

de Bruijn notation as a nested datatype

RICHARD S. BIRD

*Programming Research Group, Oxford University,
Wolfson Building, Parks Road, Oxford OX1 3QD, UK*

ROSS PATERSON

*Department of Computer Science, City University,
Northampton Square, London EC1V 0HB, UK*

“I have no data yet. It is a capital mistake to
theorise before one has data.”

Sir Arthur Conan Doyle

The Adventures of Sherlock Holmes

Abstract

de Bruijn notation is a coding of lambda terms in which each occurrence of a bound variable x is replaced by a natural number, indicating the ‘distance’ from the occurrence to the abstraction that introduced x . One might suppose that in any datatype for representing de Bruijn terms, the distance restriction on numbers would have to be maintained as an explicit datatype invariant. However, by using a nested (or non-regular) datatype, we can define a representation in which *all* terms are well-formed, so that the invariant is enforced automatically by the type system. Programming with nested types is only a little more difficult than programming with regular types, provided we stick to well-established structuring techniques. These involve expressing inductively defined functions in terms of an appropriate fold function for the type, and using fusion laws to establish their properties. In particular, the definition of lambda abstraction and beta reduction is particularly simple, and the proof of their associated properties is entirely mechanical.

Capsule Review

Many functional languages (certainly ML and Haskell) allow *nested* data types, a recursive data type in which the recursive occurrence on the right hand side of the definition is applied to different arguments than the left hand side. (The Introduction of the paper elaborates.) But such types are almost unusable unless the language supports *polymorphic recursion*, in which a function can call itself recursively at a different type to the “parent” call. Abstracting such functions into folds (as is commonly done with ordinary recursive functions) requires the fold to take a *polymorphic function as its argument*. Finally, it turns out to be essential to *abstract over type constructors*, not only over types.

This fascinating paper shows that if these extensions are supported, then nested data types become genuinely useful. Though a particular example (de Bruijn notation), Bird and Paterson show that nested data types can express and statically enforce useful invariants – which is, of course, what types systems are for.

In short, this paper demonstrates by example that some relatively modest extensions to the Hindley-Milner type system have practical utility. Whether this is an isolated example, or a member of a large and compelling class of applications, remains to be seen.

1 Introduction

A standard representation of lambda terms, with variables of type v , in Haskell involves essentially the following datatype:

$$\text{data Term } v = \text{Var } v \mid \text{App } (\text{Term } v, \text{Term } v) \mid \text{Lam } v (\text{Term } v)$$

The problem with the standard representation is that while abstraction is easy to implement, application is not. Application of a lambda term $\text{Lam } x b$ to an argument t involves substituting t for all free occurrences of x in b . Care has to be taken to avoid the capture of free variables in t by bound variables in b . To overcome this problem, de Bruijn (1972) proposed a notation for lambda expressions in which bound variables do not occur. In his notation, no variable appears after the constructor Lam , and bound variables appear as natural numbers. The number assigned to an occurrence of a bound variable x is the depth of nesting of Lam terms between that occurrence and the (closest) binding occurrence of x . For example,

$$\lambda x.x (\lambda y.x y (\lambda z.x y z))$$

translates to

$$\lambda.0 (\lambda.1 0 (\lambda.2 1 0))$$

This example is taken from Paulson (1996), which discusses de Bruijn notation in detail.

If one wants to represent lambda terms involving both bound and free variables in the de Bruijn style, then the declaration of $\text{Term } v$ has to be changed. One possibility, used in Paulson (1996), is to have two kinds of variable: free variables drawn from v , and bound variables drawn from Int . Another possibility is to use a datatype declaration

$$\begin{aligned} \text{data Term } v &= \text{Var } v \mid \text{App } (\text{Term } v, \text{Term } v) \mid \text{Lam } (\text{Term } (\text{Incr } v)) \\ \text{data Incr } v &= \text{Zero} \mid \text{Succ } v \end{aligned}$$

In the body of a lambda abstraction, the set of variables is augmented with an extra element, the variable bound by the lambda. This variable is denoted by Zero ; each free variable x is renamed $\text{Succ } x$ inside the lambda. For example, the terms $\lambda x.x$ and $\lambda x.\lambda y.x$ are represented as

$$\text{Lam } (\text{Var } \text{Zero}) \text{ and } \text{Lam } (\text{Lam } (\text{Var } (\text{Succ } \text{Zero})))$$

The term $\lambda x.\lambda y.x y z$, containing a free variable z , may be represented as the following element of Term Char :

$$\text{Lam } (\text{Lam } (\text{App } (\text{App } (\text{Var } (\text{Succ } \text{Zero}), \text{Var } \text{Zero}), \text{Var } (\text{Succ } (\text{Succ } 'z')))))$$

The type Term is an example of a *nested* datatype (Bird and Meertens, 1998) because its definition has a recursive use with a different argument from the left-hand side. Such definitions are also sometimes called *non-regular*.

