JFP **30**, e10, 10 pages, 2020.
doi:10.1017/S0956796820000064

FUNCTIONAL PEARLS

Heterogeneous binary random-access lists

WOUTER SWIERSTRA
Utrecht University
(*e-mail:* w.s.swierstra@uu.nl)

1 Introduction

Writing an evaluator for the simply typed lambda calculus is a classic example of a dependently typed program that appears in numerous tutorials (McBride, 2004; Norell, 2009, 2013; Abel, 2016). The central idea is to represent the *well-typed* lambda terms over some universe U using an inductive family (Figure 1). Before writing the evaluator for such terms, we need to define a type of environments, capturing the values associated with the free variables in a term. This is typically done using a heterogeneous list, indexed by a list of the free variables' types:

```
data Env : Ctx → Set where
  Nil  : Env Nil
  Cons : Val u → Env ctx → Env (u :: ctx)

lookup : Env ctx → Ref ctx u → Val u
lookup (Cons x ctx) Top     = x
lookup (Cons x ctx) (Pop ref) = lookup ctx ref
```

When writing the evaluator, the type indices ensure that we can reuse the host language's constructs for lambda and application, rather than having to define substitution and β-reduction ourselves. It is only in the case for variables that we have to do any real work: looking up the value of a variable from the environment:

```
eval : Term ctx u → Env ctx → Val u
eval (App t1 t2)  env = (eval t1 env) (eval t2 env)
eval (Lam body) env = λ x → eval body (Cons x env)
eval (Var i)      env = lookup env i
```

This evaluator, however, is not particularly efficient. In particular, the environment is represented as a heterogeneous list of values with linear time lookup.

This technique of using indexed data types to represent well-typed expressions has trickled down into numerous industrial applications built using Haskell, including the Crucible symbolic simulator developed by Galois and Accelerate library for generating GPU code (Chakravarty *et al.*, 2011). Christiansen *et al.* (2019) write: '*the experience*

```
data U : Set where
  ι       : U
  _⇒_ : U → U → U
Val : U → Set
Val ι           = ⊤
Val (u1 ⇒ u2) = Val u1 → Val u2
Ctx = List U
data Ref : Ctx → U → Set where
  Top : Ref (s :: ctx) s
  Pop : Ref ctx s → Ref (t :: ctx) s
data Term : Ctx → U → Set where
  App : Term ctx (s ⇒ t) → Term ctx s → Term ctx t
  Lam : Term (s :: ctx) t → Term ctx (s ⇒ t)
  Var : Ref ctx s → Term ctx s
```

Fig. 1: An inductive family of well-typed, well-scoped lambda terms

of profiling Crucible showed that linear access… imposed an unacceptable overhead on the simulator'. As a result, the Crucible developers have chosen a hybrid approach that sacrifices type safety for performance.

This pearl explores how to define a *heterogeneous* data structure that provides a lookup function with *logarithmic* complexity. The key challenge is to choose indices judiciously, thereby avoiding the need for additional lemmas or type coercions to ensure type safety. Doing so will enable us to define a more efficient evaluator, that is total, equally simple and easily verified to produce the same results as the evaluator above.

2 Binary random-access lists

Before trying to define an efficient data structure storing heterogeneous values, we will first consider the simpler homogeneous case. In this section, we will start by writing an Agda implementation of *homogeneous* binary random-access lists (Okasaki, 1999). We will then define a heterogeneous version, as required by our evaluator, indexed by the homogeneous version—much as the heterogeneous environments Env are indexed by a (homogeneous) list of types. The key challenge of implementing such data structures in Agda will be to choose data types that ensure all our definitions are *total*.

To achieve logarithmic lookup times, we need to shift from linear lists to binary trees. If we assume that we only have to store 2^n elements, we could use a perfect binary tree of depth n:

```
data Tree (a : Set) : ℕ → Set where
  Leaf : a → Tree a Zero
  Node : Tree a n → Tree a n → Tree a (Succ n)
```

To define a lookup function, we need to consider how to designate a position in the tree. One way to do so is using a path of n steps, providing directions at every internal node:

```
data Path : ℕ → Set where
  Here  : Path Zero
  Left  : Path n → Path (Succ n)
  Right : Path n → Path (Succ n)

lookup : Tree a n → Path n → a
lookup (Node t₁ t₂) (Left p)  = lookup t₁ p
lookup (Node t₁ t₂) (Right p) = lookup t₂ p
lookup (Leaf x)     Here      = x
```

Note that the index n is shared by both the depth of the tree and the length of the path, ensuring that our lookup function is *total*: we do not need to provide cases for the Node-Here, Leaf-Left or Leaf-Right constructor combinations. Throughout this paper, code in each section is in a separate module, allowing function names such as lookup to be reused liberally. Only when necessary, we will use qualified names.

