Heterogenous binary random-access lists

Functional pearl

Wouter Swierstra

Utrecht University

Lists

- Lists are one of the very first data types that we teach undergraduates learning functional programming.

Lists

- Lists are one of the very first data types that we teach undergraduates learning functional programming.

- Students that go on to *industry* use more efficient structures to store large amounts of data, such as finite maps or balanced binary trees.

Lists

- Lists are one of the very first data types that we teach undergraduates learning functional programming.
- Students that go on to *industry* use more efficient structures to store large amounts of data, such as finite maps or balanced binary trees.
- Students that stay in *academia* to do a PhD use heterogeneous lists (aka `HLists`) to write evaluators for lambda calculi.

Heterogeneous and efficient?

Question: Can we define a data structure that is both **heterogeneous** and **efficient**?

Heterogeneous and efficient?

Question: Can we define a data structure that is both **heterogeneous** and **efficient**?

This is not a theoretical problem

Christiansen et al. wrote in their paper on *Dependently Typed Haskell in Industry* at ICFP last year:

> *the experience of profiling Crucible showed that linear access... imposed an unacceptable overhead on the simulator*

Results

This pearl demonstrates how to implement *heterogeneous binary random-access lists* in Agda.

- the same API as heterogeneous lists:
  - an empty structure (`Nil`);
  - an operation to add a new element to the front (`Cons`);
  - an operation to access elements (`lookup` or `!!`)

  All these operations are total and type-safe.
- no coercions or additional lemmas needed to type check.

This talk

I won't try to cover the whole paper in this talk – but instead present *homogeneous* binary random-access lists, originally due to Okasaki.

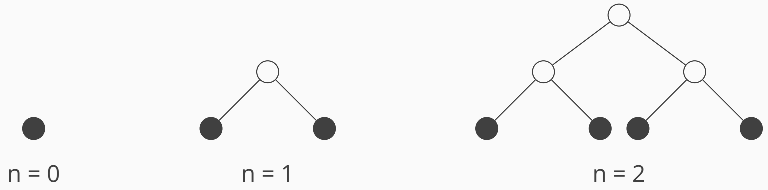
The heterogeneous version follows naturally from this, by indexing a data structure with a binary random-access list storing the types of all the values it contains.

From lists to trees

To achieve super linear access, we need to shift from lists to trees.

In a perfect world, we only ever have to store 2^n elements...

This is easy to do in a perfectly balanced binary tree of depth n

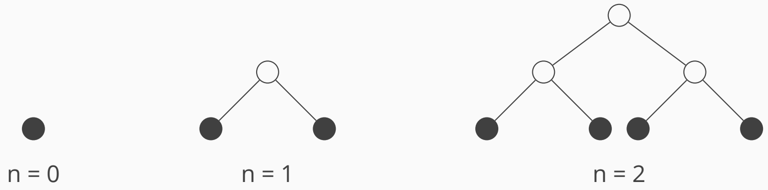


From lists to trees

To achieve super linear access, we need to shift from lists to trees.

In a perfect world, we only ever have to store 2^n elements...