Our aim in this paper is to study this novel representation of lambda terms, and to give the implementations of abstraction and application. Useful and interesting examples of nested datatypes have been rather thin on the ground until recently,

and de Bruijn notation gives us an excellent opportunity to explore the theory in the context of a specific example. We believe that the right way to proceed into the largely uncharted territory of nested types is to stick to the structuring principles provided by the now well-established theory of regular datatypes. This theory is reviewed briefly in section 2. In section 3 we introduce the type of lambda terms, and set up appropriate machinery for defining functions over this type. The implementations of abstraction and application are given in section 4. In the final section, we will generalise what we have learnt to cover an extension of de Bruijn's notation.

Another aim of the paper concerns proof. In our view, equational properties of functions are most easily proved when functions are defined as combinations of other functions, using functional composition rather than application as the primary combining form. As a consequence, proof by induction is replaced by appeal to general equational laws that make up standard theory. This material is also reviewed briefly in section 2. Proofs of the various equations were generated using the simple automatic calculator described in Bird (1998); we include a selection of them.

All programs in typewriter font are expressed in Hugs 1.3c (Jones, 1998), an extension of Haskell that provides a more flexible typing discipline.

2 Preliminaries

Let us begin, not with *Term*, but with the simpler inductive datatype of binary trees (which is equivalent to *Term* without the difficult *Lam* case):

```
data BinTree a = Leaf a | Fork (Pair (BinTree a))
type Pair a = (a,a)
```

By default, Haskell allows *Leaf* and *Fork* to be non-strict functions, so the declaration above captures partial and infinite trees as well as finite ones. However, although all functions defined in this paper are legal Haskell (extended with a more general typing discipline), we are only concerned with datatypes that are flat sets, and functions that are total in the set-theoretic sense. Thus, all functions are considered to be strict, as in ML. This will enable us to state equational laws without mentioning strictness conditions explicitly.

2.1 Functors

For each datatype constructor¹, there is a corresponding action on functions, which preserves the shape of a data structure while replacing elements within it. The classic example is the *map* function on lists, and functional programmers call these actions *mapping functions*. For the type constructor *Pair*, the mapping function is

```
mapP :: (a -> b) -> Pair a -> Pair b
mapP f (x, y) = (f x, f y)
```

¹ We do not consider type constructors that include function types.

The mapping function on binary trees is:

```
mapB :: (a -> b) -> BinTree a -> BinTree b
mapB f (Leaf x) = (Leaf . f) x
mapB f (Fork p) = (Fork . mapP (mapB f)) p
```

The slightly unusual form of the right-hand sides is intended to suggest the function-level equations

$$\text{mapB } f \cdot \text{Leaf} = \text{Leaf} \cdot f \quad (1)$$

$$\text{mapB } f \cdot \text{Fork} = \text{Fork} \cdot \text{mapP } (\text{mapB } f) \quad (2)$$

Category theorists refer to the combination of type constructor and map function as a *functor*. Hence the following laws, satisfied by any mapping function, are called *functor laws*:

$$\text{mapB } id = id \quad (3)$$

$$\text{mapB } (f \cdot g) = \text{mapB } f \cdot \text{mapB } g \quad (4)$$

A further property, called *naturality*, plays an important role in many calculations. A polymorphic function $f :: M a \rightarrow N a$, where M and N are given type constructors, may be viewed as a collection of functions, one for each instantiation of the type variable a . Because f is polymorphic, i.e. defined independently of a , these instances are related by the following naturality condition:

$$\text{mapN } k \cdot f = f \cdot \text{mapM } k \quad (5)$$

for all functions k , where mapM and mapN are the map functions for the type constructors M and N , respectively. Such functions f are called *natural transformations*.

As one example, any function of type

```
flatten :: BinTree a -> [a]
```

is a natural transformation, with naturality property

$$\text{map } f \cdot \text{flatten} = \text{flatten} \cdot \text{mapB } f \quad (6)$$

Similarly, the naturality of the *BinTree* constructors $\text{Leaf} :: a \rightarrow \text{BinTree } a$ and $\text{Fork} :: \text{Pair } (\text{BinTree } a) \rightarrow \text{BinTree } a$ is expressed by equations (1) and (2), which define the action mapB . Note that the action on functions corresponding to the identity type constructor is the identity, and a composition of type constructors corresponds to a composition of actions.