Although our lookup function is now logarithmic, we can only store a fixed number of elements in this tree. In particular, there is no way to add new elements—as is required by our interpreter. Furthermore, we may want to store a number of elements that is not equal to a power of two. Fortunately, any natural number can be written as a *sum* of powers of two—and we can use this insight to define a better data structure.

2.1 Binary arithmetic

Before doing so, however, we will need two auxiliary definitions: a data type Bin representing little-endian binary numbers and a function bsucc that computes the successor of a binary number:

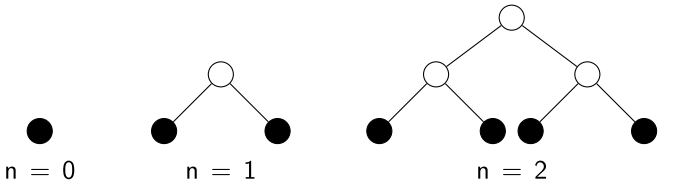
```
data Bin : Set where
  End  : Bin
  One  : Bin → Bin
  Zero : Bin → Bin

bsucc : Bin → Bin
bsucc End      = One End
bsucc (One b)  = Zero (bsucc b)
bsucc (Zero b) = One b
```

Note that this definition provides different representations of the same number—the binary numbers End and Zero End are two different representations of zero—but this will not be a problem in our setting. By construction, our definitions will always avoid unnecessary trailing zeros.

2.2 Binary random-access lists

We now turn our attention to defining a suitable structure for storing an *arbitrary* number of elements. The key insight used by Okasaki's *binary random-access lists* is that the binary representation of the number of elements we wish to store determines how to organise



these elements over a series of perfectly balanced binary trees. For example, we can store seven elements in three perfect trees of increasing depth, as illustrated above.

To store fewer elements, we can leave out any of these trees. For example, we might use the first and last trees to store five elements. The binary representation of the number of elements determines which trees must be present and which trees must be omitted.

We can make this precise in the following data type for *binary random-access lists*:

```
data RAL (a : Set) : (n : ℕ) → Bin → Set where
  Nil    : RAL a n End
  Cons₁ : Tree a n → RAL a (Succ n) b → RAL a n (One b)
  Cons₀ : RAL a (Succ n) b → RAL a n (Zero b)
```

A value of type RAL a n b consists of a series of perfectly balanced binary trees of increasing depth. The Nil constructor corresponds to an empty list of trees; the other constructors extend the current binary number with a One or Zero, respectively. In the prior case, we also have a tree of depth n; in either case, we increment the depth of the trees in the remainder of the binary random-access list.

It is worth highlighting the choice of indices here. These binary random-access lists are indexed by the current depth, n, and the binary representation of the number of elements they store. In contrast to the familiar example of vectors, the index n counts up rather than down. The depth n will be Zero initially, but is incremented in both Cons nodes. The binary number used as an index both completely determines the constructors used and counts the number of elements stored.

How do we designate a position in such a binary random-access list? We combine the linear references from the introduction and the paths from the previous section as follows:

```
data Pos : (n : ℕ) → (b : Bin) → Set where
  Here    : Path n → Pos n (One b)
  There₀ : Pos (Succ n) b → Pos n (Zero b)
  There₁ : Pos (Succ n) b → Pos n (One b)
```

The two constructors, There_0 and There_1, are used to designate a position further down the list of trees. Once we hit the desired tree, the Here constructor stores a path of suitable length, which we can use to navigate to a leaf in the tree at the head of our binary random-access list. Using both these data types, we can define a total lookup function:

```
lookup : RAL a n b → Pos n b → a
lookup (Cons₁ t ral) (Here path) = Tree.lookup t path
lookup (Cons₀ ral)   (There₀ i)  = lookup ral i
lookup (Cons₁ _ ral) (There₁ i)  = lookup ral i
```

Crucially, as the binary random-access list and position share the same depth n and binary number b, we can rule out having to search the empty binary random-access list. This proves that each value of type Pos n b is guaranteed to designate a unique value in the binary random-access list of type RAL a n b.

