High-level FFI in Haskell

Keegan McAllister

Boston Area Haskell Users' Group

April 1, 2011

Calling C from Haskell

Calling C from Haskell is easy:

```
foreign import ccall sqrtf :: Float -> Float
main = print (sqrtf 2.0)
```

Making C libraries feel like Haskell libraries is hard!

- Different philosophy regarding types
- Resource management
- Concurrency

Case study: hdis86

`udis86` is a fast, complete, flexible disassembler for x86

- easy to embed

`hdis86` provides an idiomatic Haskell interface

- and embeds `udis86` by default

Quick example

```
GHCi> import Hdis86
GHCi> import qualified Data.ByteString as BS
GHCi> let code = BS.pack [0xcc, 0xf0, 0xff,
                           0x44, 0x9e, 0x0f]
GHCi> disassemble intel64 code
[Inst [] Iint3 [],
 Inst [Lock] Iinc [Mem (Memory {
   mSize   = Bits32,
   mBase   = Reg64 RSI,
   mIndex  = Reg64 RBX,
   mScale  = 4,
   mOffset = Immediate {
     iSize  = Bits8, iValue = 15}})]]
```

Hdis86.C

Bottom layer: C types and conventions

Just a simple import with hsc2hs

```
data UD_t  -- empty type

init :: Ptr UD_t -> IO ()

set_input_buffer
  :: Ptr UD_t -> Ptr CChar -> CSize -> IO ()

disassemble  :: Ptr UD_t -> IO CUInt
get_lval_u32 :: Ptr Operand -> IO Word32
```

Hdis86.IO

Next layer: Haskell types and resource management
Still imperative, mostly 1:1 with C functions

```
data Instruction
  = Inst [Prefix] Opcode [Operand]

data UD = ...    -- abstract type

newUD :: IO UD  -- deleted automatically

setInputBuffer :: UD -> ByteString -> IO ()
advance        :: UD -> IO (Maybe Word)
getInstruction :: UD -> IO Instruction
```

Hdis86.Pure

Top layer has the simplest interface:

```
disassemble
  :: Config
  -> ByteString
  -> [Instruction]
```

How do we get here from there?

C types, Haskell types

- C types are about machine representation
- Haskell types are about program structure, correctness

How to bridge the gap?

Callbacks

The C API:

```
type CInputHook = Ptr UD_t -> IO CInt
foreign import ccall set_input_hook
  :: Ptr UD_t -> FunPtr CInputHook -> IO ()
```

A better Haskell API:

```
type InputHook = IO (Maybe Word8)
setInputHook :: UD -> InputHook -> IO ()
```

- Replace -1 for EOF with `Nothing`
- Drop the `Ptr UD_t` arg — we can close over it anyway

Manufacturing function pointers

Special kind of import makes a `FunPtr`:

```
type CInputHook = Ptr UD_t -> IO CInt

foreign import ccall "wrapper"
  mkHook :: CInputHook -> FunPtr CInputHook
```

No C source; GHC does runtime codegen

Manage memory explicitly:

```
freeHaskellFunPtr :: FunPtr a -> IO ()
```

Library state

`udis86` stores all state in a `struct ud` (threadsafe!)

Store this along with other bookkeeping:

```
data Input
  = InNone
  | InHook (FunPtr C.CInputHook)
  | InBuf  (ForeignPtr Word8)

data State = State
  { udPtr   :: Ptr C.UD_t
  , udInput :: Input }

-- exported abstract
newtype UD = UD (MVar State)
```

MVars

`MVar T` is a mutable cell

At each point in time, it's empty or it holds a `T`

```
takeMVar :: MVar a      -> IO a
putMVar  :: MVar a -> a -> IO ()
```

{Take, put} blocks when {empty, full}

Exception-safe:

```
withMVar :: MVar a -> (a -> IO b) -> IO b
```

Concurrency

Use the MVar to guarantee atomicity of operations

```
-- internal helper function
withUDPtr
  :: UD -> (Ptr C.UD_t -> IO a) -> IO a
withUDPtr (UD s) f = withMVar s g where
  g (State ptr _) = f ptr
```

e.g.

```
setIP :: UD -> Word64 -> IO ()
setIP s w = withUDPtr s $ flip C.set_pc w
```

Writing your own higher-order helpers goes a long way!