2.2 Folds

The second general operator generalises the *foldr* function on lists. For binary trees, the operator is

```
foldB :: (a -> b) -> (Pair b -> b) -> BinTree a -> b
foldB l f (Leaf x) = l x
foldB l f (Fork p) = (f . mapP (foldB l f)) p
```

The fold operator takes a function argument for each constructor of the datatype. Its action is to replace the constructors in its input with the corresponding functions. Often the effect is to reduce the data structure to a summary value, as in the first two of the following examples:

```
size    = foldB (const 1) (uncurry (+))
height  = foldB (const 0) maxp
         where maxp (x, y) = 1 + max x y
flatten = foldB wrap (uncurry (++))
         where wrap x = [x]
```

A fundamental property of all fold operators is that they produce the unique function satisfying the above defining equations. From this follows a trio of useful calculational laws. The simplest is the *identity law*, which for binary trees is

$$\text{foldB Leaf Fork} = \text{id} \quad (7)$$

The other two laws are more powerful, and heavily used in calculations. The *fusion law* states that

$$h \cdot \text{foldB } l f = \text{foldB } l' f' \iff \begin{cases} h \cdot l = l' \\ h \cdot f = f' \cdot \text{mapP } h \end{cases} \quad (8)$$

The *map-fusion law* states that

$$\text{foldB } l f \cdot \text{mapB } h = \text{foldB } (l \cdot h) f \quad (9)$$

An immediate consequence of map-fusion and the identity law is an alternative definition of *mapB* as a fold:

$$\text{mapB } h = \text{foldB } (\text{Leaf} \cdot h) \text{Fork} \quad (10)$$

The map operator for each regular datatype may be defined as fold in this way, but this does not hold for nested datatypes.

Fusion laws, functor properties, and naturality conditions, are all we need for a powerful generic equational theory of inductive datatypes. For further details, see Bird and de Moor (1997).

2.3 Monads

Monad operations provide a useful way of structuring many programs. Functional programmers are introduced to monads as a type constructor with a certain binding operation. Category theorists use a function-level definition, which is also more convenient for calculations. A monad is defined as a type constructor M with a mapping function mapM and two operations

$$\begin{aligned} \text{unit} &:: a \rightarrow M a \\ \text{join} &:: M (M a) \rightarrow M a \end{aligned}$$

These natural transformations are required to satisfy the following coherence laws:

$$\text{join} \cdot \text{mapM unit} = \text{id} \quad (11)$$

$$\text{join} \cdot \text{unit} = \text{id} \quad (12)$$

$$\text{join} \cdot \text{mapM join} = \text{join} \cdot \text{join} \quad (13)$$

In total, there are seven laws available for reasoning about a monad: the three coherence laws, the two naturality laws for *unit* and *join*, and the two functor laws for *mapM*, the mapping function associated with *M*.

A standard example of a monad is the list type constructor, with *unit* returning a singleton list, and *concat* as the join operation. Binary trees also form a monad, with unit *Leaf* and the following join function:

```
joinB :: BinTree (BinTree a) -> BinTree a
joinB = foldB id Fork
```

As we will see, lambda terms also form a monad; the unit and join operations on lambda terms will be needed in the definition of lambda abstraction and application. See Bird (1998) for further discussion of monads and monad laws, and the different ways one can describe them.

3 de Bruijn notation

We can follow the same steps with the type *Term a* of lambda terms over a type *a*:

```
data Term v = Var v | App (Pair (Term v)) | Lam (Term (Incr v))
data Incr v = Zero | Succ v
```

3.1 Maps

The first step is to identify the map operators for the newly introduced types. The mapping function corresponding to *Incr* is straightforward:

```
mapI :: (a -> b) -> Incr a -> Incr b
mapI f Zero = Zero
mapI f (Succ x) = (Succ . f) x
```

As we might expect, *Term* is more interesting:

```
mapT :: (a -> b) -> Term a -> Term b
mapT f (Var x) = (Var . f) x
mapT f (App p) = (App . mapP (mapT f)) p
mapT f (Lam t) = (Lam . mapT (mapI f)) t
```

Note the change of argument of *mapT* in the *Lam* case: the required mapping function for *Term (Incr a)* is *mapT (mapI f)*. As a result, *mapT* leaves bound variables unchanged, and replaces only free variables. In the nested definition, bound variables have become part of the shape of a term.

Note also that the argument of *mapT* in the *Lam* case also has a different type, namely *Incr a* \rightarrow *Incr b*, but this is an instance of the declared signature. The definition of *mapT* makes use of *polymorphic recursion*; it is the first function in this paper whose type signature cannot be omitted.