In contrast to perfectly balanced binary trees, we can add a single element to a binary random-access list. To do so, we begin by defining the more general consTree function that adds a tree of depth n to a binary random-access list:

```
consTree : Tree a n → RAL a n b → RAL a n (bsucc b)
consTree t Nil          = Cons₁ t Nil
consTree t (Cons₁ t' r) = Cons₀ (consTree (Node t t') r)
consTree t (Cons₀ r)    = Cons₁ t r
```

As its type suggests, this function closely follows the successor operation on binary numbers. It searches for the first occurrence of a Cons_0 constructor, accumulating any subtrees found in a Cons_1 constructor along the way. Note that the number n, counting both the depth of the tree being inserted and the position in the list of trees being traversed, increases in the recursive call in the Cons_1 branch.

Finally, we can add a single element to a binary random-access list by calling consTree with an initial tree storing the single element to be inserted:

```
cons : a → RAL a Zero b → RAL a Zero (bsucc b)
cons x r = consTree (Leaf x) r
```

Note that the cons function requires the depth of the binary random-access list to be Zero, as it calls consTree with a leaf of depth Zero as argument; this depth increases as the consTree function recurses over the binary random-access list.

Although we have an extensible data structure that supports logarithmic lookup time, we can only store elements of a single type. Using these binary random-access lists, however, we can define a heterogeneous alternative.

3 Heterogeneous binary random-access lists

In this section, we will show how to use our previous definitions to define a binary random-access list storing values of different types. For every data type definition in the previous sections, we will give a heterogeneous version indexed by a (homogeneous) structure storing type information. For example, we can define a heterogeneous perfect binary tree as follows:

```
data HTree : Tree U n → Set where
  HLeaf  : Val u → HTree (Leaf u)
  HNode : HTree us → HTree vs → HTree (Node us vs)
```

Just as the environments from the introduction were indexed by a *list of types*, we can index these heterogeneous trees by a *tree of types*, determining the types of the values stored in the leaves. The function Val : U → Set, defined in Figure 1, maps the codes from the universe U to the corresponding types. While not strictly necessary, we will prefix

the constructors of heterogeneous types with a capital 'H' to distinguish them from their homogeneous counterparts.

Next, we introduce a heterogeneous version of our Path data type. A value of type HPath t u corresponds to a path in the tree t : Tree U n, ending at a leaf storing u. The constructors closely follow the constructors for the Path data type; it is only the type indices that have changed:

```
data HPath : Tree U n → U → Set where
  HHere  : HPath (Leaf u) u
  HLeft  : HPath us u → HPath (Node us vs) u
  HRight : HPath vs u → HPath (Node us vs) u
```

We can define a lookup function with the following type by induction over the path:

```
lookup : HTree us → HPath us u → Val u
```

The definition is—up to constructor names—identical to the one we have seen previously; the only difference is in the type signature, as the type of the value that is returned may vary depending on the position in the tree.

We can now define a heterogeneous version of our binary random-access lists, indexed by its homogeneous counterpart:

```
data HRAL : RAL U n b → Set where
  HNil    : HRAL Nil
  HCons₁ : HTree t → HRAL ral → HRAL (Cons₁ t ral)
  HCons₀ : HRAL ral → HRAL (Cons₀ ral)
```

Once again, the constructors closely follow the constructors of binary random-access lists, RAL. The only key difference lies in the choice of indices. The original RAL data type was indexed by a binary number that determined its structure; here the structure is determined by a binary random-access list of types.

Similarly, we also revisit the type of positions in a binary random-access list:

```
data HPos : RAL U n b → U → Set where
  HHere   : HPath t u → HPos (Cons₁ t ral) u
  HThere₀ : HPos ral u → HPos (Cons₀ ral) u
  HThere₁ : HPos ral u → HPos (Cons₁ t ral) u
```

The lookup function on heterogeneous random-access lists follows the same structure as its homogeneous counterpart. Types aside, the only real difference is the call to the heterogeneous lookup function on binary trees, rather than the homogeneous version we saw previously:

```
lookup : HRAL ral → HPos ral u → Val u
lookup (HCons₁ t hral) (HHere tp)   = HTree.lookup t tp
lookup (HCons₀ hral)   (HThere₀ p) = lookup hral p
lookup (HCons₁ x hral) (HThere₁ p) = lookup hral p
```

Finally, the definition of cons and consTree are readily adapted to the heterogeneous setting:

```
consTree : HTree t → HRAL ral → HRAL (RAL.consTree t ral)
consTree t HNil              = HCons1 t HNil
consTree t (HCons1 t' hral) = HCons0 (consTree (HNode t t') hral)
consTree t (HCons0 hral)    = HCons1 t hral

cons : (x : Val u) → HRAL ral → HRAL (RAL.cons u ral)
cons x r = consTree (HLeaf x) r
```

Again, the only interesting change here is in the type signature. The result of the cons function uses the cons operation on homogeneous binary random-access lists defined in the previous section. Rather than incrementing the binary number counting the number of elements as we did in the homogeneous case, we extend the index binary random-access list with the type of the new element.

