Counting total-discharge expressions in Core Haskell

Maciej Zielenkiewicz and Aleksy Schubert

November 25, 2016

Our aim is to count how many of the functions in a Haskell program are in total discharge form. We will process one file function-by function using the GHC API. First the imports:

```haskell
module Main where

import DynFlags
import GHC
import GHC.Paths
import CoreSyn
import Id
import Type
import Kind
import Unique
import Name
import MonadUtils
import Outputable
import qualified Data.Map as Map
import Data.List
import Text.Printf
import System.Environment

import System.IO.Unsafe
```

We need to test types for equality and introduce ordering. Those provided with haskell have some edge cases but are good enough for our uses.

```haskell
instance Eq Type where
  t1 ≡ t2         = eqType t1 t2

instance Ord Type where
  compare t1 t2 = cmpType t1 t2
```

During processing of a bind we maintain a map from type to last introduced variable of that type. If we encounter a usage of a variable which is different from the one recorded in the map then the whole function is not in total discharge form. If we encounter a variable of a type which was never recorded in our map we assume that the variable comes from environment; we do not maintain full record of the environment so as not to handle the complexity of module system of Haskell. Together with the map we'll introduce function for inseting into the map.

$$\begin{array}{l}
\textbf{type}\ \mathit{VMap} = \mathit{Map.Map\ Kind\ Unique.Unique} \\
\mathit{pushId} :: \mathit{VMap} \to \mathit{Id} \to \mathit{VMap} \\
\mathit{pushId\ m\ id} = \mathit{Map.insert}\ (\mathit{idType\ id})\ (\mathit{idUnique\ id})\ m
\end{array}$$

The basic „processing unit" is a syntax element. We process a syntax element being given an initial type-to-variable map, potentially process its subelements and return a bool — true if the element was in tdf. When combining results of processing two syntax elements we use the function `combineAccResult`. The function `p` is defined by cases.

The usage of a variable is checked with the map, if type is present in the map then the variable used must be the same as that recorded in the map.

$$\begin{array}{l}
p :: \mathit{VMap} \to \mathit{Expr\ CoreBndr} \to \mathit{Bool} \\
p\ m\ (\mathit{Var\ id}) \\
\quad \mid \mathit{idType\ id}\ `\mathit{Map.member}`\ m = \mathit{mMap}.!\ \mathit{idType\ id} \equiv \mathit{idUnique\ id} \\
\quad \mid \mathit{otherwise} = \mathit{True} \quad \text{-- if it was not a our variable it comes from environment}
\end{array}$$

A literal is always good.

$$p\ m\ (\mathit{Lit}\ \_) = \mathit{True}$$

To check application we need just to check both parts.

$$p\ m\ (\mathit{App}\ (\mathit{exprb})\ (\mathit{argb})) = p\ m\ \mathit{exprb}\ `\mathit{combineAccResult}`\ p\ m\ \mathit{argb}$$

To check λ-abstraction we add variable to map and check the term with the new map.

$$p\ m\ (\mathit{Lam\ id}\ (\mathit{exprb})) = p\ (\mathit{pushId\ m\ id})\ \mathit{exprb}$$

A binding is a treated as a more sophisticated λ-expression with multiple variables. We translate non-recursive lets to recursive lets so that we really only have to deal with one case.

```
p m (Let (NonRec btype bexpr) exprb) = p m $ Let (Rec [(btype, bexpr)]) exprb
p m (Let (Rec letlist) exprb) =
    -- add ids in binds
  let m' = foldl pushId m $ map fst letlist
    -- check exprb
      res' = p m' exprb in
  foldl combineAccResult res' $
    -- check expressions in binds
    flip map letlist $
      λ(b, bexpr) → p (pushId m b) bexpr
```

Procesing a case expression is combination of processing the expression and all clauses. Note (FIXME) that it seems that GHC doesn't seem to use `bndr` and `btype` so we do not handle them.

```
p m (Case (bexpr) bndr btype cases) =
    -- we process bexpr first:
  let res' = p m bexpr in
    -- and all cases:
  foldl combineAccResult res' $
    flip map cases $
      λ(_, bndrs, bexpr) →
        let m' = foldl pushId m bndrs in
        p m' bexpr
```

Some syntax parts are totally uninteresting to us:

```
p m (Cast exprb coercion) = p m exprb   -- GHC handles the typechecking for us
p m (Tick _ exprb) = p m exprb   -- we do not need special profiling features
p m (Type _) = True   -- a type literal is not a variable
p m (Coercion _) = True   -- GHC handles the typechecking for us
combineAccResult b1 b2 = b1 ∧ b2
```

To interace with top-level elements we need to have a function processing a whole `CoreBind`, as program elements are `CoreBinds`. We return one boolean for each bind.

```
  -- process a single bind for counting
process :: CoreBind → [Bool]
process (NonRec cbndr expr) = [p Map.empty expr]
process (Rec l) =
    -- we process each bind separately, but combine the result
  concat $ map (λ(cbndr, expr) → process $ NonRec cbndr expr) l
```

We take parameters and set up GHC environment. First argument is the name of module to be compiled and the rest are options which are passed to GHC.

```haskell
-- process whole program into Core, return result of checking every bind
main = defaultErrorHandler defaultFatalMessager defaultFlushOut $ do
  gargs <- getArgs
  let (args, opts) = partition (isSuffixOf ".hs") gargs
  runGhc (Just libdir) $ do
    dflags <- getSessionDynFlags
    (ndflags, _, _) <- liftIO $ parseDynamicFlags dflags $ map noLoc opts
    setSessionDynFlags $ ndflags
```

compile the given file to Core Haskell and extract the binds:

```haskell
(flip mapM_) args $ \arg -> do
  cm <- compileToCoreModule arg
  let cp = cm_binds cm
```

we process every bind separately with function process. This means that, for every function defined in a module, we treat all other functions in that modules as environmental variables.

```haskell
let lines = concat $ map process cp
let good = length $ filter (\x -> x) lines
  bad = length $ filter not lines
    -- good bad
liftIO $ printf "||||||RESULT %d\t%d\n" good bad
```

The last slice is a function which can be used to output anything that is `Outputable`, although we don't need it right now.

```haskell
-- Outputs any value that can be pretty-printed using the default style
output :: (GhcMonad m, MonadIO m) => Outputable a => a -> m ()
output a = do
  dfs <- getSessionDynFlags
  let style = defaultUserStyle
  let cntx = initSDocContext dfs style
  liftIO $ print $ runSDoc (ppr a) cntx
```