Seamless
Correct
Generic } Programming over Serialised Data

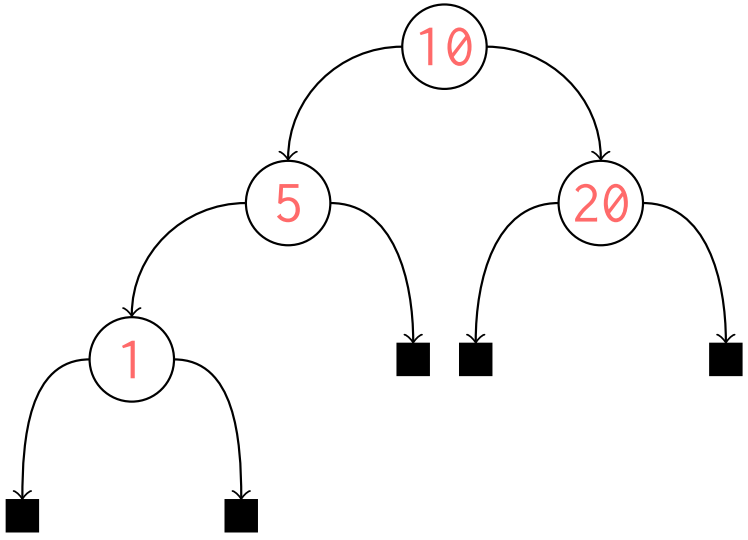
Guillaume Allais

University of St Andrews

SPLS, March 8th 2023

Table of Contents

Trees and Pattern Matching




```
data Tree
  = Leaf
  | Node Tree Bits8 Tree

sum : Tree -> Nat
sum t = case t of
  Leaf => 0
  Node l b r =>
    let m = sum l
        n = sum r
    in (m + cast b + n)
```

Serialised Data and Pointer Manipulations

```
01 01 01 00 01 00 05 00 0a 01 00 14 00
```

Serialised Data and Pointer Manipulations

```
01 01 01 00 01 00 05 00 0a 01 00 14 00
```

```
1  int sumAt (int buf[], int *ptr, int *res) {
2    int tag = buf[*ptr]; (*ptr)++;
3    switch (tag) {
4      case 0: return 0;
5      case 1:
6        sumAt(buf, ptr, res);
7        int val = buf[*ptr]; (*ptr)++;
8        *res += val;
9        sumAt(buf, ptr, res);
10       return 0;
11     default: exit(-1); }}
```

Seamless

```
sum : Data.Mu Tree -> Nat
sum t = case t of
  "Leaf" # _ => Z
  "Node" # l # b # r =>
    let m = sum l
        n = sum r
    in (m + cast b + n)
```

```
sum : Pointer.Mu Tree _ -> IO Nat
sum ptr = case !(view ptr) of
  "Leaf" # _ => pure Z
  "Node" # l # b # r =>
    do m <- sum l
       n <- sum r
       pure (m + cast b + n)
```

Correct

```
data Singleton : (x : a) -> Type where
  MkSingleton : (x : a) -> Singleton x
```

```
sum : Pointer.Mu Tree t ->
      IO (Singleton (Data.sum t))
sum ptr = case !(view ptr) of
  "Leaf" # _ => pure [| Z |]
  "Node" # l # b # r =>
    do m <- sum l
       n <- sum r
       pure [| [| m + [| cast b |] |] + n |]
```

Generic

```
record Data (nm : Type) where
  constructor MkData
  {consNumber : Nat}
  constructors : Vect consNumber (Constructor nm)
```

```
view : {cs : Data nm} ->
       forall t. Pointer.Mu cs t ->
       IO (View cs t)
```

Table of Contents

What's Next?

Already here:

- A monad to *build* serialised value
- More realistic universe (more base types)

To Do:

- Expressivity
  - Polymorphic data types
  - Indexed families
- Performance
  - Benchmarking
  - Partial evaluation / Macro-based code generation
  - More tightly packed representations
- Robustness
  - Proper error handling

Table of Contents

Descriptions

```
data Desc : (rightmost : Bool) ->
            (size : Nat) -> (offsets : Nat) ->
            Type
```

Descriptions

```
data Desc : (rightmost : Bool) ->
            (size : Nat) -> (offsets : Nat) ->
            Type
```

```
data Desc where
  None : Desc r 0 0
  Byte : Desc r 1 0
  Prod : {sl, sr, m, n : Nat} ->
         Desc False sl m -> Desc r sr n ->
         Desc r (sl + sr) (m + n)
  Rec : Desc r 0 (ifThenElse r 0 1)
```

