting bindings where dependency permits, without needing to renumber de Bruijn variables.

By contrast, editing with the basic zipper, constructed with respect to the raw definition of Expr, moving into scopes without binding variables, requires a nightmare of arithmetic. The first author did most of the implementation for his Master's project[McB98] this way, before the second author caught him at it and set him on a wiser path.

The zipper construction provides a general-purpose presentation of navigation within expressions—that's a strength when we need to cope with navigation choices made by an external agency, such as the user of a structure editor. However, it's a weakness when we wish to support more focused editing strategies. In what follows, we'll be working not with the zipper itself, but with specific subtypes of it, representing particular kinds of one-hole context, such as 'quantifier prefix' or 'argument sequence'. Correspondingly, the operations we develop should be seen as specializations of Huet's.

But hold on a moment! Before we can develop more systematic editing tools, we must address the fact that navigating under a binder requires the supply of a Name. Where is this name to come from? How is it to be represented? What has the former to do with the latter? Let's now consider naming.

4 On Naming

It's not unusual to find names represented as elements of String. However, for our purposes, that won't do. String does not have enough structure to reflect the *way* names get chosen. Choosing distinct names is easy if you're the only person doing it, because you can do it deliberately. However, if there is more than one agent choosing names, we encounter the possibility that their choices will overlap by accident. How do we make sure this cannot happen? One way is to introduce a *global* symbol generator, mangling names to ensure they are globally unique; another approach requires a global counter, incremented each time a name is chosen—variables do not have names so much as *birthdays*.

Our approach is familiar from the context of module systems, or object-oriented programming. We control the anarchy of naming by introducing a power structure—we have *hierarchical* names.

```
type Name  =  Stack (String, Int)
```

Names get longer as you move down the hierarchy: *power* is thus characterized by the partial order dual to the obvious 'prefix' ordering, induced by concatenation:

$$you \succcurlyeq (you \dashv\!< yourstaff)$$

$$\begin{array}{l} \text{infixl } 4 \ \dashv\!< \\ (\dashv\!<) :: \mathsf{Stack}\ x \to \mathsf{Stack}\ x \to \mathsf{Stack}\ x \\ xs \dashv\!< \mathsf{Empty} \quad = xs \\ xs \dashv\!< (ys :< y) = xs \dashv\!< ys :< y \end{array}$$

Why a Stack? The idea is to give names to the *agents* which perform constructions, as well as to the variables which appear in constructions. We say that agents are **independent** if their names are mutually incomparable with respect to $\succcurlyeq$. In our operations, we shall ensure that agents only choose names over which they have power—longer names than their own. It is not hard to see that independent names independently extended are still independent. This scheme of naming thus *localizes* choice of fresh names, making it easy to manage, even in recursive constructions. We only need a global name generator when printing de Bruijn syntax in user-legible form, and even then only to provide names which correspond closely to those for which the user has indicated a preference.

Why (String, Int)? The Strings give us legibility; the Ints an easy way to express uniform sequences of names $x_0, \ldots x_n$. Two little helpers will make simple names easier to construct:

$$\begin{array}{l} \text{infixl } 6 \ /\!/ \\ (/\!/) :: \mathsf{Name} \to \mathsf{String} \to \mathsf{Name} \\ me /\!/ s = me :< (s, 0) \\ \\ \mathbf{nm} :: \mathsf{String} \to \mathsf{Name} \\ \mathbf{nm}\ s = \mathsf{Empty} /\!/ s \end{array}$$

We shall develop our operations in the form of *agencies*.

$$\text{type } \mathsf{Agency}\ agent = \mathsf{Name} \to agent$$

We think of an Agency t which takes a name to an *agent* with that name. We'll be careful to maintain the power relationship between an agent's name and the names it chooses for variables and for its sub-agents. You have already seen an agency—the under-binding navigator, which may be retyped

$$\begin{array}{l} \text{infix } \longleftarrow \\ (\longleftarrow) :: \text{MonadPlus}\ m \Rightarrow \mathsf{Agency}\ (\mathsf{Expr} \to m\ (\mathsf{Binding}, \mathsf{Expr})) \end{array}$$

That is, $(me \longleftarrow)$ is the agent which binds *me* by decomposing a quantifier. Note that here the agent which creates the binding shares its name—that's a common idiom in our work.

