Functional Pearl: I am not a Number—I am a Free Variable

Conor McBride
Department of Computer Science
University of Durham
South Road, Durham, DH1 3LE, England
c.t.mcbride@durham.ac.uk

James McKinna
School of Computer Science
University of St Andrews
North Haugh, St Andrews, KY16 9SS, Scotland
james.mckinna@st-andrews.ac.uk

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 [5] 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.

Moreover, we give a hierarchical representation for names which naturally reflects the structure of the operations we implement. Name choice is safe and straightforward. Our technology combines easily with an approach to syntax manipulation inspired by Huet's 'zippers'[10].

Without the ideas in this paper, we would have struggled to implement EPIGRAM [19]. Our example—constructing inductive elimination operators for datatype families—is but one of many where it proves invaluable.

Categories and Subject Descriptors

I.1.1 [**Symbolic and Algebraic Manipulation**]: Expressions and Their Representation; D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming

General Terms

Languages, Design, Reliability, Theory

Keywords

Abstract syntax, bound variables, de Bruijn representation, free variables, fresh names, Haskell, implementing Epigram, induction principles

Haskell'04, September 22, 2004, Snowbird, Utah, USA.

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 [19]. But even as we develop new technology to support programming and reasoning in advanced type systems, but we must handle the issues they raise 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 their trade, 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

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

We should like to synthesize some statement corresponding to

$$\begin{array}{l}\forall \mathtt{P} \in \mathtt{Nat} \to \mathtt{Prop}.\\ \mathtt{P\,Zero} \to\\ (\forall \mathtt{k} \in \mathtt{Nat}.\,\mathtt{P\,k} \to \mathtt{P}\,(\mathtt{Suc\,k})) \to\\ \forall \mathtt{n} \in \mathtt{Nat}.\,\mathtt{P\,n}\end{array}$$

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 (innocently or otherwise) may decide to declare

$$\mathtt{data\ Nat} = \mathtt{Zero} \mid \mathtt{P\ Nat} \quad \text{or even} \quad \mathtt{data\ P} = \mathtt{Zero} \mid \mathtt{Suc\ P}$$

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

$$\begin{array}{l}\forall \mathtt{P} \in \mathtt{Nat} \to \mathtt{Prop}.\\ \mathtt{P\,Zero} \to\\ (\forall \mathtt{k} \in \mathtt{Nat}.\,\mathtt{P\,k} \to \mathtt{P}\,(\mathtt{P\,k})) \to\\ \forall \mathtt{n} \in \mathtt{Nat}.\,\mathtt{P\,n}\end{array} \quad \text{or} \quad \begin{array}{l}\forall \mathtt{P} \in \mathtt{P} \to \mathtt{Prop}.\\ \mathtt{P\,Zero} \to\\ (\forall \mathtt{k} \in \mathtt{P}.\,\mathtt{P\,k} \to \mathtt{P}\,(\mathtt{Suc\,k})) \to\\ \forall \mathtt{n} \in \mathtt{P}.\,\mathtt{P\,n}\end{array}$$

Fear of shadows may seem trivial, but it's no joke—some real systems have this bug, although it would be invidious to name names.

Possible alternative strategies include the adoption of one of de Bruijn's systems of nameless dummies [5] for the local quantifiers, either counting binders (including $\rightarrow$, which we take to abbreviate $\forall$ where the bound variable isn't used) from the reference outward—de Bruijn **indices**,

$$\begin{array}{l}\forall - \in \mathtt{Nat} \rightarrow \mathtt{Prop}.\\ \mathtt{0}\,\mathtt{Zero} \rightarrow\\ (\forall - \in \mathtt{Nat}.\,\mathtt{2}\,\mathtt{0} \rightarrow \mathtt{3}\,(\mathtt{Suc}\,\mathtt{1})) \rightarrow\\ \forall - \in \mathtt{Nat}.\,\mathtt{3}\,\mathtt{0}\end{array}$$

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

$$\begin{array}{l}\forall \mathtt{0} \in \mathtt{Nat} \rightarrow \mathtt{Prop}.\\ \mathtt{0}\,\mathtt{Zero} \rightarrow\\ (\forall \mathtt{2} \in \mathtt{Nat}.\,\mathtt{0}\,\mathtt{2} \rightarrow \mathtt{0}\,(\mathtt{Suc}\,\mathtt{2})) \rightarrow\\ \forall \mathtt{3} \in \mathtt{Nat}.\,\mathtt{0}\,\mathtt{3}\end{array}$$

It's 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 enemies can't choose bad names in order to make trouble.