3.2 Folds

The definition of the fold function for *Term* follows from the principle of replacing constructors by functions:

```
foldT v a l (Var x) = v x
foldT v a l (App p) = (a . mapP (foldT v a l)) p
foldT v a l (Lam t) = (l . foldT v a l) t
```

Unfortunately, the last line of this definition will not pass a standard Haskell typechecker: if *foldT val* is applied to a term of type *Term V* for some type *V*, then the *foldT val* on the right side is applied to a term of type *Term (Incr V)*. Hence the argument functions *v*, *a* and *l* must be applicable at a range of different types; effectively, they must be polymorphic. Haskell's language of types cannot express this without an extension called *rank-2* type signatures (McCracken, 1984). Such signatures have been implemented in GHC and also in Hugs 1.3c (Peyton Jones *et al.*, 1998; Jones, 1998). In the syntax of Hugs 1.3c, *foldT* can be made acceptable by adding the following type signature:

```
foldT :: (forall a. a -> n a) ->
        (forall a. Pair (n a) -> n a) ->
        (forall a. n (Incr a) -> n a) ->
        Term b -> n b
```

Here the variable *n* denotes an arbitrary type constructor.

As a consequence of the arguments being natural transformations, *foldT val* is a natural transformation, with associated property

$$\text{mapN } k \cdot \text{foldT } val = \text{foldT } val \cdot \text{mapT } k \quad (14)$$

The naturality law of *foldT* does not hold for regular datatypes, such as binary trees or lists, because the argument of the fold is not required to be natural.

The above naturality condition implies that no instance of *foldT* can manipulate the values of free variables. As a result, we cannot define all the functions we would like on terms as instances of *foldT*. This phenomenon motivates a more general definition of the fold operator on nested datatypes such as *Term*; we will call it *gfold* for *generalised* fold:

```
gfoldT :: (forall a. m a -> n a) ->
        (forall a. Pair (n a) -> n a) ->
        (forall a. n (Incr a) -> n a) ->
        (forall a. Incr (m a) -> m (Incr a)) ->
        Term (m b) -> n b
gfoldT v a l k (Var x) = v x
gfoldT v a l k (App p) = (a . mapP (gfoldT v a l k)) p
gfoldT v a l k (Lam t) = (l . gfoldT v a l k . mapT k) t
```

The two additional ingredients in the definition of *gfoldT* are, firstly, that the argument of *v* is generalised from *a* to *ma* for an arbitrary type constructor *m* and,

secondly, that an extra argument k is provided for the fold. To explain the role of the extra function k , observe that a lambda term with variables drawn from mb has type

$$\text{Term } (\text{Incr } (m b))$$

Applying $\text{mapT } k$ to this lambda term produces an element of type

$$\text{Term } (m (\text{Incr } b))$$

Applying $\text{gfoldT } v a l k$ to this element produces an element of type

$$n (\text{Incr } b)$$

This is the correct type for an argument of l . More details of generalised folds and their properties may be found in a companion paper (Bird and Paterson, 1998).

The arguments to gfoldT are natural transformations, and the result is also a natural transformation. Thus, if $\text{gfoldT } v a l k :: \text{Term } (M b) \rightarrow N b$, we have

$$\text{mapN } k \cdot \text{gfoldT } v a l k = \text{gfoldT } v a l k \cdot \text{mapT } (\text{mapM } k)$$

for all k , where mapM and mapN are the mapping functions associated with the type constructors M and N .

The advantage of the generalised fold resides in the extra degree of freedom for selecting the type constructor m . In theory, we can take $m = \text{Id}$, the identity type constructor, and so obtain

$$\text{foldT } v a l = \text{gfoldT } v a l i d \tag{15}$$

as a special case. Thus gfoldT generalises foldT . Another instance of gfoldT takes both m and n to be constant type constructors, delivering specific types for all arguments. However, type constructor polymorphism in Haskell is limited, in that type constructor variables may only be instantiated to *datatype* constructors (possibly partially applied). The alternative to expressing these special cases by installing Id and Const as new datatype constructors is to define specialised versions of gfoldT . For example, the following version corresponds to the constant type constructors case:

```
kfoldT :: (a -> b) -> (Pair b -> b) -> (b -> b) ->
  (Incr a -> a) ->
  Term a -> b
kfoldT v a l k (Var x) = v x
kfoldT v a l k (App p) = (a . mapP (kfoldT v a l k)) p
kfoldT v a l k (Lam t) = (l . kfoldT v a l k . mapT k) t
```

Note that kfoldT has exactly the same definition as gfoldT , but a different (more specific) type. For example, we can convert a lambda term to a string by

```
showT :: Term String -> String
showT = kfoldT id showP ('L':) showI
  where showP (x,y) = "(" ++ x ++ " " ++ y ++ ")"
        showI Zero = "0"
        showI (Succ x) = 'S':x
```


In particular, we can use *showT* to convert an element of type *Term Char* to a string in which individual character variables are printed without their quotes:

```
showTC :: Term Char -> String
showTC = showT . mapT wrap
  where wrap x = [x]
```

For example, applying *showTC* to

$$\text{Lam (App (Var Zero, App (Var (Succ 'x'), Var (Succ 'y'))))}$$

produces the string `L(0(x'y'))`.