5 A Construction Kit

Let's now build higher-level tools for composing and decomposing expressions. Firstly, we'll have equipment for working with a *quantifier prefix*, rather than individual bindings—here is the operator which discharges a prefix over an expression, iterating $\longrightarrow$.

$$
\begin{array}{l}
\mathrm{type}\ \mathsf{Prefix}\ =\ \mathsf{Stack\ Binding} \\[1em]
\mathrm{infixr}\ 6\ \twoheadrightarrow \\
(\twoheadrightarrow)\ ::\ \mathsf{Prefix} \to \mathsf{Expr} \to \mathsf{Expr} \\
\mathsf{Empty} \qquad\qquad\qquad\qquad \twoheadrightarrow \mathit{expr} \ = \mathit{expr} \\
(\mathit{bindings}\ :<\ \mathit{binding}) \twoheadrightarrow \mathit{range} = \mathit{bindings}\ \twoheadrightarrow\ \mathit{binding} \longrightarrow \mathit{range}
\end{array}
$$

The corresponding destructor is an *agency*. Given a name me and a string x, it delivers a quantifier prefix with names of the form $me :< (x, \_i)$ where the 'subscript' $\_i$ is numbered from 1:

$$
\begin{array}{l}
\mathbf{unprefix}\ ::\ \mathsf{Agency}\ (\mathsf{String} \to \mathsf{Expr} \to (\mathsf{Prefix}, \mathsf{Expr})) \\
\mathbf{unprefix}\ \mathit{me}\ x\ \mathit{thing}\ = \mathbf{introduce}\ 1\ (\mathsf{Empty}, \mathit{thing}) \quad \mathrm{where} \\
\quad \mathbf{introduce}\ ::\ \mathsf{Int} \to (\mathsf{Prefix}, \mathsf{Expr}) \to (\mathsf{Prefix}, \mathsf{Expr}) \\
\quad \mathbf{introduce}\ \_i\ (\mathit{bindings}, \mathit{thing})\ = \\
\quad\quad \mathrm{case}\ (\mathit{me}\ :<\ (x, \_i)) \longleftarrow \mathit{thing}\ \mathrm{of} \\
\quad\quad\quad \mathsf{Just}\ (\mathit{binding}, \mathit{range}) \to \mathbf{introduce}\ (\_i + 1)\ (\mathit{bindings}\ :<\ \mathit{binding},\ \mathit{range}) \\
\quad\quad\quad \mathsf{Nothing} \qquad\qquad\qquad\ \to (\mathit{bindings},\ \mathit{thing})
\end{array}
$$

Note that **introduce** specifically exploits the Maybe instance of the monadically lifted binding agency ($\longleftarrow$).

If me is suitably independent, and **unprefix** $me\ x\ expr\ =\ (bindings, range)$, then $range$ is unquantified and $expr = bindings\ \twoheadrightarrow\ range$.

A little example will show how these tools are used. Suppose we wish to implement the *weakening* agency, which inserts a new hypothesis y with a given domain into a quantified expression after all the old ones $(x_1, \ldots, x_n)$. Here's how we do it safely and with names, not arithmetic.

$$
\begin{array}{l}
\mathbf{weaken} :: \mathsf{Agency}\ (\mathsf{Expr} \to \mathsf{Expr} \to \mathsf{Expr}) \\
\mathbf{weaken}\ \mathit{me}\ \mathit{dom}\ \mathit{expr}\ =\ \mathit{doms}\ \twoheadrightarrow\ (\mathit{me}\ /\!/\ \text{"y"}\ :\in\ \mathit{dom}) \longrightarrow \mathit{range} \quad \mathrm{where} \\
\quad (\mathit{doms},\ \mathit{range})\ =\ \mathbf{unprefix}\ \mathit{me}\ \text{"x"}\ \mathit{expr}
\end{array}
$$

The independence of the name supplied to the agency is enough to ensure the freshness of the names chosen locally by the agent.