However, we do recommend that anyone planning to use de Bruijn syntax for systematic constructions like the above should think again. Performing constructions in either of these systems requires a lot of arithmetic. This obscures the idea being implemented, results in unreadable, unreliable, unmaintainable code, and is besides hard work. *We*, or rather *our programs*, can't 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 there's no 'social agreement' on a name.

The distinction between established linguistic signs, connecting a *signifiant* (or 'signifier') with its *signifié* (or 'signified'), and local signs, where the particular choice of signifier is arbitrary was observed in the context of natural language by Saussure [6]. In formal languages, the idea of distinguishing free and bound variables syntactically is also far from new. It's a recurrent idiom in the work of Gentzen [8], Kleene [14] and Prawitz [24]. The second author learned it from Randy Pollack who learned it in turn from Thierry Coquand [4]; the first author learned it from the second.

The idea of using free *names* and bound *indices* is not new either—it's a common representation in interactive proof systems. This also comes to the authors from Randy Pollack [23] who cites the influence of Gérard Huet in the Constructive Engine [9]. Here 'free' means 'bound globally in the context' and 'bound' means 'bound locally in the goal'. The distinction is allied 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's 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 [20, 21], although in that work, bound variables are distinguished from free variables but nonetheless named. We draw on the Huet's 'zipper' technique [10] to help us write programs which navigate and modify the structure of terms. Huet equips syntax with an auxiliary datatype of *structural* contexts. 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.

With many agents now engaged in the business of naming, we need a representation of names which readily supports the separation of namespaces between mechanical construction agents which call each other and indeed themselves. We adopt a hierarchical naming system which permits multiple agents to choose multiple fresh names in a notionally asynchronous manner, without fear of clashing. Our design choice 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.

Overview

In section 2 of this paper, we give the underlying data representation for our example syntax and develop the key operations which manipulate bound variables—only here do we perform arithmetic on de Bruijn indices, and that is limited to tracking the outermost index as we recurse under binders.

Section 3 shows the development of our basic construction and analysis operators for the syntax, and discusses navigation within expressions in the style of Huet [10]. Section 4 introduces our hierarchical technique for naming free variables in harmony with the call-hierarchy of agents which manipulate syntax.

These components come together in Section 5, where we assemble a high-level toolkit for constructions over our syntax. Section 6 puts this toolkit to work in a non-trivial example: the construction of induction principles for EPIGRAM's datatype families [7, 15, 19].

Acknowledgements

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`. Our influences are date back much further. We should like to thank all of our friends and colleagues who have encouraged us and fed us ideas through the years, in particular Gérard Huet and Thierry Coquand.

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 code we have both learned a great deal.

2 An Example Syntax

Today, let us have variables, application, and universal quantification. We choose an entirely first-order presentation:[1]

```
infixl 9 :$
infixr 6 :→
data Expr  =  F Name                    — free variables
           |  B Int                     — bound variables
           |  Expr :$ Expr              — application
           |  Expr :→ Scope             — ∀-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.

Some readers may be familiar with the use of nested datatypes and polymorphic recursion to enforce scope constraints precisely if you parametrize expressions by names [2, 3]. Indeed, with a dependently typed meta-language it's not so hard to enforce both scope and type for an object-language [1]. These advanced type systems can and should be used to give more precise types to the programs in this paper, but they would serve here only to distract readers not yet habituated to those systems from the implementation techniques which we seek to communicate here.

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** *name* turns a closed expression into a scope by turning *name* into B 0. Of course, as we push this operation under a binder, the correct index for *name* shifts along by one. That is, the image of *name* is always the *outer* de Bruijn index, hence we implement **abstract** via a helper function which tracks this value. Observe that the existing bound variables within *expr*'s Scopes remain untouched.

```
abstract :: Name → Expr → Scope
abstract name expr = Sc (nameTo 0 expr)   where
  nameTo outer (F name') | name == name' = B outer
                         | otherwise     = F name'
  nameTo outer (B index)                 = B index
  nameTo outer (fun :$ arg)              =
    nameTo outer fun :$ nameTo outer arg
  nameTo outer (dom :→ Sc body)          =
    nameTo outer dom :→ Sc (nameTo (outer + 1) body)
```

[1] The techniques in this paper adapt readily to higher-order representations of binding, but that's another story.

Meanwhile, **instantiate** *image* turns a scope into an expression by replacing the outer de Bruijn index (initially B 0) with *image*, which we presume is closed. Of course, F *name* is closed, so we can use **instantiate** (F *name*) to invert **abstract** *name*.

