Manual Memory Management in Haskell

Vanessa McHale

As I have remarked in the past, simply having linear types in a language is actually sufficient to prevent many memory errors. As far as I know, ATS is the only language that implements anything like this, but I will be focusing on a proposed Haskell extension in this example.

If you want to follow along with the example, you'll have to download the appropriate branch of the compiler from here.

```
{-# LANGUAGE GADTs         #-}

import           Foreign.Marshal.Alloc
import           Foreign.Ptr
import           Foreign.Storable

foreign import ccall unsafe "stdlib.h free" lfree :: Ptr a ->. IO ()
```

First we define `ViewType` which will help enforce safe memory management. Note the special arrow `->.` used for linear arguments.

```
data ViewType a = ViewType (Ptr a ->. IO ()) (Ptr a)
```

We will define a `Storable` instance and an example data type as follows:

```
data Point = Point Float Float
    deriving (Show)

floatWidth :: Int
floatWidth = sizeOf (undefined :: Float)

instance Storable Point where
    sizeOf _ = 2 * floatWidth
    alignment _ = floatWidth
    peek ptr = do
        x <- peekByteOff ptr 0
        y <- peekByteOff ptr floatWidth
        pure (Point x y)
    poke ptr (Point x y) =
        pokeByteOff ptr 0 x >>
        pokeByteOff ptr floatWidth y
```

The following manually modifies the second `Float` contained in a `Point`. It takes a `ViewType` and thus is safe.

```
pokeY :: Float -> ViewType Point -> IO ()
pokeY x (ViewType _ ptr) = pokeByteOff ptr floatWidth x

safePeek :: Storable a => ViewType a -> IO a
safePeek (ViewType _ ptr) = peek ptr
```

We then need a safe way to convert a value to a `ViewType`:

```
safeAlloc :: Storable a => a -> IO (ViewType a)
safeAlloc x = do
    ptr <- malloc
    poke ptr x
    pure $ ViewType lfree ptr
```

Finally, we need a way to safely free values, viz.

```
dealloc :: Storable a => ViewType a ->. IO ()
dealloc (ViewType f ptr) = f ptr
```

We write a small program to test all of this:

```
main :: IO ()
main = do
    let x = Point 1.0 3.0
    vptr <- safeAlloc x
    pokeY 2.0 vptr
    x' <- safePeek vptr
    dealloc vptr
    print x'
```

Thus, while Haskell does not give us the full safety (or ease) of Rust, we can choose to expose a limited API for manual memory management, within which all manipulations are memory safe.