We shall also need to build and decompose applications in terms of argument *sequences*, represented via [Expr]. First, we iterate $:\$$, yielding $\$\$$.

$$
\begin{array}{l}
\mathrm{infixl}\ 9\ \$\$ \\
(\$\$)\ ::\ \mathsf{Expr} \to [\mathsf{Expr}] \to \mathsf{Expr} \\
\mathit{expr}\ \$\$\ [] \qquad\qquad\quad = \mathit{expr} \\
\mathit{fun}\ \ \$\$\ (\mathit{arg}\ :\ \mathit{args}) = \mathit{fun}\ :\$\ \mathit{arg}\ \$\$\ \mathit{args}
\end{array}
$$

Next, we build the destructor—this does not need to be an agency, as it binds no names:

$$
\begin{array}{l}
\mathbf{unapplies}\ ::\ \mathsf{Expr} \to (\mathsf{Expr},\ [\mathsf{Expr}]) \\
\mathbf{unapplies}\ \mathit{expr}\ =\ \mathbf{peel}\ (\mathit{expr},\ []) \quad \mathrm{where} \\
\quad \mathbf{peel}\ (\mathit{fun}\ :\$\ \mathit{arg},\ \mathit{args}) = \mathbf{peel}\ (\mathit{fun},\ \mathit{arg}\ :\ \mathit{args}) \\
\quad \mathbf{peel}\ \mathit{funargs} \qquad\qquad\quad = \mathit{funargs}
\end{array}
$$

Meaningful formulae in this particular language of expressions all fit the pattern $\forall\, x_1 : X_1.\ \ldots \forall\, x_m : X_m.\, R\, e_1\ \ldots\ e_n$, where R is a variable. Of course, either the quantifier prefix or the argument sequence or both may be empty—this pattern excludes only meaningless applications of quantified formulae. Note that the same is not true of languages with λ-abstraction and β-redices, but here we may reasonably presume that the meaningless case never happens, and develop a one-stop analysis agency:

$$\mathrm{data}\ \mathsf{Analysis}\ =\ \mathsf{ForAll}\ \mathsf{Prefix}\ \mathsf{Name}\ [\mathsf{Expr}]$$

$$\begin{array}{l}
\textbf{analysis}\ ::\ \mathsf{Agency}\ (\mathsf{String}\ \to\ \mathsf{Expr}\ \to\ \mathsf{Analysis}) \\
\textbf{analysis}\ \mathit{me}\ x\ \mathit{expr}\ =\ \mathsf{ForAll}\ \mathit{prefix}\ \ f\ \mathit{args}\quad \mathrm{where} \\
\quad (\mathit{prefix}, \mathit{range})\ =\ \textbf{unprefix}\ \mathit{me}\ x\ \mathit{expr} \\
\quad (\mathsf{F}\ f, \mathit{args})\ =\ \textbf{unapplies}\ \mathit{range}
\end{array}$$

Again, the datatype Analysis is introduced only to make the appearance of the result suitably suggestive of its meaning, especially in patterns.

The final piece of kit we shall define in this section delivers the application of a variable to a quantifier prefix—in practice, usually the very quantifier prefix over which it is abstracted, yielding a typical application of a functional object:

$$\begin{array}{l}
\mathrm{infixl}\ 9\ \ -\$\$ \\
(\ -\$\$\ )\ ::\ \mathsf{Name}\ \to\ \mathsf{Prefix}\ \to\ \mathsf{Expr} \\
f\ -\$\$\ \mathit{parameters}\ =\ \textbf{apply}\ (\mathsf{F}\ f)\ \mathit{parameters}\quad \mathrm{where} \\
\quad \textbf{apply}\ \mathit{expr}\ \mathsf{Empty} \qquad\qquad\qquad\quad = \mathit{expr} \\
\quad \textbf{apply}\ \mathit{fun}\ \ (\mathit{bindings}\ :<\ \ a :\in\ _) = \textbf{apply}\ \mathit{fun}\ \mathit{bindings}\ :\$\ \mathsf{F}\ a
\end{array}$$

