Transforming Haskell for Tracing

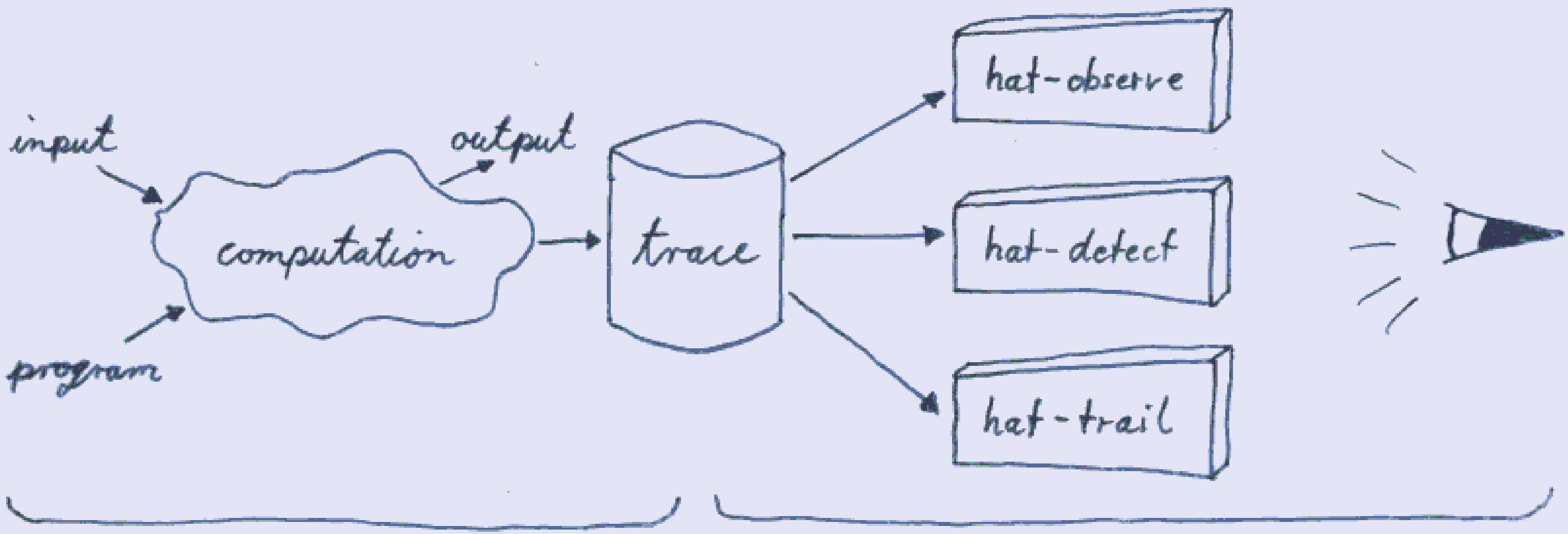
Olaf Chitil

Colin Runciman   Malcolm Wallace

The University of York

Tracing with Hat

applications: program comprehension and debugging



trace as data structure liberates from the time arrow of the computation

Trace Generation through Transformation

was: Haskell source → nhc98 → self-tracing executable

now: Haskell source → hat-trans → Haskell source → Haskell compiler → self-tracing executable

