Stateful $\subset$ stateless

Controlling C libraries via Haskell with `inline-c`. An introduction and a case study

Marco Zocca[1]

[1] **github.com/ocramz**

April 15, 2016
Zimpler, Göteborg

Contents

Who's ocramz?

- Born in Venice (IT)
- M.Sc. from DTU (DK)
- Ph.D. (ongoing) in stochastic optimization at TU Delft (NL)
- Not a CS by trade. Previous life using Matlab, stumbled upon Scheme, stayed for the λs, fell in love with Haskell

Problem statement

Scientific/numerical computation

- Legacy libraries contain domain-specific knowledge, are validated, perform well
- Performance code, book-keeping $\neq$ mathematical form

Example: solution of linear systems. Ubiquitous in practice, $O(n^3)$ if done naively

Strategy: wrap the legacy parts in a soft layer of types

A soft layer of types - binding C within Haskell

An example problem: how to represent this

```
int open( char *filename )
```

as the following

```
open :: String -> IO (Either FileException FileDescriptor)
```

- Marshalling input/output data (e.g. elementwise copy)
- Data lifetime, scope
- "Side effects", "state": allocation, in-place data modification, network/DB access ...
- Managing exceptions
- Handling optionally non-existing or invalid data

Generation of FFI bindings

Tools, in ascending order of flexibility:

- `hsc2hs` (comes with GHC)
- `bindings-dsl` [2] (builds on `hsc2hs` using C macros)
- `c2hs`
- Greencard (uncommon)
- `inline-c` [3]

Tasks:

- Representing enum's, constants
- Writing types for C pointers
- Computing sizes and offsets of pointed-at data (`Storable` instances)
- Rendering `foreign import` function signatures (caveats apply [1])

NB: often we do not need to represent e.g. a C struct $1 : 1$ (e.g. have access to all its fields); an opaque pointer e.g. `newtype A = A (Ptr A) deriving Storable` is sufficient.

Tool chain choices

- Small C library, simple compilation/install : bundle it inside Haskell package and manage compilation completely with `cabal`
- Bundled library, complex compilation/install : same as above, but declare build process within `Setup.hs` file using "hooks"
- Library not bundled with package: manual compilation and linking.

NB: Fill the `C-sources`, `include-dirs`, `includes` entries within the `Library` stanza of your project's .cabal file

Manual linking case: Don't forget `extra-include-dirs` and `extra-lib-dirs` (also available as command line options to `cabal`)

Personal considerations

Large bindings still require quite some attention from a human developer even if using these synthesis tools.
Dev must memorize yet another macro language that is not amenable to formal analysis / syntax checking ; hsc2hs and c2hs are not supported (yet?) by `flycheck`.

Usage example of c2hs:

```
{#fun unsafe Initialized as ^ {alloca- ‘Bool’ peekBool*} -> ‘()’ discard*-
#}
```

is expanded into

```
initialized :: IO Bool
initialized =
  alloca $ \a ->
    initialized'_ a >>= \res ->
      discard res >> peekBool a


foreign import ccall unsafe "Control/Parallel/MPI/Internal.chs.h
MPI_Initialized"
  initialized'_ :: Ptr CInt -> IO CInt
```

Inline-c

Recently (2015) revamped by FPComplete.

- `TemplateHaskell` + `QuasiQuotes`
- Arbitrary blocks of C from `#defined` constants to nested loops (hence the name)
- Supported by flycheck
- QuasiQuotes are expanded directly, with variable capture (much like `(macroexpand-1 ... )`)
- Ready-made marshalling to-from ByteString, Vector, function pointers

Inline-c (2)

Quick prototyping across languages:

Mutating state without returning anything useful to the Haskell runtime:

```
matSetDiagonal :: Mat -> PetscScalar_ -> CInt -> CInt -> IO ()
matSetDiagonal mat val n imm =
  [C.block | void{
      int i;
      for (i=0; i< $(int n); i++){
        MatSetValues( $(Mat mat),
         1, &i,
         1, &i,
         &$(PetscScalar val),
         $(int imm));
                                 };       }|]
```

Inline-c (3)

Allocating space, running function, reshaping output :

```
withPtr :: (Storable a) => (Ptr a -> IO b) -> IO (a, b)
withPtr f = do
  alloca $ \ptr -> do
    x <- f ptr
    y <- peek ptr
    return (y, x)

matGetOwnershipRange' :: Mat -> IO ((Int, Int), CInt)
matGetOwnershipRange' m = do
  (r2, (r1, e)) <- mgor m
  return ((fi r2, fi r1), e) where
    mgor a =
      withPtr $ \rmin ->
       withPtr $ \rmax ->
       [C.exp|int{MatGetOwnershipRange($(Mat a),
          $(PetscInt *rmin),
          $(PetscInt *rmax) )}|]
```

Inline-c (4)

Custom `Context` for storing type map

```
type PetscInt_ = CInt

petscCtx :: Context
petscCtx = baseCtx <> funCtx <> vecCtx <> bsCtx <> pctx where
  pctx = mempty {ctxTypesTable = petscTypesTable}

petscTypesTable :: Map.Map CT.TypeSpecifier TH.TypeQ
petscTypesTable = Map.fromList
                  [
                    (typeNameId "PetscInt", [t| PetscInt_ |] )
                    ]
```

The bracket pattern

From `Control.Exception`, runs cleanup (e.g. deallocation) action even if asynchronous exceptions occur :)

```
bracket
        :: IO a         -- ^ computation to run first ("acquire")
        -> (a -> IO b)  -- ^ computation to run last ("release")
        -> (a -> IO c)  -- ^ computation to run in-between
        -> IO c
bracket before after thing =
  mask $ \restore -> do
    a <- before
    r <- restore (thing a) `onException` after a
    _ <- after a
    return r
```


E.Z.Yang, The Haskell preprocessor hierarchy


Hackage, bindings-dsl


Hackage, inline-c