Modularity and Backpack

Maciek Makowski (@mmakowski)

19th January 2015

MirageOS

TODO picture of unikernel

MirageOS

TODO picture of HTTP app with dependencies

Exhibit 1

MirageOS

> *The compiler heavily emphasises static type checking, and the resulting binaries are fast native code with no runtime type information and the module system is among the most powerful in a general-purpose programming language in terms of permitting flexible and safe code reuse and refactoring.*

– Technical Background of MirageOS

ML Modules

Structures

```
structure IntInteger =
  struct
    type integer = int
    val zero = 0
    fun succ n = n + 1
    fun add a b = a + b
    fun mul a b = a * b
  end
```

ML Modules

Signatures

```
signature INTEGER =
  sig
    type integer
    val zero: integer
    val succ: integer -> integer
    val add: integer -> integer -> integer
    val mul: integer -> integer -> integer
  end
```

ML Modules

Functors

```
functor RationalFun(I: INTEGER) =
  struct
    type rational = I.integer * I.integer
    fun nom (n, _) = n
    fun denom (_, d) = d
    fun add (n1, d1) (n2, d2) =
      (I.add (I.mul n1 d2) (I.mul n2 d1),
       I.mul d1 d2)
  end
```

```
structure IntRational = RationalFun(IntInteger)
```

ML Modules

TODO: Mirage app diagram

Scala

trait ~ signature

```
trait IntegerSig {
  type Integer
}
```

object ~ structure

```
object IntInteger extends IntegerSig {
  type Integer = Int
}
```

class ~ functor

```
class RationalFun[I <: IntegerSig](i: I) {
  type Rational = (I#Integer, I#Integer)
}
```

Exhibit 2

tagstream-conduit

```
data Dec builder string = Dec
  { decToS      :: builder -> string
  , decBreak    :: (Char -> Bool) -> string ->
                   (string, string)
  , decBuilder  :: string -> builder
  , decDrop     :: Int -> string -> string
  , decEntity   :: string -> Maybe string
  , decUncons   :: string -> Maybe (Char, string)
  }
```

Exhibit 3

tagsoup

```
class (Typeable a, Eq a) => StringLike a where
    empty      :: a
    cons       :: Char -> a -> a
    uncons     :: a -> Maybe (Char, a)
    toString   :: a -> String
    fromString :: String -> a
    -- [...]
```

Backpack

Modules

```
module IntegerInt where

type Integer = Int

zero = 0
succ = (+1)
add = (+)
mul = (*)
```

Backpack

Signatures

```
module IntegerSig where

data Integer

zero :: Integer
succ :: Integer
add  :: Integer -> Integer -> Integer
mul  :: Integer -> Integer -> Integer
```

Backpack

Modules

```
module Rational where

import qualified IntegerSig as I

data Rational = R I.Integer I.Integer

nom (R n _) = n
denom (R _ d) = d
add (R n1 d1) (R n2 d2) =
  R (I.add (I.mul n1 d2) (I.mul n2 d1))
    (I.mul d1 d2)
```

Backpack

Cabal packages

Integer implementation package

name:	integer-int
exposed-modules:	IntegerInt

Backpack

Cabal packages

Integer signature package

name:	integer-sig
version:	1.0
indefinite:	True
exposed-signatures:	IntegerSig

Backpack

Cabal packages

Rational "functor" package

```
name:                rational
indefinite:          True
build-depends:       integer-sig-1.0
exposed-modules:     Rational
```