The function *gfoldT* satisfies similar fusion laws to those discussed above for binary trees. Such laws are proved from the fact that *gfoldT* is the unique function satisfying its defining equation. (This can be established by induction over terms.)

In particular, the identity law states that

$$\text{gfoldT Var App Lam id} = \text{id} \quad (16)$$

The map-fusion law states that

$$\text{gfoldT v a l k} \cdot \text{mapT h} = \text{gfoldT (v \cdot h) a l k'} \iff \text{k} \cdot \text{mapI h} = \text{h} \cdot \text{k'}$$

The fold-fusion law is the following: suppose we have the typing

$$\text{gfoldT v a l k} \quad :: \quad \text{Term (M a)} \rightarrow \text{N a}$$

Then

$$\begin{aligned} \text{h} \cdot \text{gfoldT v a l k} &= \text{gfoldT v' a' l' (mapM k' \cdot k)} \\ &\iff \begin{cases} \text{h} \cdot \text{v} = \text{v'} \\ \text{h} \cdot \text{a} = \text{a'} \cdot \text{mapP h} \\ \text{h} \cdot \text{l} = \text{l'} \cdot \text{h} \cdot \text{mapN k'} \end{cases} \end{aligned} \quad (17)$$

The proof consists of simple calculations to show that $\text{h} \cdot \text{gfoldT v a l k}$ satisfies the defining equations of $\text{gfoldT v' a' l' (mapM k' \cdot k)}$. The *Lam* case is the longest:

$$\begin{aligned} &\text{h} \cdot \text{gfoldT v a l k} \cdot \text{Lam} \\ &= \quad \{\text{definition of gfoldT}\} \\ &\text{h} \cdot \text{l} \cdot \text{gfoldT v a l k} \cdot \text{mapT k} \\ &= \quad \{\text{assumption}\} \\ &\text{l'} \cdot \text{h} \cdot \text{mapN k'} \cdot \text{gfoldT v a l k} \cdot \text{mapT k} \\ &= \quad \{\text{naturality}\} \\ &\text{l'} \cdot \text{h} \cdot \text{gfoldT v a l k} \cdot \text{mapT (mapM k')} \cdot \text{mapT k} \\ &= \quad \{\text{functor}\} \\ &\text{l'} \cdot \text{h} \cdot \text{gfoldT v a l k} \cdot \text{mapT (mapM k' \cdot k)} \end{aligned}$$

3.3 A monad

The type constructor *Term* is also a monad, with *Var* as the unit operator, and *joinT* defined by

```

joinT :: Term (Term a) -> Term a
joinT = gfoldT id App Lam distT

distT :: Incr (Term a) -> Term (Incr a)
distT Zero = Var Zero
distT (Succ x) = mapT Succ x

```

The function *distT* replaces *Succs* on terms by *Succs* on variables. It satisfies the following properties², easily established by cases:

$$\text{distT} \cdot \text{mapI Var} = \text{Var} \quad (18)$$

$$\text{distT} \cdot \text{mapI joinT} = \text{joinT} \cdot \text{mapT distT} \cdot \text{distT} \quad (19)$$

Using these equations, and the fusion laws for *gfoldT*, we can prove the coherence laws for the monad operations on *Term*:

$$\text{joinT} \cdot \text{Var} = \text{id} \quad (20)$$

$$\text{joinT} \cdot \text{mapT Var} = \text{id} \quad (21)$$

$$\text{joinT} \cdot \text{mapT joinT} = \text{joinT} \cdot \text{joinT} \quad (22)$$

For example, we give the proof of equation (22):

$$\begin{aligned}
& \text{joinT} \cdot \text{mapT joinT} \\
= & \quad \{\text{definition of joinT}\} \\
& \text{gfoldT id App Lam distT} \cdot \text{mapT joinT} \\
= & \quad \{\text{map fusion, by distribution law (19)}\} \\
& \text{gfoldT (id} \cdot \text{joinT) App Lam (mapT distT} \cdot \text{distT)} \\
= & \quad \{\text{identity}\} \\
& \text{gfoldT (joinT} \cdot \text{id) App Lam (mapT distT} \cdot \text{distT)} \\
= & \quad \{\text{fusion (backwards), since joinT} \cdot \text{Lam} = \text{Lam} \cdot \text{joinT} \cdot \text{mapT distT}\} \\
& \text{joinT} \cdot \text{gfoldT id App Lam distT} \\
= & \quad \{\text{definition of joinT}\} \\
& \text{joinT} \cdot \text{joinT}
\end{aligned}$$

4 Abstraction and application

It is time now to return to the main problem in hand, namely, to give the implementations of abstraction and application.

