Utrecht University

Calculating datastructures

Ralf Hinze and *Wouter Swierstra*

TU Kaiserslautern and Utrecht University

Calculating datastructures

There are tons of (purely functional) datastructures:

- binary random access lists;
- 2-3 trees;
- finger trees;
- binomial heaps;
- Braun trees;
- ...

Calculating datastructures

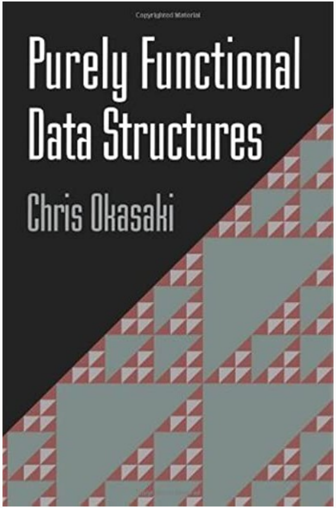
There are tons of (purely functional) datastructures:

- binary random access lists;
- 2-3 trees;
- finger trees;
- binomial heaps;
- Braun trees;
- ...

Who comes up with these?

Purely functional datastructures

...data structures that can be cast as numerical representations are surprisingly common, but only rarely is the connection to a number system noted explicitly.



Calculating datastructures

- We will fix a particular API, keeping the numerical representation we use abstract for the moment.
- We can then show how different choices of numerical representation lead to different *implementations* of this API.
- Using the properties our API must satisfy, we can apply familiar *type isomorphisms* to *calculate* the *datastructure* that implements the API.

All these calculations can be performed and verified in Agda.

Flexible arrays – the interface

```
Number   : Set
Index    : Number → Set
Array    : Number → Set → Set

lookup   : Array n elem → (Index n → elem)
tabulate : (Index n → elem) → Array n elem

nil      : Array 0 elem
cons     : elem → Array n elem → Array (1 + n) elem
head     : Array (1 + n) elem → elem
tail     : Array (1 + n) elem → Array n elem
```

Take 0 : Peano numbers

```
data Peano : Set where
  zero  : Peano
  succ  : Peano → Peano

data Index : Peano → Set where
  izero : Peano (succ n)
  isucc : Peano n → Peano (succ n)
```

Towards calculation...

```
lookup   : Array n elem → (Index n → elem)
tabulate : (Index n → elem) → Array n elem
```

These two functions should form an isomorphism.

If we perform induction on `n`, we can calculate a definition of `Array`.

Index isomorphisms

$$
\begin{aligned}
Index(0) &\cong \bot \\
Index(1) &\cong \top \\
Index(m+n) &\cong Index(m) \uplus Index(n) \\
Index(m \cdot n) &\cong Index(m) \times Index(n) \\
Index(n^m) &\cong Index(m) \to Index(n)
\end{aligned}
$$

Note – these isomorphisms are not unique! There are many different choices:

- interleaving vs appending
- column major vs row major
- ...

While these choices are all correct, they lead to *different* datastructures.

Calculating with generic tries

We'll try to find an isomorphism given by the lookup and tabulate functions to 'discover' an implementation of a datastructure.

If we 'calculate' this iso using familiar laws – we can hopefully use this to read off the datastructures that arise.

In particular, we'll use the laws of exponents:

$$
\begin{aligned}
X^0 &\cong 1 \\
X^1 &\cong X \\
X^{A+B} &\cong X^A \cdot X^B \\
X^{A \cdot B} &\cong \left(X^B\right)^A
\end{aligned}
$$

These should be familiar from high school – but can also be read as type isomorphisms.

Example: vectors – base case

```
proof
  (Index zero → elem)
  ≅ -- Index-0 law
  (⊥ → elem)
  ≅ -- law of exponents
  ⊤
  ≅ -- use as definition
  Array zero elem
∎
```

Example: vectors – inductive step

```
proof
  (Index (succ n) → elem)
  ≅ -- definition of Index
  ((⊤ ⊎ Index n) → elem)
  ≅ -- law of exponents
  (⊤ → elem) × (Index n → elem)
  ≅ -- law of exponents
  elem × Array n elem
  ≅ -- use as definition
  Array (succ n) elem
```

In this way, we have connected Peano naturals to vectors – but that's hardly interesting…

Binary numbers

```
data Leibniz : Set where
  0b : Leibniz
  _1 : Leibniz → Leibniz
  _2 : Leibniz → Leibniz

convert : Leibniz → Peano
convert 0b = 0
convert (n 1) = convert n · 2 + 1
convert (n 2) = convert n · 2 + 2
```

This representation of binary numbers is *unique*.

From vectors to trees

I'll go through one of the two cases in some detail:

```
(Index (n 2) → elem)
≅ -- arithmetic on indices
(⊤ ⊎ ⊤ ⊎ Index n ⊎ Index n → elem)
≅ -- laws of exponents
elem × elem × (Index n → elem) × (Index n → elem)
≅ -- recurse
elem × elem × Array n elem × Array n elem
≅ -- use as definition
Array (n 2) elem
```

1-2 trees

In this style, we can (re)discover the type of 1-2 trees:

```
data Array : Leibniz → Set → Set where
  Leaf  : Array 0b
  Node₁ : elem → Array n elem → Array n elem → Array (n 1) elem
  Node₂ : elem × elem → Array n elem → Array n elem → Array (n 2) elem
```

The construction of the isos give us the definition of lookup and tabulate for free.