An example of this in action is the *generalization* functional. This takes a prefix and a binding, returning a transformed binding, abstracted over the prefix, together with the function which updates expressions accordingly.

$$\begin{array}{l}
\textbf{generalize}\ ::\ \mathsf{Prefix}\ \to\ \mathsf{Binding}\ \to\ (\mathsf{Binding}, \mathsf{Expr} \to \mathsf{Expr}) \\
\textbf{generalize}\ \mathit{bindings}\ (\mathit{me}\ :\in\ \ \mathit{expr})\ = \\
\quad (\mathit{me}\ :\in\ \ \mathit{bindings}\ \twoheadrightarrow\ \ \mathit{expr},\ \textbf{substitute}\ (\mathit{me}\ \ -\$\$\ \mathit{bindings})\ \mathit{me})
\end{array}$$

Indeed, working in a λ-calculus, these tools make it easy to implement λ-lifting [Joh85], and also the ‘raising’ step in Miller’s unification algorithm, working under a mixed prefix of existential and universal quantifiers [Mil92].

6 Example—inductive elimination operators for datatype families

We shall now use our tools to develop our example—constructing induction principles. To make things a little more challenging, and a little closer to home, let us consider the more general problem of constructing the inductive elimination operator for a *datatype family* [Dyb91].

Datatype families are collections of sets defined not parametrically as in Hindley-Milner languages, but by *mutual* induction, *indexed* over other data. They are the cornerstone of our dependently typed programming language, EPIGRAM [MM04]. We present them by first declaring the **type constructor**, explaining the indexing structure, and then the **data constructors**, explaining how larger elements of types in the family are built from smaller ones. A common example is the family of *vectors*—lists indexed by element type and *length*. In EPIGRAM, we would write:

$$\begin{array}{l}
\mathrm{data}\ \left(\dfrac{X\ :\ \star\ ;\ \ n\ :\ \mathsf{Nat}}{\mathsf{Vec}\ X\ n\ :\ \star}\right) \\
\mathrm{where}\ \left(\dfrac{}{\mathsf{Vnil}\ :\ \mathsf{Vec}\ X\ \mathsf{Zero}}\right)\ ;\ \left(\dfrac{x\ :\ X\ ;\ \ xs\ :\ \mathsf{Vec}\ X\ n}{\mathsf{Vcons}\ x\ xs\ :\ \mathsf{Vec}\ X\ (\mathsf{Suc}\ n)}\right)
\end{array}$$

That is, the Vnil constructor only makes *empty* vectors, whilst Vcons extends length by *exactly one*. This definition would elaborate (by a process rather like Hindley-Milner type inference) to a series of more explicit declarations in a language rather like that which we study in this paper:

$$\begin{aligned}\mathsf{Vec} &: \forall X \in \mathsf{Set}.\ \forall n \in \mathsf{Nat}.\ \mathsf{Set}\\ \mathsf{Vnil} &: \forall X \in \mathsf{Set}.\ \mathsf{Vec}\ X\ \mathsf{Zero}\\ \mathsf{Vcons} &: \forall X \in \mathsf{Set}.\ \forall n \in \mathsf{Nat}.\ \forall x \in X.\ \forall xs \in \mathsf{Vec}\ X\ n.\ \mathsf{Vec}\ X\ (\mathsf{Suc}\ n)\end{aligned}$$

The elimination operator for vectors takes three kinds of arguments: first, the *targets*—the vector to be eliminated, preceded by the indices of its type; second, the *motive*,[1] explaining what is to be achieved by the elimination; and third, the *methods*, explaining how the motive is to be pursued for each constructor in turn. Here it is, made fully explicit:

$$\begin{array}{l}\mathsf{Vec}\text{-}\mathbf{Ind} \in\\ \quad \left.\begin{array}{l}\forall X \in \mathsf{Set}.\\ \forall n \in \mathsf{Nat}.\\ \forall xs \in \mathsf{Vec}\ X\ n.\end{array}\right\} \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \mathbf{targets}\\ \quad \forall P \in \forall X \in \mathsf{Set}.\ \forall n \in \mathsf{Nat}.\ \forall xs \in \mathsf{Vec}\ X\ n.\ \mathsf{Set}. \qquad \qquad \qquad \mathbf{motive}\\ \quad \left.\begin{array}{l}\forall m_n \in \forall X \in \mathsf{Set}.\ P\ X\ \mathsf{Zero}\ (\mathsf{Vnil}\ X).\\ \forall m_c \in \forall X \in \mathsf{Set}.\ \forall n \in \mathsf{Nat}.\ \forall x \in X.\ \forall xs \in \mathsf{Vec}\ X\ n.\\ \qquad \qquad \forall h \in P\ X\ n\ xs.\ P\ X\ (\mathsf{Suc}\ X)\ (\mathsf{Vcons}\ X\ n\ x\ xs).\end{array}\right\} \quad \mathbf{methods}\\ \quad P\ X\ n\ xs\end{array}$$

It is not hard to appreciate that constructing such expressions using only strings for variables provides a legion of opportunities for accidental capture and abuse. On the other hand, the arithmetic involved in a purely de Bruijn indexed construction is truly terrifying. But with our tools, the construction is safe and sweatless.

To simplify the exposition, we shall presume that the declaration of the family takes the form of a binding for the type constructor and a context of data constructors which have already been checked for validity, say, according to the schema given by Luo [Luo94]—checking as we go just requires a little extra work and a shift to an appropriate monad. Luo's schema is a sound (but by no means complete) set of syntactic conditions on family declarations which guarantee the existence of a semantically meaningful induction principle. The relevant conditions and the corresponding constructions are

1. The type constructor is typed as follows

   $\mathsf{F} : \forall i_1 : I_1. \ldots \forall i_n : I_n.\ \mathsf{Set}$

   Correspondingly, the target prefix is $\forall \vec{i} : \vec{I}.\ \forall x : \mathsf{F}\ \vec{i}$, and the motive has type $P : \forall \vec{i} : \vec{I}.\ \forall x : \mathsf{F}\ \vec{i}.\ \mathsf{Set}$.

2. Each constructor has type

   $\mathsf{c} : \forall a_1 : A_1. \ldots \forall a_m. : A_m.\ \mathsf{F}\ s_1 \ldots s_n$

   where the $\vec{s}$ do not mention F. The corresponding method has type

   $\forall \vec{a} : \vec{A}.\ \forall \vec{h} : \vec{H}.\ P\ \vec{s}\ (\mathsf{c}\ \vec{a})$

   where the $\vec{H}$ are the inductive hypotheses, specified as follows.

3. Non-recursive constructor arguments $a : A$ do not mention F in A and contribute no inductive hypothesis.

[1] We prefer 'motive' [McB02] to 'induction predicate', because a motive need not be a predicate (i.e., a constructor of *propositions*) nor need an elimination operator be inductive.

4. Recursive constructor arguments have form

   $a : \forall y_1 : Y_1. \ldots \forall y_k : Y_k. \mathsf{F}\,\vec{r}$

   where F is not mentioned[2] in the $\vec{Y}$ or the $\vec{r}$. The corresponding inductive hypothesis is

   $h : \forall \vec{y} : \vec{Y}.\, P\,\vec{r}\,(a\,\vec{y})$

Observe that condition 4 allows for the inclusion of higher-order recursive arguments, parametrized by some $\vec{y} : \vec{Y}$. These support structures containing infinitary data, such as

$$\text{data } \mathsf{InfTree} : \star \text{ where } \mathsf{Leaf} : \mathsf{InfTree} \,;\;\; \mathsf{Node} : (\mathsf{Nat} \to \mathsf{InfTree}) \;\to\; \mathsf{InfTree}$$

We neglected to include these structures in our paper presentation of Epigram [MM04] because they would have reduced our light-to-heat ratio for no profit—we gave no examples which involved them. However, as you shall shortly see, they do not complicate the implementation in the slightest—the corresponding inductive hypothesis is parametrized by the same prefix.