Abstracting with respect to a free variable *x* is easy: each occurrence of *x* in a term is replaced by *Zero*, and each occurrence of a variable *y* \neq *x* is replaced by *Succ y*. This is implemented by:

```

abstract :: Eq a => a -> Term a -> Term a
abstract x = Lam . mapT (match x)

```

² These equations are part of the statement that *distT* is a distributive law (Barr and Wells, 1984) between the monads on *Term* and *Incr*.

```

match :: Eq a => a -> a -> Incr a
match x y = if x == y then Zero else Succ y

```

The definition of application is also quite short. We define application as a function that takes a term t and the body b of a lambda abstraction, and replaces every occurrence of *Zero* (the nameless variable bound by the abstraction) in b by t :

```

apply :: Term a -> Term (Incr a) -> Term a
apply t = joinT . mapT (subst t . mapI Var)

```

The function $mapT (subst t \cdot mapI Var)$ returns an element of *Term* (*Term a*), a term of terms. The function $joinT$ ‘flattens’ such elements into ordinary terms.

The actual substitution is done by the function $subst t$, a left inverse of $match t$:

```

subst :: a -> Incr a -> a
subst x Zero = x
subst x (Succ y) = y

```

Note that the type of $subst$ implies the following ‘free theorem’ (Wadler, 1989):

$$f \cdot subst\ x = subst\ (f\ x) \cdot mapI\ f \quad (23)$$

To check this definition of $apply$, let us prove that substituting an abstracted variable returns the original term:

$$\begin{aligned}
& apply\ (Var\ x) \cdot mapT\ (match\ x) \\
= & \quad \{\text{definitions}\} \\
& joinT \cdot mapT\ (subst\ (Var\ x) \cdot mapI\ Var) \cdot mapT\ (match\ x) \\
= & \quad \{\text{law (23)}\} \\
& joinT \cdot mapT\ (Var \cdot subst\ x) \cdot mapT\ (match\ x) \\
= & \quad \{\text{functor}\} \\
& joinT \cdot mapT\ (Var \cdot subst\ x \cdot match\ x) \\
= & \quad \{\text{functor, monad law (21)}\} \\
& mapT\ (subst\ x \cdot match\ x) \\
= & \quad \{\text{definitions of } subst, match\} \\
& mapT\ id \\
= & \quad \{\text{functor}\} \\
& id
\end{aligned}$$

5 An extension of de Bruijn’s notation

Substitution on de Bruijn terms transforms arguments as well as function bodies, thus precluding sharing. Consider the example term from section 1, with the variables rewritten in unary notation:

$$\lambda.0\ (\lambda.S0\ 0\ (\lambda.SS0\ S0\ 0))$$

If this term is applied to the term $\lambda.0\ S0$, the result is

$$(\lambda.0\ S0)\ (\lambda.(\lambda.0\ SS0)\ 0\ (\lambda.(\lambda.0\ SSS0)\ S0\ 0))$$

where the three versions of the argument are underlined. There is a generalisation of de Bruijn notation in which S can be applied to any term, not just a variable (Paterson, 1991). Its effect is to escape the scope of the matching λ . With this looser representation of terms, one can avoid transforming arguments while substituting. In the above example, substitution yields

$$(\underline{\lambda.0} \ S0) (\underline{\lambda.S}(\underline{\lambda.0} \ S0) \ 0) (\underline{\lambda.SS}(\underline{\lambda.0} \ S0) \ S0 \ 0)$$

In effect, we have postponed pushing the S 's down to the variables.

We still require that each S or 0 have a matching lambda. This constraint is captured by the following definition:

```
data TermE a = VarE a
              | AppE (Pair (TermE a))
              | LamE (TermE (Incr (TermE a)))
```

Note that *TermE* is doubly nested. A similar definition can be used to model quasiquotation (literal data with an escape operator) as in Scheme (Clinger and Rees, 1991) or multi-stage programming languages like MetaML (Taha and Sheard, 1997).

Though *TermE* is more complex, we can follow the same steps as for *BinTree* and *Term*. The mapping function for *TermE* is given by:

```
mapE :: (a -> b) -> TermE a -> TermE b
mapE f (VarE x) = (VarE . f) x
mapE f (AppE p) = (AppE . mapP (mapE f)) p
mapE f (LamE t) = (LamE . mapE (mapI (mapE f))) t
```

The generalised fold operator is

```
gfoldE :: (forall a. m a -> n a) ->
         (forall a. Pair (n a) -> n a) ->
         (forall a. n (Incr (n a)) -> n a) ->
         (forall a. Incr a -> m (Incr a)) ->
         TermE (m b) -> n b
gfoldE v a l k (VarE x) = v x
gfoldE v a l k (AppE p) = (a . mapP (gfoldE v a l k)) p
gfoldE v a l k (LamE t) = (l . gfoldE v a l k .
                           mapE (k . mapI (gfoldE v a l k))) t
```