1-2 trees

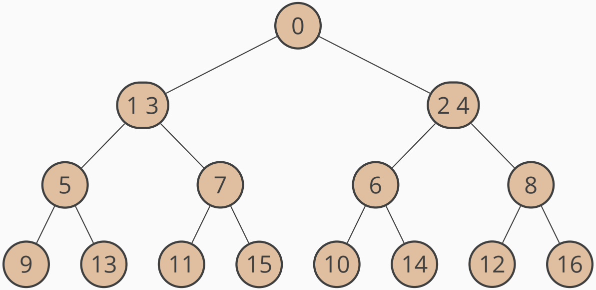
In this style, we can (re)discover the type of 1-2 trees:

```
data Array : Leibniz → Set → Set where
  Leaf  : Array 0b
  Node₁ : elem → Array n elem → Array n elem → Array (n 1) elem
  Node₂ : elem × elem → Array n elem → Array n elem → Array (n 2) elem
```

The construction of the isos give us the definition of lookup and tabulate for free.

What about the other operations?

Example: a 1-2 tree with 17 elements



- Each node has 1 or 2 elements: just enough to ensure the remainding number of elements is even.
- Note that ‘odd elements’ are stored in one subtree and ‘even elements’ in the other.

Adding new elements

To add a new element to the 'front' of the tree, we distinguish three cases:

```
cons : elem → Array n elem → Array (succ n) elem
cons x₀ (Leaf)                = Node₁ x₀ Leaf Leaf
cons x₀ (Node₁ x₁     l r)   = Node₂ x₀ x₁ l r
cons x₀ (Node₂ x₁ x₂  l r)   = Node₁ x₀ (cons x₁ l) (cons x₂ r)
```

- A `Node₁` becomes a `Node₂`, with the new element at the front.
- A `Node₂` becomes a `Node₁` – but we need to add the two elements to the respective subtrees.

Alternatives

Once we have this infrastructure, it is easy to explore variations..

```
(Index (n 2) → elem)
≅ -- arithmetic on indices
(⊤ ⊎ Index (succ n) ⊎ Index n → elem)
≅ -- laws of exponents
elem × (Index (succ n) → elem) × (Index n → elem)
≅ -- use as definition
Array (n 2) elem
```

Instead of having 1-2 nodes – we can have nodes with a single element.

Braun trees

```
data Array : Leibniz → Set → Set where
  Leaf   : Array 0b elem
  Node₁  : elem → Array n         elem → Array n elem → Array (n 1) elem
  Node₂  : elem → Array (succ n)  elem → Array n elem → Array (n 2) elem
```

Each node stores a single element; the two subtrees may store a different number of elements, but differ by at most one.

Extending Braun trees

```
cons : elem → Array n elem → Array (succ n) elem
cons x₀ (Leaf)           = Node₁ x₀ Leaf Leaf
cons x₀ (Node₁ x₁ l r)  = Node₂ x₀ (cons x₁ r) l
cons x₀ (Node₂ x₁ l r)  = Node₁ x₀ (cons x₁ r) l
```

The two subtrees swap! Every even element becomes odd and visa versa.

Random access lists

```
(Index (n 2) → elem)
≅ -- arithmetic on indices
(⊤ ⊎ ⊤ ⊎ Index (2 · n) → elem)
≅ -- laws of exponents
elem × elem × (Index n → elem × elem)
≅ -- use as definition
Array (n 2) elem
```

Instead of having *two* subtrees, we can also have one 'tail' with twice as many elements.

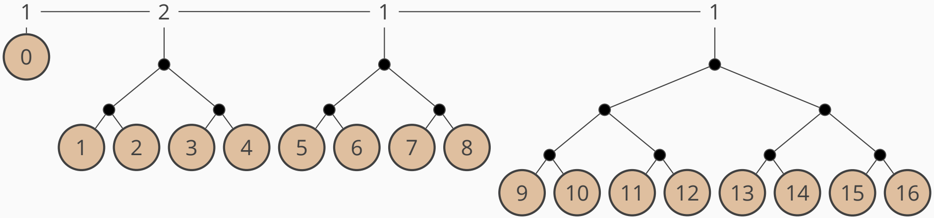
Random access lists

```
data Array : Leibniz → Set → Set where
  nil  : Array 0b elem
  one  : elem          → Array n (elem × elem) → Array (n 1) elem
  two  : elem → elem   → Array n (elem × elem) → Array (n 2) elem
```

A linear structure with a subtree of pairs rather than pair of subtrees.

As a result, we no longer use the interleaving of even-odd elements, but rather elements are stored in ‘usual’ order.

Example: random access list of 17 elements



What else?

We go through a lot more details in the paper:

- explicit proofs of isomorphisms;
- computing index types for various structures;
- many more operations: cons, snoc, tail, lookup, etc.
- lots of pretty pictures

What next?

- Ko has already shown how to describe binary heaps as ornaments on skey binary numbers.
- Isomorphisms are quite a strong criteria – do weaker conditions suffice?
- Isomorphisms are quite a strong criteria – can we get more out of them by going cubical?