Our agency for inductive elimination operators follows Luo's recipe directly. The basic outline is as follows:

$$
\begin{array}{l}
\textbf{makeInductiveElim} :: \mathsf{Agency}\;(\mathsf{Binding} \to \mathsf{Prefix} \to \mathsf{Binding}) \\
\textbf{makeInductiveElim}\; \mathit{me}\; (\mathit{family}\; :\in\; \mathit{famtype})\; \mathit{constructors}\; = \\
\quad \mathit{me} \;:\in\; \mathit{targets} \twoheadrightarrow \\
\qquad\qquad \mathit{motive} \longrightarrow \\
\qquad\qquad \textbf{fmap method}\; \mathit{constructors} \twoheadrightarrow \\
\qquad\qquad \textbf{name}\; \mathit{motive} -\$\$\; \mathit{targets} \\
\qquad\qquad\quad \text{where} \quad \text{— constructions from condition 1} \\
\qquad\qquad\qquad \mathsf{ForAll}\; \mathit{indices}\;\; \mathit{set}\,[\,] \;=\; \textbf{analysis}\; \mathit{me}\; \text{“i”}\; \mathit{famtype} \\
\qquad\qquad\qquad \mathit{targets} \;=\; \mathit{indices} \;\;:<\;\; \mathit{me}\,/\!/\; \text{“x”} \;:\in\; \mathit{family} -\$\$\, \mathit{indices} \\
\qquad\qquad\qquad \mathit{motive} \;=\; \mathit{me}\,/\!/\; \text{“P”} \;\;:\in\;\; \mathit{targets} \twoheadrightarrow \mathsf{F}\,(\textbf{nm}\; \text{“Set”}) \\
\qquad\qquad\qquad \textbf{method} :: \mathsf{Binding} \to \mathsf{Binding} \\
\qquad\qquad\qquad \ldots
\end{array}
$$

As we have seen before, **makeInductiveElim** is an agency which constructs a binding—the intended name of the elimination operator is used as the name of the agent. The **analysis** function readily extracts the indices from the type of the family (we presume that this ranges over Set). From here, we can construct the type of an element with those indices to compute the prefix of *targets* over which the *motive* is abstracted. Presuming we can construct an appropriate method for each constructor, we can now assemble our induction principle.

But how do we construct a method for a constructor? Let us implement the constructions corresponding to condition 2.

[2] This condition is known as *strict positivity.*

```
method :: Binding → Binding
method (con :∈ contype) =
  meth :∈ conargs ->>
          (conargs >>= indhyp) ->>
          var motive $$ conindices :$ (con -$$ conargs)
            where
              meth = me // "m" +< con
              ForAll conargs fam conindices = analysis meth "a" contype
              indhyp :: Binding → Prefix
              ...
```

The method's type says that the motive should hold for those targets which can possibly be built by the constructor, given the constructor's arguments, together with inductive hypotheses for those of its arguments which happen to be recursive. We can easily combine the hypothesis constructions for non-recursive and recursive arguments (3 and 4, above) by making Stack an instance of the MonadPlus class in exactly the same 'list of successes' style as we have for ordinary lists [Wad85]. The non-recursive constructor arguments give rise to an empty Prefix (= Stack Binding) of inductive hypothesis bindings.

```
indhyp :: Binding → Prefix
indhyp (arg :∈ argtype) = do
  guard (argfam == family)   -- yield Empty if arg is non-recursive
  return (arg // "h" :∈ argargs ->>
                         var motive $$ argindices :$ (arg $$ argargs))
    where ForAll argargs argfam argindices = analysis meth "y" argtype
```

With this, our construction is complete.

Epilogue

In this paper, we have shown how to manipulate syntax with binding using a mixed representation of names for free variables (with respect to the task in hand) and de Bruijn indices [dB72] for bound variables. By doing so, we retain the advantages of both representations: naming supports easy, arithmetic-free manipulation of terms; de Bruijn indices eliminate the need for α-conversion. Further, we have ensured that not only the user but also the *implementation* need never deal with de Bruijn indices, except within key basic operations such as **abstract** and **instantiate**.

Moreover, we have chosen a representation for names which readily supports a power structure naturally reflecting the structure of agents within the implementation. Name choice is safe and straightforward. Our technology combines easily with an approach to syntax manipulation inspired by Huet's 'zippers'[Hue97].

