A SAFE IMPLEMENTATIONS OF FOLD

We include below the alternative definitions of `fold` (respectively processing inductive data and data stored in a buffer) which are seen as total by Idris 2. Each of them is mutually defined with what is essentially the supercompilation of (`\ d => fmap d (fold alg)`).

```
parameters {cs : Data nm} (alg : Alg cs a)

  fold : Data.Mu cs -> a
  fmapFold : (d : Desc{}) ->
             Data.Meaning d (Data.Mu cs) -> Data.Meaning d a

  fold (k # t) = alg k (fmapFold (description k) t)

  fmapFold None t = t
  fmapFold Byte t = t
  fmapFold (Prod d e) (s # t)
    = (fmapFold d s # fmapFold e t)
  fmapFold Rec t = fold t

parameters {cs : Data nm} (alg : Alg cs a)

  fold : Pointer.Mu cs t -> IO (Singleton (fold alg t))
  fmapFold : (d : Desc{}) -> forall t. Pointer.Meaning d cs t ->
             IO (Singleton (fmapFold alg d t))

  fold ptr
    = do k # t <- out ptr
         rec <- fmapFold (description k) t
         pure (alg k <$> rec)

  fmapFold d ptr = poke ptr >>= go d where

    go : (d : Desc{}) -> forall t. Poke d cs t ->
         IO (Singleton (fmapFold alg d t))
    go None {t} v = rewrite etaUnit t in pure (pure ())
    go Byte v = pure v
    go (Prod d e) (v # w)
      = do v <- fmapFold d v
           w <- fmapFold e w
           pure [| v # w |]
    go Rec v = fold v
```

B ACCESS PATTERNS: VIEWING VS. POKING

In this example we implement `rightmost`, the function walking down the rightmost branch of our type of binary trees and returning the content of its rightmost node (if it exists).

The first implementation is the most straightforward: use `view` to obtain the top constructor as well as an entire layer of deserialised values and pointers to substructures and inspect the constructor. If we have a leaf then there is no byte to return. If we have a node then call `rightmost` recursively and inspect the result: if we got `Nothing` back we are at the rightmost node and can return the current byte, otherwise simply propagate the result.

```
rightmost : Pointer.Mu Tree t -> IO (Maybe Bits8)
rightmost ptr = case !(view ptr) of
  "Leaf" # _ => pure Nothing
  "Node" # _ # b # r => do
    mval <- rightmost r
    case mval of
      Just _ => pure mval
      Nothing => pure (Just (getSingleton b))
```

In the alternative implementation we use `out` to expose the top constructor and then, in the node case, call `poke` multiple times to get our hands on the pointer to the right subtree. We inspect the result of recursively calling `rightmost` on this subtree and only deserialise the byte contained in the current node if the result we get back is `Nothing`.

```
rightmost : Pointer.Mu Tree t -> IO (Maybe Bits8)
rightmost ptr = case !(out ptr) of
  "Leaf" # _ => pure Nothing
  "Node" # el => do
    (_ # br) <- poke el
    (b # r) <- poke br
    mval <- rightmost !(poke r)
    case mval of
      Just _ => pure mval
      Nothing => do
        b <- poke b
        pure (Just (getSingleton b))
```

This will give rise to two different access patterns: the first function will have deserialised all of the bytes stored in the nodes along the tree's rightmost path whereas the second will only have deserialised the rightmost byte. Admittedly deserialising a byte is not extremely expensive but in a more realistic example we could have for instance been storing arbitrarily large values in these nodes. In that case it may be worth trading convenience for making sure we are not doing any unnecessary work.

C ANALOGY TO SEPARATION LOGIC

Some readers may feel uneasy about the fact that parts of our library are implemented using Idris 2 escape hatches. This section justifies this practice by drawing an analogy to separation logic and highlighting that this practice corresponds to giving an axiomatisation of the runtime behaviour of our library's core functions.

C.1 Interlude: Separation Logic

A Hoare triple [Hoare 1969] of the form

$$\{\, P \,\}\ e\ \{\, v.\, Q \,\}$$

states that under the precondition P, and binding the result of evaluating the expression e as v, we can prove that Q holds. One of the basic predicates of separation logic [Reynolds 2002] is a 'points to' assertion $(\ell \mapsto t)$ stating that the label ℓ points to a memory location containing t.

A separation logic proof system then typically consists in defining a language and providing axioms characterising the behaviour of each language construct. The simplest example involving memory is perhaps a language with pointers to bytes and a single `deref` construct dereferencing a

pointer. We can then give the following axiom

$$\{\ \ell \mapsto bs\ \}\ \texttt{deref}\ \ell\ \{\ v.\ bs = v * \ell \mapsto bs\ \}$$

to characterise `deref` by stating that the value it returns is precisely the one the pointer is referencing, and that the pointer is still valid and still referencing the same value after it has been dereferenced.

The axioms can be combined to prove statements about more complex programs such as the following silly one for instance. Here we state that if we dereference the pointer a first time, discard the result and then dereference it once more then we end up in the same situation as if we had dereferenced it only once.

$$\{\ \ell \mapsto bs\ \}\ \texttt{deref}\ \ell;\ \texttt{deref}\ \ell\ \{\ v.\ bs = v * \ell \mapsto bs\ \}$$

Note that in all of these rules bs is only present in the specification layer. `deref` itself cannot possibly return bs directly, it needs to actually perform an effectful operation that will read the memory cell's content.