```
instantiate :: Expr → Scope → Expr
instantiate image (Sc body) = replace 0 body   where
  replace outer (B index) | index == outer = image
                          | otherwise      = B index
  replace outer (F name)                   = F name
  replace outer (fun :$ arg)               =
    replace outer fun :$ replace outer arg
  replace outer (dom :→ Sc body)           =
    replace outer dom :→ Sc (replace (outer + 1) body)
```

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

It is perfectly reasonable to develop these operations for other representations of bound variables, just as long as they're still kept separate from the free variables. A de Bruijn level representation still has the benefit of canonical name-choice and cheap α-equivalence, but it does mean that *image* must be shifted one level when we push it under a binder. Moreover, if we were willing to pay for α-equivalence and fresh-name generation for bound variables, we could even use names, modifying the definition of Scope to pack them up. We feel that, whether or not you want to *know* the names of bound variables, it's better to arrange things so you don't have to *care* about the names of bound variables.

Those with an eye for a generalization will have spotted that both **abstract** and **instantiate** can be expressed as instances of a single general-purpose higher-order substitution operation, parametrized by arbitrary operations on free and bound variables, themselves parametrized by *outer*.

```
varChanger :: (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. However, this operation is unsafe in the wrong hands.

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, outside of these operations, we maintain the invariant that Expr is only used for closed expressions and that Scopes have just one dangling index. The data constructors B and Sc have served their purpose—we forbid any further use of them. From now on, there are no de Bruijn numbers, only free variables.

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 name = instantiate image · abstract name
```

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 Basic Analysis and Construction Operators

We may readily define operators which attempt to analyse 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:

```
unapply :: MonadPlus m ⇒ Expr → m (Expr, Expr)
unapply (fun :$ arg) = return (fun, arg)
unapply _            = mzero
```

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.

```
infix 5 :∈
data Binding = Name :∈ Expr

bName :: Binding → Name
bName (name :∈ _) = name
bVar :: Binding → Expr
bVar = F · bName
```

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

```
infixr 6 ⟶
(⟶) :: Binding → Expr → Expr
(name :∈ dom) ⟶ range = dom :→ abstract name range

infix ⟵
(⟵) :: MonadPlus m ⇒ Name → Expr → m (Binding, Expr)
name ⟵ (dom :→ scope) = return (name :∈ dom,
                                instantiate (F name) scope)
name ⟵ _              = mzero
```

3.1 Inspiration—the 'Zipper'

We can give an account of one-hole contexts in the style of Huet's 'zippers' [10]. 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 $(\mathsf{Zipper}, \mathsf{Expr})$.

```
infixl 4 :<
data Stack x = Empty | Stack x :< x   deriving (Show, Eq)

type Zipper = Stack Step

data Step = Fun () Expr
          | Arg Expr ()
          | Dom () 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 filling the 'hole'. Note that we don't bind the variable when we edit a domain: it's 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 zipper constructed with respect to the raw definition of Expr—moving into scopes without binding variables—often requires a nightmare of arithmetic. The first author banged his head on his Master's project [16] this way, before the second author caught him at it.

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.

The machine must avoid choosing names already reserved by the user, whether or not those names have yet appeared. Moreover, as our programs decompose tasks into subtasks, we must avoid naming conflicts between the subprograms which address them. Indeed, we must avoid naming conflicts arising from different appeals to the *same* subprogram.

How do we achieve this? 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. This state-based approach fills names with meaningless numbers, and it unnecessarily sequentializes the execution of operations—a process cannot begin to generate names until its predecessors have finished doing so.

Our approach is familiar from the context of module systems or object-oriented programming. We control the anarchy of naming by introducing *hierarchical* names.

$$\text{type Name} = \text{Stack}(\text{String}, \text{Int})$$

We can use hierarchical names to reflect the hierarchy of tasks. We ensure that each subtask has a distinct prefix from which to form its names by extension. This directly rules out the possibility that different subtasks might choose the same name by accident and allows them to choose fresh names asynchronously. The remaining obligation—to ensure that each subtask makes distinct choices for the names under its *own* control—is easily discharged.

