Safe Haskell
Goals

• Allow implementation of custom security policies for running untrusted Haskell code.

• Make Safe Haskell as unobtrusive as possible.
What is safety?

• The types of symbols exported from untrusted modules can be trusted. “Types don’t lie.”

• Untrusted code cannot break the encapsulation of other modules.
Components of Safe Haskell

- Safe subset of the language that can be used to write untrusted modules.
- System for determining which modules can be safely imported by untrusted code.
- Trust system for packages.
Two ways for a module API to be safe:

• Machine verifiable:
  – Safe subset of Haskell
  – Only safe imports

• Dependent on trust:
  – Provides a safe API
  – End user trusts module’s package
3 module categories

- XSafe
- XTrustworthy
- XUnsafe
A
-XUnsafe

B
-XTrustworthy

C
-XSafe
A
-XUnsafe

B
-XTrustworthy

C
-XSafe
Safe Language

• Type Safety
• Referential Transparency
• Module Encapsulation
• Modular Reasoning
• Semantic Consistency
Safe Language

• Disallowed language extensions:
  – Template Haskell
  – Generalized Newtype Deriving
  – RULES pragmas

• Restricted language extensions:
  – FFI imports must have IO return types.
  – Overlapping instances must be in the same module.
  – Data.Typeable instances must be derived.
    (before GHC 7.8)
{-# LANGUAGE Trustworthy #-}
module PluginAPI (
  RIO(), runRIO, rioReadFile, rioWriteFile
) where

-- Notice that symbol UnsafeRIO is not exported
-- from this module!
newtype RIO a = UnsafeRIO { runRIO :: IO a }

instance Monad RIO where
  return = UnsafeRIO . return
  (UnsafeRIO m) >>= k = UnsafeRIO $ m >>= runRIO . k

-- Returns True iff access is allowed to file name
pathOK :: FilePath → IO Bool
pathOK file = {- Implement some security policy -}

rioReadFile :: FilePath → RIO String
rioReadFile file = UnsafeRIO $ do
  ok ← pathOK file
  if ok then readFile file else return ""

rioWriteFile :: FilePath → String → RIO ()
rioWriteFile file contents = UnsafeRIO $ do
  ok ← pathOK file
  if ok then writeFile file contents else return ()

(Terei et al., 2012)
vector

bytestring

my package

aeson
Questions
Breaking module encapsulation with Generalized Newtype Deriving

```haskell
module MinList (MinList, newMinList, insertMinList)
  where

  data MinList a = MinList a [a] deriving Show

  newMinList n = MinList n []

  insertMinList s@(MinList m xs) n
    | n > m     = MinList m (n:xs)
    | otherwise = s
```

(Terei et al., 2012)
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

module MinList where

import MinList

class IntIso t where
  intIso :: c t \rightarrow c Int

instance IntIso Int where
  intIso = \_ \rightarrow \_

newtype I = I Int deriving (Eq, IntIso)
  -- we reverse the usual comparison order
instance Ord I where
  compare (I a) (I b) = compare b a

nums = [1,4,0,1,-5,2,3]

goodList :: MinList Int
goodList = foldl insertMinList
  (newMinList $ head nums)
  (tail nums)

badList :: MinList Int
badList = intIso $ foldl (\x y \rightarrow insertMinList x $ I y)
  (newMinList $ I $ head nums)
  (tail nums)

main = do
  print goodList
  print badList

When running this code the output is:

  MinList 1 [3,2,4]
  MinList 1 [-5,0]

(Terei et al., 2012)
References