Backpack

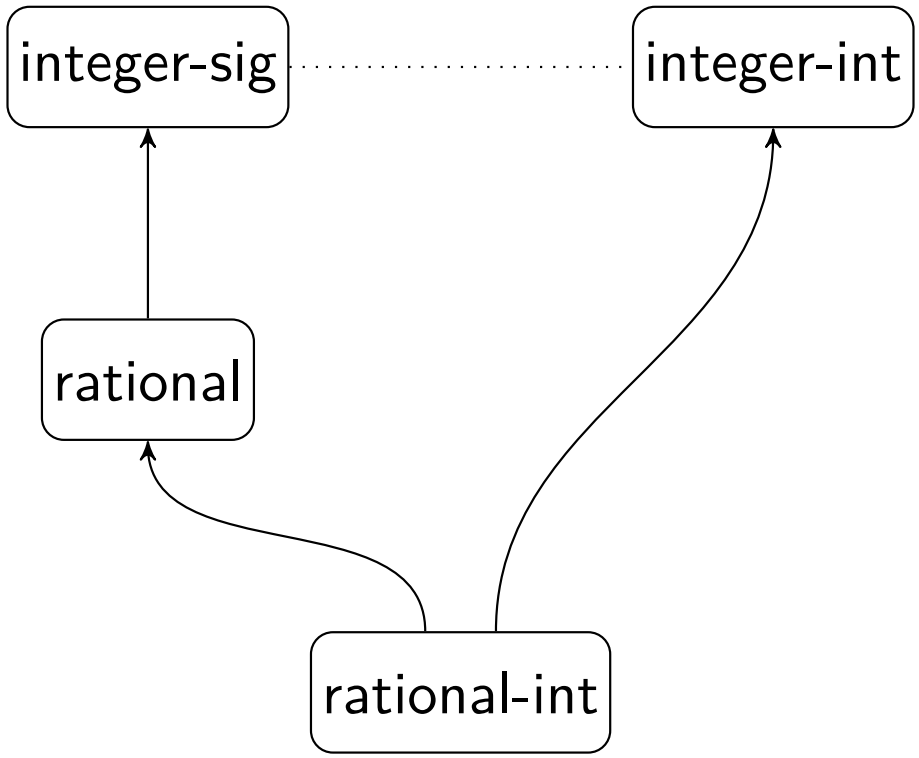
Mixing in

Rationals based on Integers

```
name:                    rational-int
build-depends:
   integer (IntegerInt as IntegerSig),
   rational
reexported-modules: Rational as RationalInt
```

Backpack

Package hierarchy



Instantiation Semantics

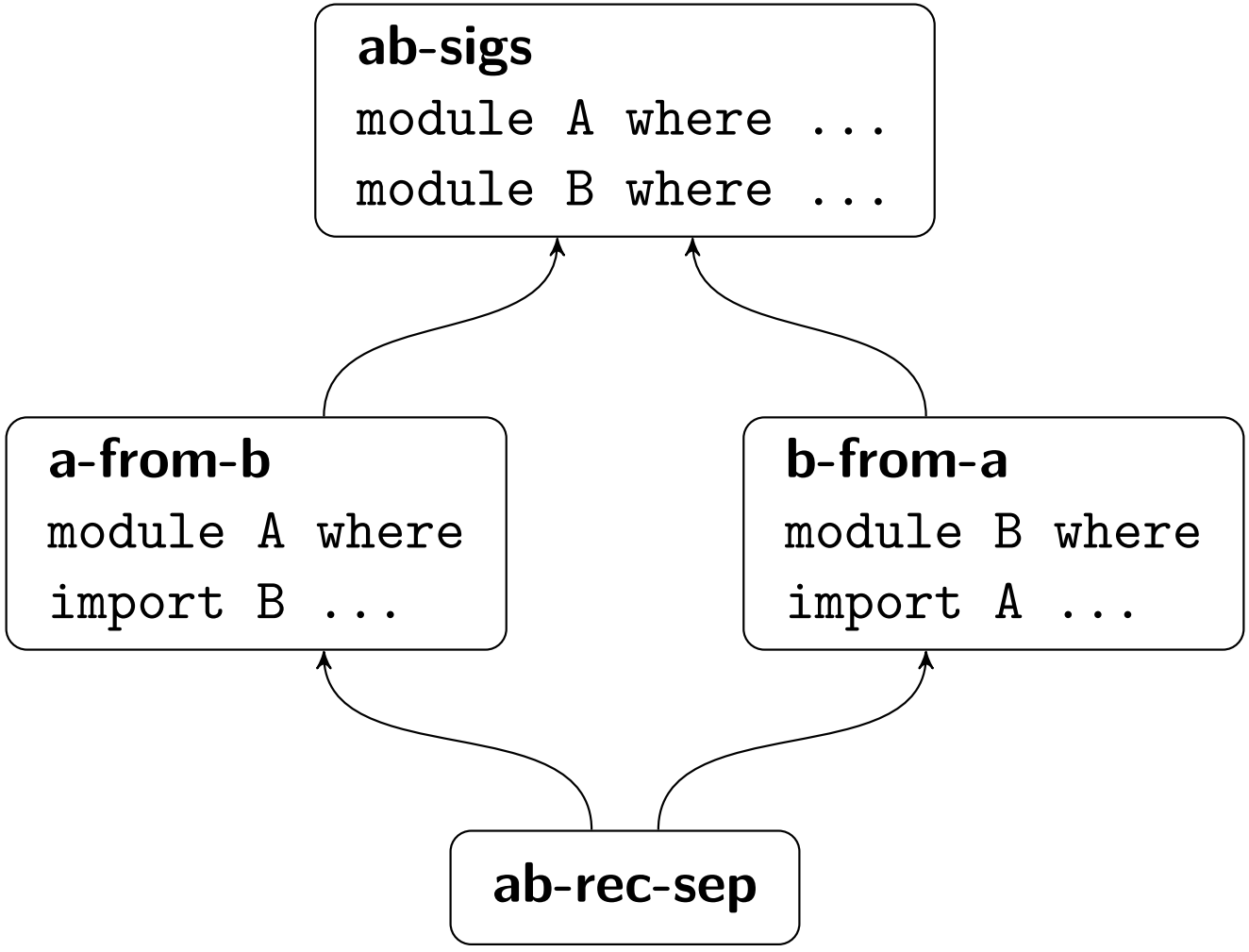
```
functor RationalFunD(I: INTEGER) =
  struct
    datatype rational =
      R of I.integer * I.integer
    val zero = R (I.zero, I.succ I.zero)
  end
structure R1 = RationalFunD(IntInteger)
structure R2 = RationalFunD(IntInteger)
```

Is R1.rational the same type as R2.rational?

- *generative*: no
- *applicative*: yes

Backpack

Recursive linking



Module Systems

Features