Programming Over Serialised Data

Guillaume Allais

SPLV St Andrews

July 24^{th}–28^{th} 2023

Revisiting Our Universe

Descriptions

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

Descriptions

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

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

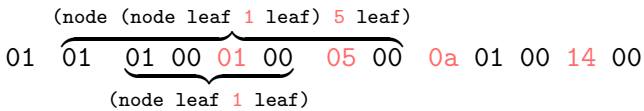
Tweaked Constructor and Data

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

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

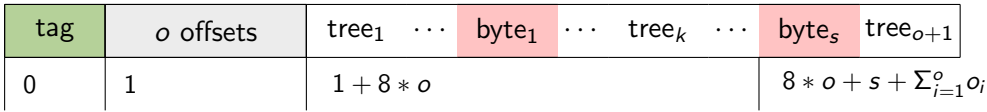
Choosing a Serialisation Format

Staring at a Hexdump (sorry)

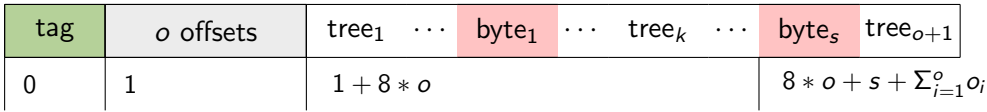


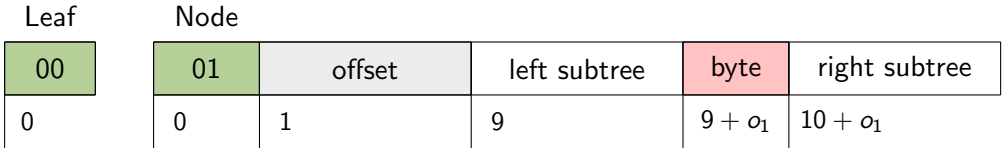

```
87654321   00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF
00000000:  07 00 00 00 00 00 00 00 02 00 02 03 02 01 03 01
00000010:  17 00 00 00 00 00 00 00 01 0c 00 00 00 00 00 00
00000020:  00 01 01 00 00 00 00 00 00 00 00 01 00 05 00 0a
00000030:  01 01 00 00 00 00 00 00 00 00 14 00
```

Data Layout


tag
o offsets
tree1 ··· byte1 ··· treek ··· bytes treeo+1
0
1
1 + 8 ∗ o
8 ∗ o + s + Σ_{i=1}^{o} oi

Data Layout


tag
o offsets
tree1 ··· byte1 ··· treek ··· bytes treeo+1
0
1
1 + 8 ∗ o
8 ∗ o + s + Σ_{i=1}^{o} oi


Leaf
Node
00
01
offset
left subtree
byte
right subtree
0
0
1
9
9 + o1
10 + o1

Meaning as Pointers

Mu Pointers

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

Meaning Pointers

```
record Meaning (d : Desc r s o) (cs : Data nm)
               (t : Data.Meaning d (Data.Mu cs)) where
  constructor MkMeaning
  subterms : Vect o Int
  meaningBuffer : Buffer
  meaningPosition : Int
  meaningSize : Int
```

Inspecting a Buffer's Content: Axioms

Poke

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

```
data Poke' : (d : Desc r s o) -> (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)
```

Poke (continued)

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

Out

```
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 a Buffer's Content: Derived Notions

Layer

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

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

Layer (continued)

```
layer : {0 cs : Data nm} -> {d : Desc r s o} ->
        forall t. Pointer.Meaning d cs t -> IO (Layer d cs t)
layer el = poke el >>= go d where

  go : forall r, s, o. (d : Desc r s o) ->
       forall t. Poke d cs t -> IO (Layer d cs t)
  go None p = pure ()
  go Byte p = pure p
  go (Prod d e) (p # q) = [| layer p # layer q |]
  go Rec p = pure p
```

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)
view ptr = do k # el <- out ptr
              vs <- layer el
              pure (k # vs)
```

Generic Programming

Deserialisation

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

Fold

```
fmap : (d : Desc r s o) ->
       (0 f : Data.Mu cs -> b) ->
       (forall t. Pointer.Mu cs t -> IO (Singleton (f t))) ->
       forall t. Pointer.Meaning d cs t ->
       IO (Singleton (Data.fmap d f t))
```

Fold (continued)

```
fold : {cs : Data nm} -> Alg cs a -> Mu cs -> a
fold alg t
  = let (k # v) = t in
    let rec = assert_total (fmap _ (fold alg) v) in
    alg k rec
```

```
fold : {cs : Data nm} -> (alg : Alg cs a) ->
       forall t. Pointer.Mu cs t ->
       IO (Singleton (Data.fold alg t))
fold alg ptr
  = do (k # t) <- out ptr
       rec <- assert_total (fmap _ _ (fold alg) t)
       pure (alg k <$> rec)
```

Demo!

Conclusion

Other Results

- Serialisation DSL
- Port to Agda

What's Next?

Ongoing:

- More realistic universe (more base types)
- Benchmarking

To Do:

- Expressivity
  - Polymorphic data types
  - Indexed families
  - Descriptions with internal fixpoints
- Performance
  - Partial evaluation / Macro-based code generation
  - More tightly packed representations
- Robustness
  - Proper error handling in serialisation code