C.2 Characterising Our Library

We are going to explain that we can see our library as a small embedded Domain Specific Language (eDSL) [Hudak 1996] that has `poke` and `out` as sole language constructs. Our main departure from separation logic is that we want to program in a correct-by-construction fashion and so the types of `poke` and `out` have to be just as informative as the axioms we would postulate in separation logic. This dual status of the basic building blocks being both executable programs *and* an axiomatic specification of their respective behaviour is precisely why their implementations in Idris 2 necessarily uses unsafe features.

We are going to write $\ell \overset{[\![\, d \,]\!]\ (\mu\, cs)}{\longmapsto} t$ for the assumption that we own a pointer ℓ of type (`Pointer.Meaning d cs t`), and $\ell \overset{\mu\, cs}{\longmapsto} t$ for the assumption that we own a pointer ℓ of type (`Pointer.Mu cs t`). In case we do not care about the type of the pointer at hand (e.g. because it can be easily inferred from the context), we will simply write $\ell \mapsto t$.

C.2.1 Axioms for `poke`. Thinking in terms of Hoare triples, if we have a pointer ℓ to a term t known to be a single byte then (`poke` ℓ) will return a byte bs and allow us to observe that t is equal to that byte.

$$\{\ \ell \overset{[\![\, \texttt{Byte} \,]\!]\ (\mu\, cs)}{\longmapsto} t\ \}\ \texttt{poke}\ \ell\ \{\ bs.\ t = bs * \ell \mapsto t\ \}$$

Similarly, if the pointer ℓ is for a pair then (`poke` ℓ) will reveal that the term t can be taken apart into the pairing of two terms t_1 and t_2 and return a pointer for each of these components.

$$\{\ \ell \overset{[\![\, \texttt{Prod}\, d_1 d_2 \,]\!]\ (\mu\, cs)}{\longmapsto} t\ \}$$

$$\texttt{poke}\ \ell$$

$$\{\ (\ell_1,\ \ell_2).\ \exists t_1.\ \exists t_2.\ t = (t_1\ \#\ t_2) * \ell \mapsto t * \ell_1 \overset{[\![\, d_1 \,]\!]\ (\mu\, cs)}{\longmapsto} t_1 * \ell_2 \overset{[\![\, d_2 \,]\!]\ (\mu\, cs)}{\longmapsto} t_2\ \}$$

Last but not least, poking a pointer with the `Rec` description will return another pointer for the same value but at a different type.

$$\{\ \ell \overset{[\![\, \texttt{Rec} \,]\!]\ (\mu\, cs)}{\longmapsto} t\ \}\ \texttt{poke}\ \ell\ \{\ \ell_1.\ \ell_1 \overset{\mu\, cs}{\longmapsto} t * \ell \mapsto t\ \}$$

C.2.2 Example of a Derived Rule for `layer`. Given that `layer` is defined in terms of `poke`, we do not need to postulate any axioms to characterise it and can instead prove lemmas. We will skip the proofs here but give an example of a derived rule. Using the description (`Prod Rec` (`Prod Byte Rec`)) of the arguments to a node in our running example of binary trees, `layer`'s behaviour would be characterised by the following statement.

$$\{\ \ell \xmapsto{[\![\texttt{Prod Rec}\ (\texttt{Prod Byte Rec})]\!]\ (\mu\, cs)} t\ \}$$

$$\texttt{layer}\ \ell$$

$$\{\ (\ell_1,\, bs,\, \ell_2).\ \exists t_1.\ \exists t_2.\ t = (t_1\ \#\ bs\ \#\ t_2) * \ell_1 \xmapsto{\mu\, cs} t_1 * \ell_2 \xmapsto{\mu\, cs} t_2 * \ell \mapsto t\ \}$$

It states that provided a pointer to such a meaning, calling `layer` would return a triple of a pointer ℓ_1 for the left subtree, the byte bs stored in the node, and a pointer ℓ_2 for the right subtree.

C.2.3 Axiom for `out`. The only other construct for our small DSL is the function `out`. Things are a lot simpler here as the return type is not defined by induction on the description. As a consequence we only need the following axiom.

$$\{\ \ell \xmapsto{\mu\, cs} t\ \}\ \texttt{out}\ \ell\ \{\ (k,\, \ell_1).\ \exists t_1.\ t = (k\ \#\ t_1) * \ell \mapsto t * \ell_1 \xmapsto{[\![\, cs_k\,]\!]\ (\mu\, cs)} t_1\ \}$$

It states that under the condition that ℓ points to t, (`out` ℓ) returns a pair of an index and a pointer to the meaning of the description associated to that index by cs, and allows us to learn that t is constructed using that index and that meaning.

By combining `out` and `layer` we could once more define a derived rule and prove e.g. that every tree can be taken apart as either a leaf or a node with a pointer to a left subtree, a byte, and a pointer to a right subtree i.e. what `view` does in our library.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009