Note the change in type for the last argument k : a lambda abstraction for extended terms with variables of type mb has type

$$TermE (Incr (TermE (mb)))$$

Applying *mapE* (*mapI* (*gfoldE v a l k*)) to a value of this type produces an element of type

$$TermE (Incr (nb))$$

Applying *mapE k* to this element produces an element of type

$$TermE (m (Incr (nb)))$$

A second recursive application of $gfoldE\ v\ a\ l\ k$ now produces an element of the type required by l , namely,

$$n(\text{Incr } (nb))$$

The identity law for extended terms is

$$gfoldE'\ VarE\ AppE\ LamE\ id = id \quad (24)$$

The map-fusion law is

$$gfoldE\ v\ a\ l\ (h \cdot k) \cdot mapE\ h = gfoldE\ (v \cdot h)\ a\ l\ k \quad (25)$$

The fusion law for $gfoldE\ v\ a\ l\ k :: TermE\ (M\ a) \rightarrow N\ a$ is

$$h \cdot gfoldE\ v\ a\ l\ k = gfoldE\ v'\ a'\ l'\ (mapM\ k' \cdot k) \quad (26)$$

$$\Leftarrow \begin{cases} h \cdot v = v' \\ h \cdot a = a' \cdot mapP\ h \\ h \cdot l = l' \cdot h \cdot mapN\ (k' \cdot mapI\ h) \end{cases}$$

Extended terms also comprise a monad, with unit $VarE$ and join operator defined by:

```
joinE :: TermE (TermE a) -> TermE a
joinE = gfoldE id AppE LamE VarE
```

Verification of the monad laws is straightforward. For example, we will prove that

$$joinE \cdot mapE\ joinE = joinE \cdot joinE \quad (27)$$

We have

$$\begin{aligned} & joinE \cdot mapE\ joinE \\ = & \{ \text{definition of } joinE \} \\ & gfoldE\ id\ AppE\ LamE\ VarE \cdot mapE\ joinE \\ = & \{ \text{definition of } joinE \} \\ & gfoldE\ id\ AppE\ LamE\ (joinE \cdot VarE \cdot VarE) \cdot mapE\ joinE \\ = & \{ \text{map fusion} \} \\ & gfoldE\ joinE\ AppE\ LamE\ (VarE \cdot VarE) \\ = & \{ \text{naturality of } VarE \} \\ & gfoldE\ joinE\ AppE\ LamE\ (mapE\ VarE \cdot VarE) \\ = & \{ \text{fusion (backwards)} \} \\ & joinE \cdot gfoldE\ id\ AppE\ LamE\ VarE \\ = & \{ \text{definition of } joinE \} \\ & joinE \cdot joinE \end{aligned}$$

With the definitions above, we can define abstraction and application:

```
abstractE :: Eq a => a -> TermE a -> TermE a
abstractE x = LamE . mapE (mapI VarE . match x)
```

```
applyE :: TermE a -> TermE (Incr (TermE a)) -> TermE a
applyE t = joinE . mapE (subst t)
```

Finally, let us see how to convert extended terms into ordinary ones. We want a function

$$\text{cvtE} :: \text{TermE } a \rightarrow \text{Term } a$$

We will define cvtE as an instance of gfoldE . Typing considerations dictate that $m = \text{Id}$ and $n = \text{Term}$ in the type assignment for gfoldE . Once again Haskell forces us to define a variant gfoldE' , whose definition is the same as that of gfoldE , but with m specialised to Id . We define

$$\text{cvtE} = \text{gfoldE}' \text{ Var App (Lam . joinT . mapT distT) id}$$

To check this definition, we can show that cvtE is a *monad morphism*, that is, it satisfies the equations:

$$\text{cvtE} \cdot \text{VarE} = \text{Var} \tag{28}$$

$$\text{cvtE} \cdot \text{joinE} = \text{joinT} \cdot \text{mapT} \text{cvtE} \cdot \text{cvtE} \tag{29}$$