Finalizers

Finalizer: runs sometime after last ref disappears

```
newUD :: IO UD
newUD = do
  p <- mallocBytes C.sizeof_ud_t
  C.init p
  s <- newMVar (State p InNone)
  addMVarFinalizer s (finalizeState s)
  return (UD s)

finalizeState :: MVar State -> IO ()
finalizeState = flip withMVar go where
  go st@(State ptr _)
    = setInput InNone st >> free ptr
```

Releasing old input source

When we set an input source, release the old one

```
-- internal helper function
setInput :: Input -> State -> IO State
setInput new_inpt st@(State _ old_inpt) = do
  case old_inpt of
    InHook fp  -> freeHaskellFunPtr fp
    InBuf  ptr -> touchForeignPtr ptr
    _          -> return ()
  return $ st { udInput = new_inpt }
```

ByteString internals

A `ByteString` is a raw byte array, with offset and length:

```
data ByteString = PS
  {-# UNPACK #-} !(ForeignPtr Word8)
  {-# UNPACK #-} !Int   -- offset
  {-# UNPACK #-} !Int   -- length
```

This allows efficient indexing, slicing, etc.
Uses `ForeignPtr` for garbage collection

Passing a ByteString to C

Can use a ByteString in C without a copy

```
setInputBuffer :: UD -> ByteString -> IO ()
setInputBuffer (UD s) bs
  = modifyMVar_ s go where
  go st@(State ud_ptr _) = do
    let (bs_ptr, off, len)
          = BS.toForeignPtr bs
    C.set_input_buffer ud_ptr
      (unsafeForeignPtrToPtr bs_ptr
        `plusPtr` off)
      (fromIntegral len)
    setInput (InBuf bs_ptr) st
```

Touch the ForeignPtr after last use, to delay finalizer

A little ugly here provides a nice API for users

Global state

Libraries with global state are harder!

Terrible:

```haskell
main = do
  initFoo
  ...
  cleanupFoo
```

Bad:

```haskell
main = withFoo $ do
  ...
```

Types as evidence

Adapted from `mersenne-random`:

```
-- exported abstract
data MTGen = MTGen

newMTGen :: Word32 -> IO MTGen
newMTGen seed = do
  dup <- c_get_initialized
  if dup == 0
    then do
      c_init_gen_rand (fromIntegral seed)
      return MTGen
    else error "only one gen per process!"

random :: (MTRandom a) => MTGen -> IO a
```

Global variables

Sometimes we need a global lock, init count, etc.
Handle this in C, or use the global variable hack:

```haskell
{-# NOINLINE globalState #-}
globalState :: MVar (Maybe State)
globalState
  = unsafePerformIO (newMVar Nothing)
```

Don't create polymorphic variables this way!
I wish GHC had JHC's "affine central IO" extension

Bound threads

GHC moves Haskell threads between OS threads
Can confuse C libs that use thread-local state

A "bound thread" uses a single OS thread for FFI

```
-- bind this thread
runInBoundThread :: IO a -> IO a

-- make a bound thread
forkOS :: IO () -> IO ThreadId
```

Does *not* affect where Haskell code executes

Laziness

Normally, Haskell separates execution from evaluation
Want evaluation of list cells to trigger execution of C calls

```
unsafeRunLazy :: UD -> IO a -> IO [a]
unsafeRunLazy ud get = fix $ \loop -> do
  n <- advance ud
  case n of
    Nothing -> return []
    Just _  -> liftA2 (:)
      get (unsafeInterleaveIO loop)
```

Not safe in general, but ok if get is just "observational"

Purity

The disassembler in aggregate is a pure function

```
disassemble
  :: Config
  -> ByteString
  -> [Instruction]
disassemble cfg bs = unsafePerformIO $ do
  ud <- newUD
  setInputBuffer ud bs
  setConfig      ud cfg
  unsafeRunLazy  ud (getInstruction ud)
```

Correctness of `unsafePerformIO` depends on the C library

Building FFI code with Cabal

Cabal will invoke `hsc2hs` automatically

```
$ ls Hdis86/
Types.hs
C.hsc
...

$ cat hdis86.cabal
...
library
  exposed-modules:
    Hdis86.Types
  , Hdis86.C
  , ...
```