Hat library → Haskell compiler

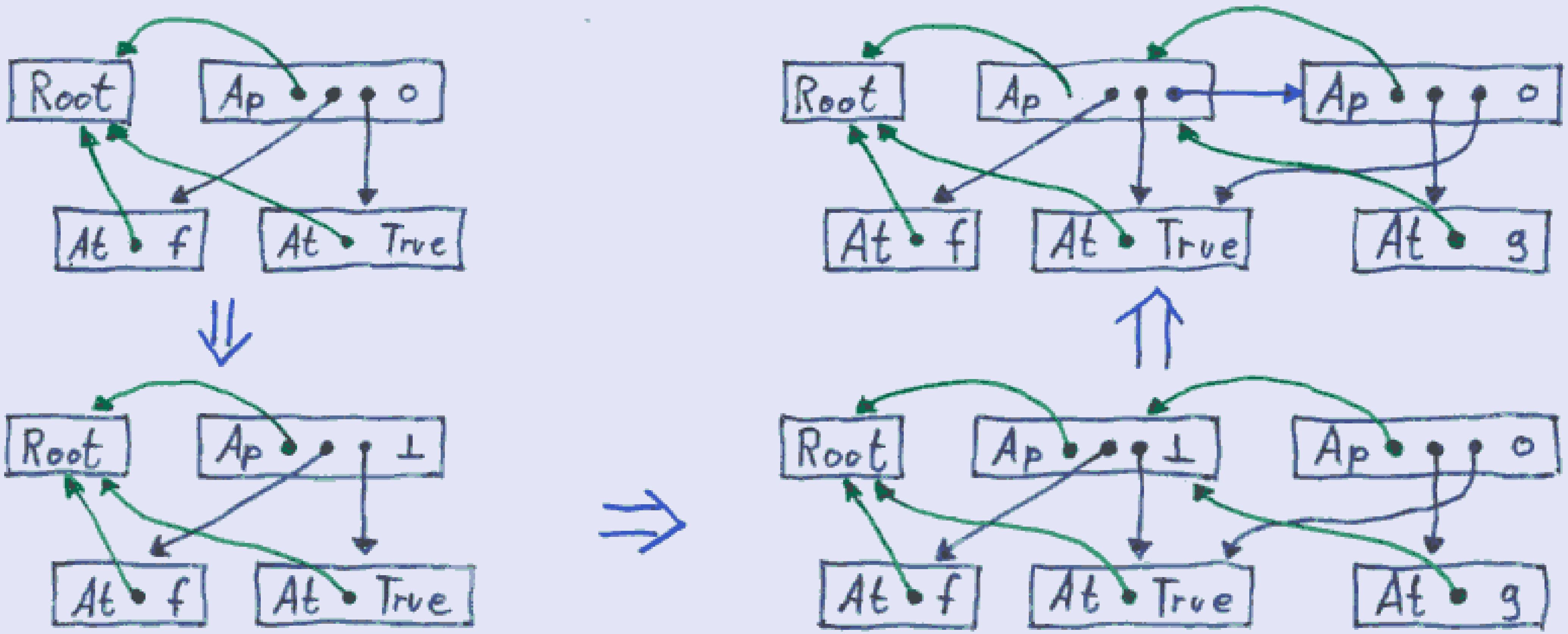
Why?

- comprehensible
- simplifies development
- combinable with different compilers

What does the trace of a reduction look like?

$$f\ x = g\ x$$

$$f\ True \Rightarrow g\ True$$

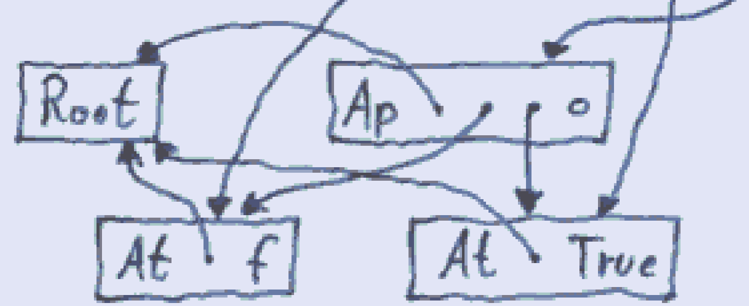


Do we use the IO-monad for writing the trace?

Do we use reflection to obtain meta-information such as identifier names?

How can evaluation of original expression and writing of trace be kept in synchrony?

```
data R a = R a RefExp        f True  ~>  R (R f ) (R True )
```



Does the transformation need type information?

Hat Library

- primitive combinators write trace file : mkAt, mkAp, entRedex, updResult, ..
- high-level combinators simplify the tranformation : ap, ...