Superiority within the hierarchy of names is just the partial order induced by 'being a prefix':

$$xs \succcurlyeq (xs \lhd\!\!+ ys)$$

$$\begin{aligned}&\text{infixl } 4 \lhd\!\!+\\ &(\lhd\!\!+) :: \text{Stack}\, x \to \text{Stack}\, x \to \text{Stack}\, x\\ &xs \lhd\!\!+ \text{Empty} \quad = xs\\ &xs \lhd\!\!+ (ys :\!\prec y) = xs \lhd\!\!+ ys :\!\prec y\end{aligned}$$

We say that two names are **independent**, $xs \perp ys$, if neither $xs \succcurlyeq ys$ nor $ys \succcurlyeq xs$. Two independent names must differ at some leftmost point in the stack: whatever extensions we make of them, they will still differ at that point in the stack.

$$xs \perp ys \to (xs \lhd\!\!+ xs') \perp (ys \lhd\!\!+ ys')$$

In order to work correctly with hierarchical names, the remaining idea we need is to name the *agents* which carry out the tasks, as well as the free variables. Each agent must choose independent names not only for the free variables it creates, but also for the sub-agents it calls: this is readily accomplished by ensuring that every agent only ever chooses names which strictly and independently extend its own 'root' name. This ensures that the naming hierarchy of reflects the call-hierarchy of agents.

$$root \begin{cases} root\text{'s } \textbf{variables}:\\ root :\!\prec (\text{“x”}, 0),\ \ldots,\ root :\!\prec (\text{“x”}, m),\\ root :\!\prec (\text{“y”}, 0),\ \ldots,\ root :\!\prec (\text{“y”}, n),\\ \ldots\\ root\text{'s } \textbf{agents}:\\ \quad root :\!\prec (\text{“a”}, 0) \begin{cases} (root :\!\prec (\text{“a”}, 0))\text{'s } \textbf{variables}:\\ \quad root :\!\prec (\text{“a”}, 0) :\!\prec (\text{“x”}, 0),\ \ldots\\ (root :\!\prec (\text{“a”}, 0))\text{'s } \textbf{agents}:\\ \quad root :\!\prec (\text{“a”}, 0) :\!\prec (\text{“a”}, 0),\ \ldots \end{cases}\\ \quad \vdots\\ \quad root :\!\prec (\text{“a”}, k) \begin{cases} (root :\!\prec (\text{“a”}, k))\text{'s } \textbf{variables}:\\ \quad root :\!\prec (\text{“a”}, k) :\!\prec (\text{“x”}, 0),\ \ldots\\ (root :\!\prec (\text{“a”}, k))\text{'s } \textbf{agents}:\\ \quad root :\!\prec (\text{“a”}, k) :\!\prec (\text{“a”}, 0),\ \ldots \end{cases} \end{cases}$$

Note the convenience of $(\text{String}, \text{Int})$ as the type of name elements. The Strings give us legibility; the Ints an easy way to express uniform sequences of distinct name-extensions $x_0, \ldots x_n$. Two little helpers will make simple names easier to construct:

$$\begin{aligned}&\text{infixl } 6 \mathbin{/\!\!/}\\ &(\mathbin{/\!\!/}) :: \text{Name} \to \text{String} \to \text{Name}\\ &root \mathbin{/\!\!/} s = root :\!\prec (s, 0)\end{aligned}$$

$$\begin{aligned}&\textbf{nm} :: \text{String} \to \text{Name}\\ &\textbf{nm}\, s = \text{Empty} \mathbin{/\!\!/} s\end{aligned}$$

Our 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.

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

$$\text{type Agency } agentT = \text{Name} \to agentT$$

That is an Agency *agentT* takes a 'root' name to an agent of type *agentT* with that name.

You've already seen an agency—the under-binding navigator, which may be retyped

$$\begin{aligned}&\text{infix } \longleftarrow\\ &(\longleftarrow) :: \text{MonadPlus}\, m \Rightarrow\\ &\qquad \text{Agency}\ (\text{Expr} \to m\, (\text{Binding}, \text{Expr}))\end{aligned}$$

That is, $(root \longleftarrow)$ is the agent which binds *root* by decomposing a quantifier. Note that here the agent which creates the binding shares its name: the variable means 'the thing made by the agent', so this arrangement is quite convenient. It fits directly with our standard practice of using 'metavariables' to stand for the unknown parts of a construction, each associated with an agent trying to deduce its value.