4 An alternative evaluator

Finally, we can write a variation of our original evaluator. We begin by defining two functions. The first calculates the length of a (linear) context represented as a binary number; the second converts a context to a binary random-access list:

```
lengthBin : Ctx → Bin
lengthBin Nil       = End
lengthBin (u :: ctx) = bsucc (lengthBin ctx)

makeRAL : (ctx : Ctx) → RAL.RAL U Zero (lengthBin ctx)
makeRAL Nil       = RAL.Nil
makeRAL (u :: ctx) = RAL.cons u (makeRAL ctx)
```

The random-access list produced in this fashion has type RAL U Zero (lengthBin ctx). As we are constructing a binary random-acesss by repeatedly calling the cons function, the second index must be Zero; (the binary representation of) the number of elements is computed using the lengthBin function.

Next, we would like to convert the linear references from the introduction to the positions in a binary random-access list. This amounts to defining a function:

```
toHPos : Ref ctx u → HPos (makeRAL ctx) u
```

Rather than attempt to define such a function directly, we instead show that the *constructors* of the Ref data type, Top and Pop, have the following counterparts:

```
pop : HPos ral u → HPos (RAL.cons v ral) u

top : HPos (RAL.cons u ral) u
```

Both these functions are defined by induction on the binary random-access list ral. The top function computes the 'first' element of a non-empty binary random-access list; the pop function updates an existing position, after a new element has been added. Just as we saw for the cons function, both pop and top can be defined in terms of a pair of more

general functions that accumulate the desired positions as they traverse the underlying binary random-access lists.

Using these definitions, it is entirely straightforward to convert a position in a linear list to one in the corresponding binary random-access list:

```
toHPos : Ref ctx u → HPos (makeRAL ctx) u
toHPos Top      = top
toHPos (Pop ref) = pop (toHPos ref)
```

We now generalise the lambda terms from the introduction, abstracting over the choice of how to represent variables:

```
data Term (var : Ctx → U → Set) : Ctx → U → Set where
  App : Term var ctx (u ⇒ v) → Term var ctx u → Term var ctx v
  Lam : Term var (u : ctx) v → Term var ctx (u ⇒ v)
  Var : var ctx u → Term var ctx u
```

By choosing to use the linear references, Ref, from the introduction to represent variables, we can redefine the original evaluator:

```
evalRef : Term Ref ctx u → Env ctx → Val u
```

Alternatively, we can write an evaluator that uses our heterogeneous binary random-access lists and the corresponding positions:

```
HP : Ctx → U → Set
HP ctx u = HPos (makeRAL ctx) u

evalHPos : Term HP ctx u → HRAL (makeRAL ctx) → Val u
evalHPos (App t₁ t₂)  env = (evalHPos t₁ env) (evalHPos t₂ env)
evalHPos (Lam body) env = λ x → evalHPos body (HRAL.cons x env)
evalHPos (Var i)      env = HRAL.lookup env i
```

Crucially, the definition does not require type coercions or any additional proofs to type check. The type indices of the Term data type ensure we can still safely use Agda's lambda abstraction and application; the cons operations on homogeneous and heterogeneous random-access lists used in the types and values, respectively, share the same structure.

Can we prove these two evaluators are equal? To relate them, we need to convert linear environments that the previous evaluator used to a heterogeneous binary random-access list:

```
toHRAL : Env ctx → HRAL (makeRAL ctx)
toHRAL Nil          = HNil
toHRAL (Cons x env) = cons x (toHRAL env)
```

Next, we can show that our conversions in representation, toHRAL and toHPos, respect the lookup operation by proving the following equality:

```
lookupLemma : (env : Env ctx) (x : Ref ctx u) →
  Intro.lookup env x ≡ HRAL.lookup (toHRAL env) (toHPos x)
```