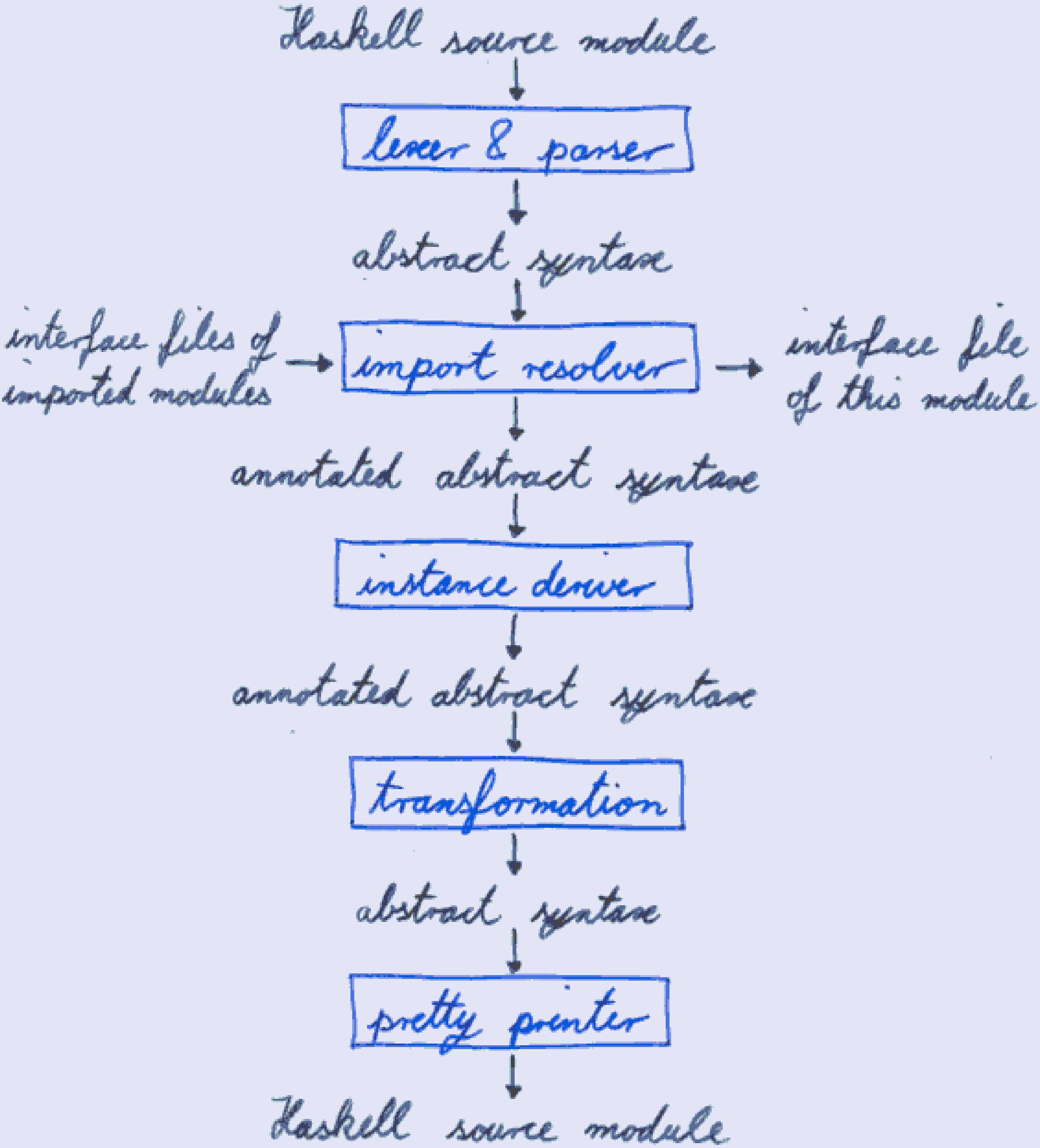
N-ary application and abstraction :

```
ap1  :: RefSrcPos -> RefExp -> R (Fun a z) -> R a -> R z
ap2  :: RefSrcPos -> RefExp -> R (Fun a (Fun b z)) -> R a -> R b -> R z

fun1 :: RefAtom -> RefSrcPos -> RefExp -> (R a -> RefExp -> R z) -> R (Fun a z)
fun2 :: RefAtom -> RefSrcPos -> RefExp -> (R a -> R b -> RefExp -> R z)
          -> R (Fun a (Fun b z))
```