5 A Higher-Level 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 $\longrightarrow$.

$$\text{type Prefix} = \text{Stack Binding}$$

$$\begin{aligned}&\text{infixr } 6 \twoheadrightarrow\\ &(\twoheadrightarrow) :: \text{Prefix} \to \text{Expr} \to \text{Expr}\\ &\text{Empty} \qquad\quad \twoheadrightarrow expr \quad = expr\\ &(binds :\!\prec bind) \twoheadrightarrow range = binds \twoheadrightarrow bind \longrightarrow range\end{aligned}$$

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

$$\begin{aligned}&\textbf{unprefix} :: \text{Agency}\ (\text{String} \to \text{Expr} \to (\text{Prefix}, \text{Expr}))\\ &\textbf{unprefix}\ root\ x\ expr = \textbf{intro}\, 1\, (\text{Empty}, expr) \quad \text{where}\\ &\quad \textbf{intro} :: \text{Int} \to (\text{Prefix}, \text{Expr}) \to (\text{Prefix}, \text{Expr})\\ &\quad \textbf{intro}\ \_i\ (binds, expr) = \text{case}\ (root :\!\prec (x, \_i)) \longleftarrow expr\ \text{of}\\ &\qquad \text{Just}\, (bind, range) \to \textbf{intro}\ (\_i + 1)\ (binds :\!\prec bind, range)\\ &\qquad \text{Nothing} \qquad\quad \to (binds, expr)\end{aligned}$$

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

If *root* is independent of all the names in *expr*—which it will be, if we maintain our hierarchical discipline—and

$$\textbf{unprefix}\ root\ x\ expr\ =\ (binds, range)$$

then *range* is unquantified and $expr = binds \twoheadrightarrow range$.

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.

```
weaken :: Agency (Expr → Expr → Expr)
weaken root dom expr =
  xdoms ↠ (root // "y" :∈ dom) ⟶ range
  where (xdoms, range) = unprefix root "x" expr
```

As ever, the independence of the root 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 $$.

```
infixl 9 $$
($$) :: Expr → [Expr] → Expr
expr $$ []           = expr
fun  $$ (arg : args) = fun :$ arg $$ args
```

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

```
unapplies :: Expr → (Expr, [Expr])
unapplies expr = peel (expr, [])  where
  peel (fun :$ arg, args) = peel (fun, arg : args)
  peel funargs            = funargs
```

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 applications of quantified formulae, and these are meaningless. 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:

```
data Analysis = ForAll Prefix Name [Expr]

analysis :: Agency (String → Expr → Analysis)
analysis root x expr = ForAll prefix f args  where
  (prefix, range) = unprefix root x expr
  (F f, args)     = unapplies range
```

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:

```
infixl 9 –$$
(–$$) :: Name → Prefix → Expr
f –$$ parameters = apply (F f) parameters  where
  apply expr Empty              = expr
  apply fun  (binds :< a :∈ _) = apply fun binds :$ F a
```

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.

```
generalize :: Prefix → Binding → (Binding, Expr → Expr)
generalize binds (name :∈ expr) =
  (me :∈ binds ↠ expr, substitute (name –$$ binds) name)
```

Indeed, working in a λ-calculus, these tools make it easy to implement λ-lifting [12], and also the 'raising' step in Miller's unification algorithm, working under a mixed prefix of existential and universal quantifiers [22].

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* [7].

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 [19]. 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:

$$\text{data}\ \left(\frac{X : \star;\ n : \mathsf{Nat}}{\mathsf{Vec}\, X\, n : \star}\right)$$
$$\text{where}\ \left(\frac{}{\mathsf{Vnil} : \mathsf{Vec}\, X\, \mathsf{Zero}}\right);\ \left(\frac{x : X;\ xs : \mathsf{Vec}\, X\, n}{\mathsf{Vcons}\, x\, xs : \mathsf{Vec}\, X\, (\mathsf{Suc}\, n)}\right)$$

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 in-

dices of its type; second, the *motive*,[2] 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:

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

It is not hard to appreciate that constructing such expressions using only strings for variables provides a legion of opportunities for unlawful 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 straightforward and safe..

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 [15]—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 $\forall \vec{\imath} : \vec{I}.\, \forall x : \mathsf{F}\,\vec{\imath}$, and the motive has type $P : \forall \vec{\imath} : \vec{I}.\, \forall x : \mathsf{F}\,\vec{\imath}.\, \mathsf{Set}$.

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.

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[3] 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

```
data InfTree : ⋆ where Leaf : InfTree
                       Node : (Nat → InfTree) → InfTree
```

We neglected to include these structures in our paper presentation of EPIGRAM [19] 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 $\vec{y} : \vec{Y}$.

