ART memo 6 (version 4)
Proposal for tracing primitive functions in nhc98 via the new FFI

Malcolm Wallace
University of York, UK

24th Oct 2000

1 Tracing primitives

Originally, tracing primitives, like normal primitives, were written by hand within the runtime system. Tracing primitives were more complex than normal ones, and therefore slightly more difficult to write correctly; this slowed down the process of introducing new facilities. Indeed, a major reason why the tracing compiler was very slow to support full Haskell'98 was the combination of this need to write separate interfacing code specially for tracing primitives, together with the tediously large number of them that were required.

The original primitives were largely simple calls to C library functions, but with a "wrapper" of extra interfacing code, also written in C, needed to extract arguments from the stack, unbox them, re-box the final result, and finally push it back on the stack.

The new common FFI is a mechanism that automates all of the "wrapping" for basic types, given the underlying C call that is being used.

We have migrated most of the existing primitives from the original nhc-specific interfacing mechanism based on the keyword `primitive`, to the new common FFI based on `foreign import`. In this *stage one*, the interface wrapping code (in C) for normal (non-tracing) primitives can largely be generated automatically by the FFI mechanism rather than relying too heavily on hand-written code.

We have also (*stage two*) extended the source-to-source tracing transformation to deal with foreign functions imported via the common FFI mechanism. When compiling for tracing, the more complex interface wrapping that deals with source references and trace arguments is generated automatically, in Haskell, in a similar manner to the tracing transformation used for normal functions. Hence, no extensions or changes to the common FFI mechanism itself were required, (and the source-to-source transformation therefore remains portable, in principle, to other systems that support the common FFI).

(An extra benefit of the transformational approach is that third-party library code, which often interfaces to C library code, will be able to use the tracing compiler directly without needing manual intervention to deal with the foreign code in a special way. Provided that the library interface is written entirely using the import mechanism of the common FFI, the tracing transformation proposed here should deal with it transparently.)

2 Common FFI

The new common FFI is not described in this memo - see the current URL

```
http://www.dcs.gla.ac.uk/fp/software/hdirect/ffi.html
```

although this is soon to be moved to a more generic location, linked from `haskell.org`.

3 Example

One might imagine that an FFI function declaration

```
foreign import cf :: Int -> Bool
```

might be transformed for tracing by exactly the same scheme as for normal functions, resulting in:

```
foreign import cf :: Trace -> R (Trace -> R Int -> (R Bool))
```

The first Trace is the trail corresponding the origin of the function; the second is for the application of the function. Each of the R values also encodes a trace – the first R contains the trail of the function; the second R contains the trail of the `Int` argument; and the third R contains the trail of the `Bool` result.

However this transformation, although correct for normal functions, is wrong for foreign functions. For one thing, it is too higher-order, and consequently there is no way for the foreign function to extract its arguments! (For instance, notice that the argument is not actually given to the foreign function – instead it computes a result function that could take the `Int` as one of *its* arguments.) Also, since the foreign function is written in C, and probably belongs to a library whose source cannot be changed, the foreign function must in fact retain its original type.

Hence, the source-to-source transformation must have a different rule for foreign functions. It must introduce a Haskell wrapper around the foreign call, dealing with the change of argument and result type representation, and correctly threading the trace structures into the result.

```
foreign import "cf" cf' :: Int -> Bool
cf :: Trace -> R (Trace -> R Int -> R Bool)
cf t = let tf = Nm t "cf" in
   R (\t0 (R i _)-> let v = cf' i in R v (Sat t0 (Nm t0 v))) tf
```

The Haskell wrapper steals the name of the original foreign function, and thus the foreign function must be renamed on the Haskell side of the interface, although of course the original name remains on the C side. The original foreign function is called in a fully saturated application, in order that it can get all its arguments at once in an accessible form.

One issue to consider carefully, is the representation of user-defined datatypes after the tracing transformation. Any values of constructed types that are passed across the FFI have a representation not expected by the foreign code. However, the common FFI explicitly excludes such user-defined types, permitting only basic types (i.e. types without constructors). Hence, the simple transformation given here is safe.

4 Transformation Rules

The formal transformation rules for a function imported via the FFI are as follows. First, the change in name of the real foreign function, and the introduction of a new wrapper function:

$$\mathcal{F}[\![\texttt{foreign import } f :: a_0 \rightarrow a_1 \rightarrow \ldots \rightarrow a_n]\!] \Rightarrow \texttt{foreign import "}f\texttt{" } f' :: a_0 \rightarrow a_1 \rightarrow \ldots \rightarrow a_n$$

$$\mathcal{F}[\![\texttt{foreign import } f :: a_0 \rightarrow a_1 \rightarrow \ldots \rightarrow a_n]\!] \Rightarrow f = \texttt{prim}_n\ f'\ \mathcal{N}[\![f]\!]$$

And second, the definition of a combinator that collects the primitive arguments one-by-one and passes them to the foreign function.

```
prim_n f' nm t =
   R ( λ t_1 (R e_1 t_{e_1}) →
      R ( λ t_2 (R e_2 t_{e_2}) →
         ...
            R ( λ t_n (R e_n t_{e_n}) →
               let (v,vt) = enter (Nm t_n (toNm v)) (f' e_1 e_2 ...e_n) in
               R v (Sat t_n vt))
            t_{n-1})
         ...
      t_1)
   (Nm t nm)
```

Finally, we note that these rules could form the basis of a scheme to interface traced Haskell functions with untraced Haskell functions.

5 Exports

So far, we have talked only of primitive functions imported into Haskell. However, the common FFI also allows for Haskell functions to be exported to the outside world. It would be nice to allow exports of traced versions of functions. Unfortunately, this is rather difficult since the foreign caller expects the Haskell function to have its original type, but the transformed version has a rather different type. It is not clear how a wrapper function could "invent" a trace from nothing to pass into an exported function.

Since exported functions are much rarer than imported functions, we leave consideration of this question for the future.

6 Foreign I/O

It is not immediately clear how the transformation scheme given above extends from pure foreign functions to impure foreign I/O actions. For this, we rely on a particular implementation model for the Haskell I/O monad.

Given the following definition of the I/O type as an encapsulated function from the real world to a value:

```
data IO a = IO (World -> Either IOError a)
```

we assume the existence of a family of functions that can lift non-I/O functions of different arities into this representation. For instance, the following function

```
_mkIOok2 :: (c->b->a) -> (c->b->IO a)
_mkIOok2  fn = \\a b-> IO (\\_->Right $! fn a b)
```

lifts its first argument (a normal function of arity 2) to become an I/O action, also of arity 2, but whose result is within the I/O monad.

Now, when we see a common FFI declaration of a foreign I/O action, we insist that it be rewritten (by an automatic transformation) into a form like that just described. For instance,

```
foreign import getRandom :: Int -> IO Int
```

would be re-formulated as

```
foreign import "getRandom" getRandom# :: Int -> Int
getRandom :: Int -> IO Int
getRandom = _mkIOok1 getRandom#
```

Here, we deliberately change the foreign I/O action into what looks like a pure function, although of course it is actually impure and causes side-effects. However, this is ok because the only place it is used is inside the `_mkIOok1` wrapper, which immediately lifts it back inside the IO monad – the name of the side-effecting function is not visible to the outside world.

Through this "worker/wrapper" split, we ensure that the foreign I/O action, at least locally, has a type that enables the tracing transformation to be applied to it as if it were a pure function.