only after evaluation of function know if application is saturated

Hat-trans



Haskell source module
lexer & parser
abstract syntax
interface files of imported modules
import resolver
interface file of this module
annotated abstract syntax
instance deriver
annotated abstract syntax
transformation
abstract syntax
pretty printer
Haskell source module

Transformation

```
f :: Bool -> Bool          ~>   f :: RefExp -> Fun Bool Bool
f x = g x                       f p = R (Fun (\x r -> ap r (g r) x))
                                        (mkAt p "f")

f True                     ~>   case (let p = mkRoot
                                      in ap p (f p) (R True (mkAt p "True"))) of
                                  R x _ -> x
```

```
data R a = R a RefExp          <- augmented expressions

newtype Fun a b = Fun (R a -> RefExp -> R b)

ap :: RefExp -> R (Fun a b) -> R a -> R b
ap p (R (Fun f) rf) a@(R _ ra) =
  let r = mkAp p rf ra
  in R (entRedex r `seq` case f a r of R y ry -> updResult r ry `seq` y) r
```

Transformation: Types

```
   data Point = P Integer Integer

~> data Point = P (R Integer) (R Integer)


   sort :: Ord a => [a] -> [a]

~> gsort :: Ord a => RefSrcPos -> RefExp -> R (Fun (List a) (List a))
```

no defaulting!

Transformation: Pattern-Matching

```
reverse []     = ...
reverse (x:xs) = ...
```

```
greverse p j = fun1 areverse p j hreverse

hreverse (R Nil) j = ...
hreverse (R (Cons fx fxs) _) j = ...
```

- k and n+k ⇝ explicit ==, fromInteger, ...
- irrefutable pattern ⇝ local pattern binding
- guard ⇝ continuation style

How does the transformation effect performance/complexity?

- sharing of constants

  monomorphic restriction enables identification of pseudo constants

- tail recursion

How do we handle errors?

Catching errors

- extend incomplete patterns
- define error specially
- catch Control-C and arithmetic errors with C signals
- use Haskell IO-catch

→ library variants for ghc and nhc98

The Trace Stack

last entered redex without result raised error

How do we call primitive functions?

```
toChar   :: RefExp -> R Char -> Char
fromChar :: RefExp -> Char -> R Char

toList :: (RefExp -> R a -> b) -> RefExp -> R (List a) -> [b]

toString :: RefExp -> R String -> Prelude.String
toString = toList toChar

toFun :: (RefExp -> c -> R a) -> (RefExp -> R b -> d) -> RefExp
      -> R (Fun a b) -> (c -> d)

foreign import haskell "Char.isUpper" isUpper :: Char -> Bool
```

How do we implement trusted/untraced code?

Wrapping does not work

- `(++) :: [a]->[a]->[a]` ⇝ `... fromList (toList x ++ toList y) ...`

- `elem :: Eq a => a -> [a] -> Bool`
  `gelem :: Eq a => RefSrcPos -> RefExp -> R (Fun a (Fun (List a) Bool))`

Combinators for Trusting

- record: call, result, traced subcomputation
- demand-driven recording of result

Conclusions

- Tracing through Program Transformation
  - Hood : library + manual transformation
  - Freja : a special compiler
  - a Haskell interpreter
- Transforming Haskell (powerful but large with irregularities)
- Compiler independent

Hat http://www.cs.york.ac.uk/fp/hat