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

```
makeIndElim :: Agency (Binding → Prefix → Binding)
makeIndElim root (family :∈ famtype) constructors =
  root :∈ targets -:>>
          motive -:>
          fmap method constructors -:>>
          bName motive -$$ targets
            where  — constructions from condition 1
              ForAll indices set [] =
                analysis root "i" famtype
              targets = indices :<
                        root // "x" :∈ family -$$ indices
              motive = root // "P" :∈ targets -:>>
                                      F (nm "Set")
              method :: Binding → Binding
              ...
```

As we have seen before, **makeIndElim** 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 make the type of an element with those indices, and hence compute the prefix of *targets* over which the *motive* is abstracted. Presuming we can build an appropriate method for each constructor, we can now assemble our induction principle.

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

```
method :: Binding → Binding
method (con :∈ contype) =
  meth :∈ conargs -:>>
          (indhyp =:<< conargs) -:>>
          bVar motive $$ conindices :$ (con -$$ conargs)
            where
              meth = root // "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

[2] We prefer 'motive' [17] to 'induction predicate', because a motive need not be a predicate (i.e., a constructor of *propositions*) nor need an elimination operator be inductive.

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

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 [25]. 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)   — no hyp if arg non-recursive
  return (arg // "h" :∈ argargs →»
                    bVar 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 [5] 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'[10].

Of course, it takes some effort to ensure that name-roots are propagated correctly through the call hierarchy of a large system. We can manage the details of this in practice by working within an appropriate monad. The monad which we use also manages the bookkeeping for the recursive solution of metavariables by expressions in terms of other metavariables (whose names are extensions of the original)—this process is beyond the scope of this paper.

Without the ideas in this paper (amongst many others) it would have been much more difficult to implement EPIGRAM [18]. Our example—constructing inductive elimination operators for datatype families—is but one of many where it proves invaluable. Others indeed include λ-lifting [12] and Miller-style unification [22].

More particularly, this technology evolved from our struggle to implement the 'elimination with a motive' approach [17], 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.

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

Moreover, EPIGRAM source code is edited and elaborated into an underlying type theory incrementally, in no fixed order and with considerable dependency between components. The elaboration process is, in effect, code-driven tactical theorem-proving working on multiple interrelated problems simultaneously. Our principled approach to manipulating abstract syntax within multiple agents provides the key discipline we need in order to manage this process easily. We simply could not afford to leave these issues unanalysed.

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, and without becoming a prisoner of numbers.

7 References

[1] T. Altenkirch and B. Reus. Monadic presentations of lambda-terms using generalized inductive types. In *Computer Science Logic 1999*, 1999.

[2] F. Bellegarde and J. Hook. Substitution: A formal methods case study using monads and transformations. *Science of Computer Programming*, 1995.

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

[4] T. Coquand. An algorithm for testing conversion in type theory. In Huet and Plotkin [11].

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

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

[7] P. Dybjer. Inductive Sets and Families in Martin-Löf's Type Theory. In Huet and Plotkin [11].

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

[9] G. Huet. The Constructive Engine. In R. Narasimhan, editor, *A Perspective in Theoretical Computer Science*. World Scientific Publishing, 1989. Commemorative Volume for Gift Siromoney.

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

[11] G. Huet and G. Plotkin, editors. *Logical Frameworks*. CUP, 1991.

[12] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In Jouannaud [13], pages 190–203.

[13] J.-P. Jouannaud, editor. *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*. Springer-Verlag, 1985.

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

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

[16] C. 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.

[17] C. McBride. Elimination with a Motive. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs (Proceedings of the International Workshop, TYPES'00)*, volume 2277 of *LNCS*. Springer-Verlag, 2002.

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

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

[20] J. McKinna and R. Pollack. Pure type systems formalized. In M. Bezem and J.-F. Groote, editors, *Int. Conf. Typed Lambda Calculi and Applications TLCA'93*, volume 664 of *LNCS*. Springer-Verlag, 1993.

[21] J. McKinna and R. 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).

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

[23] R. Pollack. Closure under alpha-conversion. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, LNCS 806, pages 313–332. Springer-Verlag, 1994. Selected papers from the Int. Workshop TYPES '93, Nijmegen, May 1993.

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

[25] P. Wadler. How to Replace Failure by a list of Successes. In Jouannaud [13], pages 113–128.