The first is immediate from the definition, and the second is an appeal to fusion:

$$\begin{aligned} & \text{cvtE} \cdot \text{joinE} \\ = & \quad \{\text{definition of joinE}\} \\ & \text{cvtE} \cdot \text{gfoldE id AppE LamE VarE} \\ = & \quad \{\text{fusion}\} \\ & \text{gfoldE cvtE App (Lam . joinT . mapT distT) (mapE id . VarE)} \\ = & \quad \{\text{identity}\} \\ & \text{gfoldE cvtE App (Lam . joinT . mapT distT) VarE} \\ = & \quad \{\text{map fusion (backwards)}\} \\ & \text{gfoldE id App (Lam . joinT . mapT distT) (cvtE \cdot \text{VarE}) \cdot \text{mapE cvtE}} \\ = & \quad \{\text{definition of cvtE}\} \\ & \text{gfoldE id App (Lam . joinT . mapT distT) Var \cdot \text{mapE cvtE}} \\ = & \quad \{\text{identity}\} \\ & \text{gfoldE id App (Lam . joinT . mapT distT) (Var \cdot \text{id}) \cdot \text{mapE cvtE}} \\ = & \quad \{\text{fusion (backwards)}\} \\ & \text{joinT \cdot gfoldE' Var App (Lam . joinT . mapT distT) id \cdot \text{mapE cvtE}} \\ = & \quad \{\text{definition of cvtE}\} \\ & \text{joinT \cdot cvtE \cdot \text{mapE cvtE}} \\ = & \quad \{\text{naturality of cvtE}\} \\ & \text{joinT \cdot \text{mapT} cvtE \cdot \text{cvtE}} \end{aligned}$$

This equation is used in the proof that substitution on extended terms correctly mirrors substitution on de Bruijn terms:

$$\text{cvtE} \cdot \text{applyE } t = \text{apply} (\text{cvtE } t) \cdot \text{cvtBodyE} \tag{30}$$

where cvtBodyE converts an extended abstraction body to a simple one:

$$\begin{aligned} \text{cvtBodyE} & :: \text{TermE (Incr (TermE } a)) \rightarrow \text{Term (Incr } a) \\ \text{cvtBodyE} & = \text{joinT} \cdot \text{mapT} \text{distT} \cdot \text{cvtE} \cdot \text{mapE} (\text{mapI cvtE}) \end{aligned}$$

The proof is lengthy but routine, and we omit it.

6 Conclusion

Our representation of de Bruijn terms illustrates the ability of nested datatypes to express constraints on data structures, so that they can be enforced by the type checker. It has also served as a test case for the extension to nested datatypes of structuring principles developed for regular datatypes, using maps, folds and monads. In the case of de Bruijn terms, these operators do most of the work, including handling bound variables, so that the definition of application and abstraction is particularly simple. Moreover, with programs structured in this way most proofs are mechanical, and were indeed generated using the simple automatic calculator described in Bird (1998).

Programs that manipulate nested types require a number of recently explored extensions of the Hindley-Milner type system. The limited form of type constructor polymorphism provided by Haskell has been an occasional hindrance, forcing us to define specialised versions of polymorphic functions, or new datatypes that are equivalent to existing types; in both cases an opportunity for reuse is lost. It might be reasonable to design a language in which these restrictions were lifted, at the cost of explicit abstraction and instantiation with respect to type constructors, but not types.

Acknowledgements

Oege de Moor suggested using a nested datatype for lambda terms. An anonymous referee suggested a number of improvements.

References

- Barr, M. and Wells, C. (1984) *Toposes, triples and theories*. Grundlehren der Mathematischen Wissenschaften, no. 278. Springer-Verlag.
- Bird, R. and de Moor, O. (1997) *The Algebra of Programming*. Prentice Hall.
- Bird, R. (1998) *Introduction to Functional Programming using Haskell*. Prentice Hall.
- Bird, R. and Paterson, R. (1998) *Generalised folds for nested datatypes*. Submitted.
- Bird, R. S. and Meertens, L. (1998) Nested datatypes. *Mathematics of Program Construction: Lecture Notes in Computer Science 1422*, pp. 52–67. Springer-Verlag.
- Clinger, W. and Rees, J. (1991) Revised⁴ report on the algorithmic language Scheme. *ACM Lisp pointers*, **IV**(July–September).
- de Bruijn, N. G. (1972) Lambda calculus notation with nameless dummies. *Indagationes Mathematicae*, **34**, 381–392.
- Jones, M. (1998) *A technical summary of the new features in Hugs 1.3c*. Unpublished.
- McCracken, N. J. (1984) The typechecking of programs with implicit type structure. *Semantics of Data Types: Lecture Notes in Computer Science 173*, pp. 301–315. Springer-Verlag.
- Paterson, R. A. (1991) Non-deterministic lambda-calculus: A core for integrated languages. *Declarative programming, Sassbachwalden*. Springer-Verlag.
- Paulson, L. C. (1996) *ML for the Working Programmer (2nd edn)*. Cambridge University Press.
- Peyton Jones, S. et al. (1998) *The Glasgow Haskell Compiler*. Department of Computer Science, University of Glasgow.
- Taha, W. and Sheard, T. (1997) Multi-stage programming with explicit annotations. *ACM Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pp. 203–217.
- Wadler, P. (1989) Theorems for free! *4th Conference on Functional Programming Languages and Computer Architecture*, pp. 347–359. IFIP.