Under consideration for publication in J. Functional Programming

FUNCTIONAL PEARL
Heterogeneous 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; Norell, 2013; Abel, 2016). The central idea is to represent the *well-typed* lambda terms over some universe U using an inductive family:

```
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 Γ (s ⇒ t) → Term Γ s → Term Γ t
  Lam : Term (s :: Γ) t → Term Γ (s ⇒ t)
  Var : Ref Γ s → Term Γ s
```

When writing the evaluator, the type indices ensure that we can reuse the host language's lambdas and application, rather than having to define substitution and β-reduction ourselves:

```
data Env : Ctx → Set where
  Nil : Env Nil
  Cons : Val u → Env ctx → Env (u :: ctx)
eval : Term Γ s → Env Γ → Val s
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 pearl explores how to write such an interpreter using a more efficient data structure, namely *random-access lists*. The key challenge is to choose indices judiciously, ensuring the resulting evaluator is equally simple and does not rely on additional lemmas or type coercions.

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, using the homogeneous version—much as the heterogeneous environments Env are indexed by a (homogeneous) list of types.

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 it the tree. One way to do so, is using a vector of length n, providing direction at every internal node:

```
data Dir : Set where
  Left   : Dir
  Right : Dir
lookup : Tree a n → Vec Dir n → a
lookup (Node l r) (Cons Left xs)   = lookup l xs
lookup (Node l r) (Cons Right xs) = lookup r xs
lookup (Leaf x)    Nil                     = x
```

Note that the index n is shared by both the depth of the tree and the length of the vector, ensuring that our lookup function is total: we do not need to provide cases for the Node-Nil or Leaf-Cons 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, will we 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.

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.

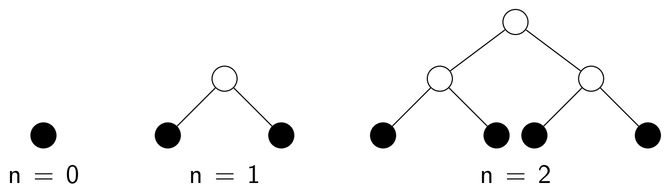
```
data Bin : Set where
  End  : Bin
  One  : Bin → Bin
  Zero : Bin → Bin
```

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

Note that this simple definition provides different representations of the same number—but this will not be a problem in our setting.

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 *random-access lists* is that if we want to store n elements efficiently, the binary representation of n tells us 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:



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 *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 random-access list.

It is worth highlighting the choice of indices here. These random-access lists are indexed by the current depth, n, and the binary representation of the number of elements they store. The depth n will typically be Zero initially, but is incremented in along every Cons node. The binary number used as an index completely determines the constructors used.

How do we designate a position in such a random-access list? We mimic the usual well-typed references used in the introduction:

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

Given a vector of directions, we can navigate to a leaf in the tree at the head of our random-access list, if it exists. Otherwise, there are two constructors, There_0 and There_1, to designate a position further down the list. Using both these definitions, we can define a total lookup function:

```
lookup : (ral : 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 random-access list and position share the same depth n and binary number b, we can rule out having to search the empty random-access list.

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

Unsurprisingly, 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. We can add a single element to a 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
```

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

3 Heterogeneous random-access lists

In this section, we will show how to adapt our previous definitions, allowing them to store heterogeneous elements. For every data type definition in the previous, we will give a heterogeneous version indexed by a (homogeneous) structure storing the type information. For example, we can define a heterogeneous perfect binary tree as follows:

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

Just as the environment from the introduction was indexed by a *list* of types, we can index these heterogeneous trees by a tree of types, that determine the types of the values stored in the leaves. Here we assume that the function Val : U → Set maps the codes from the universe U to the corresponding types.

Rather than use vectors as we did previously, we now introduce a separate data type to describe a path through a heterogeneous tree, navigating to a particular value of type U:

```
data TreePath : Tree U n → U → Set where
  Here  : TreePath (Leaf u) u
  Left  : TreePath us u → TreePath (Node us vs) u
  Right : TreePath vs u → TreePath (Node us vs) u
```

Once again, we can define the desired lookup function by induction over the tree path:

```
lookup : HTree ut → TreePath ut u → Val u
```

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

Similarly, we can revisit random-acces lists and present a heterogeneous version, indexed by its homogeneous counterpart:

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

The type of positions now tracks the type of the designated value:

```
data Pos : RAL U n b → U → Set where
  Here   : TreePath t u → Pos (Cons₁ t ral) u
  There₀ : Pos ral u → Pos (Cons₀ ral) u
  There₁ : Pos ral u → Pos (Cons₁ t ral) u
```

The lookup function traverses the list of perfect trees until it can use use the lookup function on perfect binary trees:

```
lookup : HRAL ral → Pos ral u → Val u
lookup (Cons₁ t hral) (Here tp)   = HTree.lookup t tp
lookup (Cons₀ hral)   (There₀ p) = lookup hral p
lookup (Cons₁ x hral) (There₁ 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 Nil             = Cons₁ t Nil
consTree t (Cons₁ t' hral) = Cons₀ (consTree (Node t t') hral)
consTree t (Cons₀ hral)    = Cons₁ t hral

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

The only interesting change here is in the type signature. The result of cons function uses the cons operation on homogeneous random-access lists defined in the previous section.