Without the technology in this paper, we could not have implemented Epigram [McB04]. Our example—constructing inductive elimination operators for datatype families—is but one of many where it proves invaluable. Others indeed include λ-lifting [Joh85] and Miller-style unification [Mil92]. More particularly, this technology evolved from our struggle to implement the 'elimination with a motive' approach [McB02], central to the elaboration of Epigram programs into Type Theory. This transforms a problem containing a *specific instance* of a datatype family

$$\forall \vec{s} : \vec{S}.\, \forall x : \mathsf{F}\, \vec{t}.\ T$$

into an equivalent problem which is immediately susceptible to elimination with operators like those constructed in our example.

$$\begin{array}{l} \forall \vec{i} : \vec{I}.\, \forall x' : \mathsf{F}\, \vec{i}. \\ \forall \vec{s} : \vec{S}.\, \forall x : \mathsf{F}\, \vec{t}.\ T. \\ \vec{i} = \vec{t} \to\ x' = x \to \\ T \end{array}$$

Whatever the syntax you may find yourself manipulating, and whether or not it involves dependent types, the techniques we have illustrated provide one way to make the job easier. By making computers using names the way *people* do, we hope you can accomplish such tasks straightforwardly, without becoming a prisoner of numbers.

References

[AR99] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda-terms using generalized inductive types. In *Computer Science Logic 1999*, 1999.

[BP99] Richard Bird and Ross Paterson. de Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–92, 1999.

[Coq91] Thierry Coquand. An algorithm for testing conversion in type theory. In Huet and Plotkin [HP91].

[dB72] Nicolas G. de Bruijn. Lambda Calculus notation with nameless dummies: a tool for automatic formula manipulation. *Indagationes Mathematicæ*, 34:381–392, 1972.

[dS83] Ferdinand de Saussure. *Course in General Linguistics*. Duckworth, 1983. English translation by Roy Harris.

[Dyb91] Peter Dybjer. Inductive Sets and Families in Martin-Löf's Type Theory. In Huet and Plotkin [HP91].

[Gen69] Gerhard Gentzen. *The collected papers of Gerhard Gentzen*. North-Holland, 1969. Edited by Manfred Szabo.

[HP91] Gérard Huet and Gordon Plotkin, editors. *Logical Frameworks*. CUP, 1991.

[Hue97] Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[Joh85] Thomas Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In Jouannaud [Jou85], pages 190–203.

[Jou85] Jean-Pierre Jouannaud, editor. *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*. Springer-Verlag, 1985.

[Kle52] S.C. Kleene. *Introduction to Metamathematics*. van Nostrand Rheinhold, Princeton, 1952.

[Luo94] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.

[McB98] Conor McBride. Inverting inductively defined relations in LEGO. In E. Giménez and C. Paulin-Mohring, editors, *Types for Proofs and Programs, '96*, volume 1512 of *LNCS*, pages 236–253. Springer-Verlag, 1998.

[McB02] Conor McBride. Elimination with a Motive. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs (Proceedings of the International Workshop, TYPES'00)*, volume 2277 of *LNCS*. Springer-Verlag, 2002.

[McB04] Conor McBride. Epigram, 2004. `http://www.dur.ac.uk/CARG/epigram`.

[Mil92] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, 1992.

[MM04] C. McBride and J. McKinna. The view from the left. *J. of Functional Programming*, 14(1), 2004.

[MP93] James McKinna and Robert Pollack. Pure type systems formalized. In Marc Bezem and Jan-Friso Groote, editors, *Int. Conf. Typed Lambda Calculi and Applications TLCA'93*, volume 664 of *LNCS*. Springer-Verlag, 1993.

[MP99] James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23:373–409, 1999. (Special Issue on Formal Proof, editors Gail Pieper and Frank Pfenning).

[Pra65] Dag Prawitz. *Natural Deduction—A proof theoretical study.* Almquist and Wiksell, Stockholm, 1965.

[Wad85] Philip Wadler. How to Replace Failure by a list of Successes. In Jouannaud [Jou85], pages 113–128.