The proof relies on a pair of auxiliary lemmas, relating the top and pop functions to the lookup of our heterogeneous binary random-access lists:

```
lookupPop : (x : Val v) (env : HRAL ral) (p : HPos ral u) →
  lookup env p ≡ lookup (cons x env) (pop p)
lookupTop : (x : Val u) (env : HRAL.HRAL ral) →
  x ≡ lookup (cons x env) top
```

Furthermore, we can map one choice of variable representation to another by defining:

```
mapTerm : (∀ {ctx u} → A ctx u → B ctx u) → Term A ctx u → Term B ctx u
```

Finally, we can prove that, assuming functional extensionality, our two evaluators produce identical results:

```
correct : (t : Term Ref ctx u) (env : Env ctx) →
  evalRef t env ≡ evalHPos (mapTerm toHPos t) (toHRAL env)
```

The proof itself, using our lookupLemma, is only three lines long.

5 Discussion

Although the code presented here is written in Agda, it can be converted to Haskell using various modern language features, including data kinds (Yorgey *et al.*, 2012), generalised algebraic data types (Vytiniotis *et al.*, 2006), type families (Eisenberg *et al.*, 2014) and multi-parameter type classes with functional dependencies (Jones, 2000). Very few of the definitions presented here rely on *computations* appearing in types; as a result, the translation is mostly straightforward. The most problematic parts are the top and pop functions from Section 4—the cons function that appears on the type level complicates their definition. Using type classes rather than type families, we can express the desired computations more easily as a relation, rather than a (total) function in Agda as we did in this paper.

This Haskell version opens the door to more realistic performance studies. While beyond the scope of this pearl, we expect heterogeneous binary random-access lists to be quite a bit slower Haskell's highly optimised Data.Map library. Nonetheless, we would hope that the performance improvement over heterogeneous lists is sufficient to address the bottlenecks mentioned in the introduction.

Finally, there is clearly a more general pattern at play here: creating a heterogeneous data structure by indexing it by its homogeneous counterpart. Using ornaments (Dagand, 2013; McBride, 2010 (unpublished)), we might be able to give a generic account of this construction. In particular, the work by Ko & Gibbons (2016) on binomial heaps would be a valuable starting point for such exploration.

Acknowledgments

The Software Technology Reading Club, James McKinna and anonymous reviewers gave insightful and detailed feedback, for which I am most grateful. This paper is dedicated in loving memory of Doaitse Swierstra, who originally suggested this problem.

Conflict of interest

None.

References

Abel, A. (2016) Agda tutorial. In 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4–6, 2016, Proceedings. Springer.

Chakravarty, M. M. T., Keller, G., Lee, S., McDonell, T. L. & Grover, V. (2011) Accelerating haskell array codes with multicore gpus. In Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming. ACM, pp. 3–14.

Christiansen, D. T., Diatchki, I. S., Dockins, R., Hendrix, J. & Ravitch, T. (2019) Dependently typed haskell in industry (experience report). *Proc. ACM Program. Lang.* **3**(ICFP), 100:1–100:16.

Dagand, P.-E. (2013) *A cosmology of datatypes*. Ph.D. thesis, University of Strathclyde.

Eisenberg, R. A., Vytiniotis, D., Peyton Jones, S. & Weirich, S. (2014) Closed type families with overlapping equations. Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14. ACM, pp. 671–683.

Jones, M. P. (2000) Type classes with functional dependencies. European Symposium on Programming. Springer, pp. 230–244.

Ko, H.-S. & Gibbons, J. (2016) Programming with ornaments. *J. Funct. Program.*, **27**(December). https://doi.org/10.1017/S0956796816000307

McBride, C. (2004) Epigram: Practical programming with dependent types. In International School on Advanced Functional Programming. Springer, pp. 130–170.

Norell, U. (2009) Dependently typed programming in Agda. In Advanced Functional Programming: 6th International School AFP. Berlin, Heidelberg: Springer, pp. 230–266.

Norell, U. (2013) Interactive programming with dependent types. In Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP '13. New York, NY, USA: ACM, pp. 1–2.

Okasaki, C. (1999) *Purely Functional Data Structures*. Cambridge University Press.

Vytiniotis, D., Weirich, S. & Peyton Jones, S. (2006) Simple unification-based type inference for gadts. In ACM SIGPLAN International Conference on Functional Programming (ICFP '06). ACM Press, pp. 50–61.

Yorgey, B., Weirich, S., Cretin, J., Peyton Jones, S., Vytiniotis, D. & Magalhaes, J. P. (2012) Giving haskell a promotion. In *Proceedings of TLDI'12*. ACM, pp. 50–61.