4 An alternative evaluator

Finally, we can write a variation of our original evaluator. We begin by defining functions that calculate the binary number associated with a (linear) context, and convert a context to a random-access list:

```
sizeBin : Ctx → Bin
sizeBin Nil        = End
sizeBin (x :: ctx) = bsucc (sizeBin ctx)

makeRAL : (ctx : Ctx) → RAL.RAL U Zero (sizeBin ctx)
makeRAL Nil        = RAL.Nil
makeRAL (x :: ctx) = RAL.cons x (makeRAL ctx)
```

Next we will define two functions, pop and top, to refer to the first element of a random-access list and tail of a random-access list respectively:

```
pop : Pos ral s → Pos (RAL.cons t ral) s
top : Pos (RAL.cons x ral) x
```

The definitions of these functions require several auxiliary definitions to manipulate the binary trees involved. Using these definitions, however, it is entirely straightforward to convert a position in a linear list to one in the corresponding random-access list:

```
toPos : Ref ctx s → Pos (makeRAL ctx) s
toPos Top       = top
toPos (Pop ref) = pop (toPos ref)
```

We now generalize 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 Γ (s ⇒ t) → Term var Γ s → Term var Γ t
  Lam : Term var (s :: Γ) t → Term var Γ (s ⇒ t)
  Var : var Γ s → Term var Γ s
```

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

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

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

```
P : Ctx → U → Set
P ctx u = Pos (makeRAL ctx) u

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

Crucially, the definition does not require type coercions or any additional proofs to type check. Can we prove these two evaluators are equal? To relate them, we need to relate the random-access lists and linear environments the previous evaluator used:

$$
\begin{aligned}
&\mathsf{toEnv} : \mathsf{Env}\,\Gamma \to \mathsf{HRAL}\,(\mathsf{makeRAL}\,\Gamma)\\
&\mathsf{toEnv}\,\mathsf{Nil} = \mathsf{Nil}\\
&\mathsf{toEnv}\,(\mathsf{Cons}\,\mathsf{x}\,\mathsf{env}) = \mathsf{cons}\,\mathsf{x}\,(\mathsf{toEnv}\,\mathsf{env})
\end{aligned}
$$

We can show that the toEnv and toPos relate the lookup in our linear environments and random-access lists.

$$
\begin{aligned}
&\mathsf{lookupLemma} : (\mathsf{env} : \mathsf{Env}\,\Gamma) \to (\mathsf{x} : \mathsf{Ref}\,\Gamma\,\mathsf{s}) \to\\
&\quad \mathsf{Intro.lookup}\,\mathsf{env}\,\mathsf{x} \equiv \mathsf{HRAL.lookup}\,(\mathsf{toEnv}\,\mathsf{env})\,(\mathsf{toPos}\,\mathsf{x})
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathsf{lookupPop} : (\mathsf{p} : \mathsf{Pos}\,\mathsf{ctx}\,\mathsf{s}) \to \mathsf{lookup}\,\mathsf{env}\,\mathsf{p} \equiv \mathsf{lookup}\,(\mathsf{cons}\,\mathsf{y}\,\mathsf{env})\,(\mathsf{pop}\,\mathsf{p})\\
&\mathsf{lookupTop} : \mathsf{x} \equiv \mathsf{lookup}\,(\mathsf{cons}\,\mathsf{x}\,\mathsf{env})\,\mathsf{top}
\end{aligned}
$$

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

$$
\mathsf{mapTerm} : (\textbf{forall}\,\{\mathsf{u}\}\,\{\Gamma\} \to \mathsf{A}\,\mathsf{u}\,\Gamma \to \mathsf{B}\,\mathsf{u}\,\Gamma) \to \mathsf{Term}\,\mathsf{A}\,\Gamma\,\mathsf{s} \to \mathsf{Term}\,\mathsf{B}\,\Gamma\,\mathsf{s}
$$

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

$$
\begin{aligned}
&\mathsf{correct} : (\mathsf{t} : \mathsf{Term}\,\mathsf{Ref}\,\Gamma\,\mathsf{s})\,(\mathsf{env} : \mathsf{Env}\,\Gamma) \to\\
&\quad \mathsf{evalRef}\,\mathsf{t}\,\mathsf{env} \equiv \mathsf{evalPos}\,(\mathsf{mapTerm}\,\mathsf{toPos}\,\mathsf{t})\,(\mathsf{toEnv}\,\mathsf{env})
\end{aligned}
$$

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

Acknowledgements I would like to thank Doaitse Swierstra for suggesting this problem. The Software Technology Reading Club provided valuable advice on an earlier draft of this paper.

References

Abel, Andreal. (2016). Agda tutorial. *13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*. Springer.

McBride, Conor. (2004). Epigram: Practical programming with dependent types. *Pages 130–170 of: International School on Advanced Functional Programming*. Springer.

Norell, Ulf. (2009). *Dependently typed programming in agda*. Berlin, Heidelberg: Springer Berlin Heidelberg. Pages 230–266.

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

Okasaki, Chris. (1999). *Purely functional data structures*. Cambridge University Press.