Bundling a C library

```
extra-source-files:
    udis86-1.7/udis86.h, ...

flag external-udis86
  default: False
  description:
    Dynamically link external udis86

library
  if flag(external-udis86)
    extra-libraries: udis86
  else
    include-dirs: udis86-1.7
    c-sources:
      udis86-1.7/libudis86/udis86.c
```

Using the flag

```
$ cabal configure
Resolving dependencies...
Configuring hdis86-0.1...

$ cabal configure --flags=external-udis86
cabal: Missing dependency on a foreign library:
* Missing C library: udis86
[...]
use the flags --extra-include-dirs= and
--extra-lib-dirs= to specify where it is.
```

What's it good for?

Analyze machine code by pattern-matching

Let's detect register renaming:

```
data Constraint = Register :-> Register

(==>) :: [Bool] -> a -> Maybe a
ps ==> x = guard (and ps) >> Just x

operand :: Operand -> Operand
        -> Maybe [Constraint]
operand (Reg rx) (Reg ry) = Just [rx :-> ry]
```

Memory operands

Check equality of non-register fields

```
-- size, base reg, index reg, scale, offset
operand (Mem (Memory sx bx ix kx ox))
        (Mem (Memory sy by iy ky oy))

  = [sx == sy, kx == ky, ox == oy]
    ==> [bx :-> by, ix :-> iy]
```

`Nothing` means they differ beyond registers

Other operands

Immediate operands add no constraints

```
operand (Ptr px) (Ptr py)
  = [px == py] ==> []

operand (Imm ix) (Imm iy)
  = [ix == ix] ==> []
```

Different constructors means no match

```
operand _ _ = Nothing
```

Checking instructions

Check operands pairwise; collect constraints

```
inst :: Instruction
     -> Instruction
     -> Maybe [Constraint]

inst (Inst px ox rx) (Inst py oy ry) = do
  guard $ and [px == py, ox == oy,
               length rx == length ry]
  concat `fmap` zipWithM operand rx ry
```

Unifying constraints

Check for consistency:

```
unify :: [Constraint]
      -> Maybe (M.Map Register Register)

unify = foldM f M.empty where

  f m (rx :-> ry) = case M.lookup rx m of
    Nothing  -> Just (M.insert rx ry m)
    Just ry' -> [ry == ry'] ==> m
```

Invoking

Check in both directions

```
regMap :: [Instruction]
       -> [Instruction]
       -> Maybe (M.Map Register Register)

regMap xs ys = do
  cs <- concat `fmap` zipWithM inst xs ys
  let swap (x :-> y) = (y :-> x)
  _ <- unify $ map swap cs
  unify cs
```

Testing

```
main :: IO ()
main = print $ regMap (f prog_a) (f prog_b)
  where f = disassemble intel64

prog_a = BS.pack
 [0x7e,0x3a                  -- jle 0x3c
 ,0x48,0x89,0xf5             -- mov rbp, rsi
 ,0xbb,1,0,0,0               -- mov ebx, 0x1
 ,0x48,0x8b,0x7d,0x08]       -- mov rdi, [rbp+0x8]
prog_b = BS.pack
 [0x7e,0x3a                  -- jle 0x3c
 ,0x48,0x89,0xf3             -- mov rbx, rsi
 ,0xbd,1,0,0,0               -- mov ebp, 0x1
 ,0x48,0x8b,0x7b,0x08]       -- mov rdi, [rbx+0x8]
```

Results

Running our program:

```
$ runhaskell regmap.hs
Just (fromList
  [ (RegNone,    RegNone)
  , (Reg32 RBX, Reg32 RBP)
  , (Reg64 RBP, Reg64 RBX)
  , (Reg64 RSI, Reg64 RSI)
  , (Reg64 RDI, Reg64 RDI) ])
```

Change one jump target:

```
$ runhaskell regmap.hs
Nothing
```

This analysis is really incomplete. . .

Lessons learned

- Easy to call C libs; hard to make them feel like Haskell
- Use several layers of wrappers
- Make GHC manage your resources automatically
- Don't fear ugliness in your code, if it makes a pretty API
- Use unsafe operations, if they make the API better
    - but be careful!

Questions?

Slides online at `http://t0rch.org`