This is easy to do in a perfectly balanced binary tree of depth *n*




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

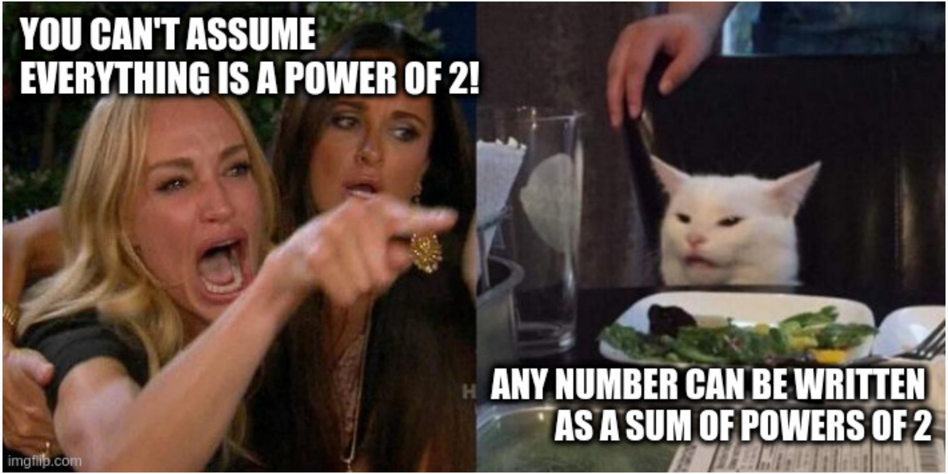
Accessing elements in a tree

To denote a particular value stored in a tree of depth `n`, we need to `n` steps telling us to continue in the left subtree or the right subtree.

```
data Path : Nat → 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: the indices ensure we that this function is total.

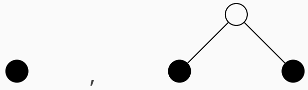

YOU CAN'T ASSUME EVERYTHING IS A POWER OF 2!
ANY NUMBER CAN BE WRITTEN AS A SUM OF POWERS OF 2
imgflip.com

Binary random-access lists

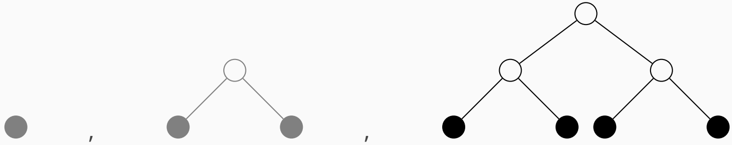
A *binary random-access list* consists of a list of perfect binary trees of increasing depth.

At the i-th position in this list, there may or may not be a perfect binary tree of depth i.

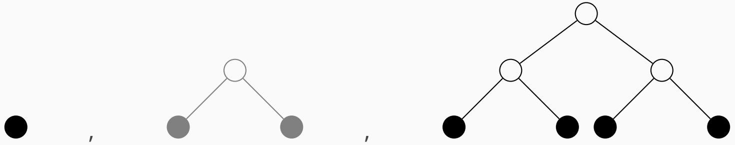
Binary random-access lists storing three elements



Binary random-access lists storing four elements



Binary random-access lists storing five elements



Binary numbers

Every number can be written as a sum of powers of two.

A number’s representation in binary determines the shape of the binary random-access list storing that many elements.

Binary numbers

Every number can be written as a sum of powers of two.

A number's representation in binary determines the shape of the binary random-access list storing that many elements.

```
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
```

Random access lists

```
data RAL (a : Set) (n : Nat) : 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)
```

- the binary number counts the number of elements and uniquely determines the shape of our random-access list
- the number `n` increases as we go down the list – the next tree is going to have more elements (unlike vectors, for example)
- we usually start counting from n = Zero, but it's useful to be a bit more general.

Positions and lookup

We can now define a type `Pos n b` that denotes an element stored in a `RAL a n b`:

```
data Pos (n : Nat) : 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)
```

Each position traverses the outer list of trees, ending with a path of depth `n`.

```
lookup : RAL a n b → Pos n b → a
```

Adding elements

Finally, we might want to add new elements to the binary random-access list.

A first attempt might be to define a function such as:

```
cons : a → RAL a Zero b → RAL a Zero (bsucc b)
```

Adding elements

Finally, we might want to add new elements to the binary random-access list.

A first attempt might be to define a function such as:

```
cons : a → RAL a Zero b → RAL a Zero (bsucc b)
```

But we quickly get stuck – we cannot make any recursive calls as the ‘tail’ of the binary random-access list stores larger trees.

Adding elements

Finally, we might want to add new elements to the binary random-access list.

A first attempt might be to define a function such as:

```
cons : a → RAL a Zero b → RAL a Zero (bsucc b)
```

But we quickly get stuck – we cannot make any recursive calls as the 'tail' of the binary random-access list stores larger trees.

Instead, we need to define a more general operation 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)
```

Conclusions

- We can extend this to the heterogeneous case:

```
data HRAL : RAL U n b → Set where ...
```

- Despite the apparent complexity, writing an 'efficient' lambda calculus evaluator written using heterogeneous binary random-access lists is no harder than using heterogeneous lists.
- 'Easy' to port to Haskell in 130 lines of code...

Conclusions

- We can extend this to the heterogeneous case:

```
data HRAL : RAL U n b → Set where ...
```

- Despite the apparent complexity, writing an ‘efficient’ lambda calculus evaluator written using heterogeneous binary random-access lists is no harder than using heterogeneous lists.
- ‘Easy’ to port to Haskell in 130 lines of code...
- ...of which 10% is language extensions pragmas

Conclusions

- We can extend this to the heterogeneous case:

```
data HRAL : RAL U n b → Set where ...
```

- Despite the apparent complexity, writing an ‘efficient’ lambda calculus evaluator written using heterogeneous binary random-access lists is no harder than using heterogeneous lists.
- ‘Easy’ to port to Haskell in 130 lines of code…
- …of which 10% is language extensions pragmas

Choose the right datastructure

- and ensure that your type indices capture the key invariants.

Recap

Question: Can we define a data structure that is both **heterogeneous** and **efficient**?

Results: This pearl demonstrates how to implement *heterogeneous binary random-access lists* in Agda.

- the same API as heterogeneous lists;
- All these operations are total and type-safe; no coercions or additional lemmas needed to type check.

Key insight: any number can be expressed as a sum of powers of two; any number of elements can be stored in a series of perfect trees of increasing depth.