Meaning as Strictly Positive Functors

```
record Tuple (a, b : Type) where
  constructor (#)
  fst : a
  snd : b
```

```
Meaning : Desc r s n -> Type -> Type
Meaning None x = ()
Meaning Byte x = Bits8
Meaning (Prod d e) x = Tuple (Meaning d x) (Meaning e x)
Meaning Rec x = x
```

Constructor descriptions

```
record Constructor (nm : Type) where
  constructor (::)
  name : nm
  {size : Nat}
  {offsets : Nat}
  description : Desc True size offsets


Leaf : Constructor String
Leaf = "Leaf" :: None

Node : Constructor String
Node = "Node" :: Prod Rec (Prod Byte Rec)
```

Data descriptions

```
record Data (nm : Type) where
  constructor MkData
  {consNumber : Nat}
  constructors : Vect consNumber (Constructor nm)
```

```
Tree : Data String
Tree = MkData [Leaf, Node]
```

Meaning as Trees

```
record Index (cs : Data nm) where
  constructor MkIndex
  getIndex : Fin (consNumber cs)

Alg : Data nm -> Type -> Type
Alg cs x = (k : Index cs) -> Meaning (description k) x -> x

data Mu : Data nm -> Type where
  (#) : Alg cs (assert_total (Mu cs))
```

Example

```
leaf : Mu Tree
leaf = "Leaf" # ()

node : Mu Tree -> Bits8 -> Mu Tree -> Mu Tree
node l b r = "Node" # l # b # r

example : Mu Tree
example = node (node (node leaf 1 leaf) 5 leaf)
               10
               (node leaf 20 leaf)
```

Table of Contents

Serialisation Format

```
example : Mu Tree
example = node (node (node leaf 1 leaf) 5 leaf)
               10
               (node leaf 20 leaf)
```

```
87654321  00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF
00000000: 07 00 00 00 00 00 00 00 02 00 02 03 02 01 03 01
00000010: 17 00 00 00 00 00 00 00 01 0c 00 00 00 00 00 00
00000020: 00 01 01 00 00 00 00 00 00 00 00 01 00 05 00 0a
00000030: 01 01 00 00 00 00 00 00 00 00 14 00
```

Meaning as Pointers

```
record Meaning (d : Desc r s n) (cs : Data nm)
               (t : Data.Meaning d (Data.Mu cs)) where
  constructor MkMeaning
  subterms : Vect n Int
  elemBuffer : Buffer
  elemPosition : Int


record Mu (cs : Data nm) (t : Data.Mu cs) where
  constructor MkMu
  muBuffer : Buffer
  muPosition : Int
```

Inspecting Buffers: Head Constructor

```
data Out : (cs : Data nm) -> (t : Data.Mu cs) -> Type where
  (#) : (k : Index cs) ->
        forall t. Pointer.Meaning (description k) cs t ->
        Out cs (k # t)


out : {cs : Data nm} -> forall t. Pointer.Mu cs t ->
      IO (Out cs t)
```

Inspecting Buffers: Extracting Head Data

```
Poke : (d : Desc r s n) -> (cs : Data nm) ->
       Data.Meaning d (Data.Mu cs) -> Type
Poke None _ t = ()
Poke Byte cs t = Singleton t
Poke d@(Prod _ _) cs t = Poke' d cs t
Poke Rec cs t = Pointer.Mu cs t

data Poke' : (d : Desc r s n) -> (cs : Data nm) ->
             Data.Meaning d (Data.Mu cs) -> Type where
  (#) : Pointer.Meaning d cs t ->
        Pointer.Meaning e cs u ->
        Poke' (Prod d e) cs (t # u)
```

Inspecting Buffers: Extracting Head Data (ct'd)

```
poke : {0 cs : Data nm} -> {d : Desc r s n} ->
       forall t. Pointer.Meaning d cs t ->
       IO (Poke d cs t)
```

A More Convenient View

```
data View : (cs : Data nm) -> (t : Data.Mu cs) -> Type where
  (#) : (k : Index cs) ->
        forall t. Layer (description k) cs t ->
        View cs (k # t)


view : {cs : Data nm} ->
       forall t. Pointer.Mu cs t ->
       IO (View cs t)
```