Transforming Haskell for Tracing

Olaf Cheitel
Colin Runciman  Malcolm Wallace

The University of York
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

**Why?**
- Comprehensible
- Simplifies development
- Combinable with different compilers
What does the trace of a reduction look like?

\[ f \times = g \times \]
\[ f \ True \Rightarrow g \ True \]
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 \( R a = R a \text{RefExp} \)

\[ f \text{True} \mapsto R (R f) (R \text{True}) \]

Does the transformation need type information?
Hat Library

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

N-ary application and abstraction:

\[
\begin{align*}
ap 1 & : \text{RefSrcPos} \to \text{RefExp} \to R (\text{Fun} \ a \ z) \to R a \to R z \\
ap 2 & : \text{RefSrcPos} \to \text{RefExp} \to R (\text{Fun} \ a \ (\text{Fun} \ b \ z)) \to R a \to R b \to R z \\
\text{fun1} & : \text{RefAtom} \to \text{RefSrcPos} \to \text{RefExp} \to (R a \to \text{RefExp} \to R z) \to R (\text{Fun} \ a \ z) \\
\text{fun2} & : \text{RefAtom} \to \text{RefSrcPos} \to \text{RefExp} \to (R a \to R b \to \text{RefExp} \to R z) \\
& \qquad \to R (\text{Fun} \ a \ (\text{Fun} \ b \ z)) \\
\end{align*}
\]

only after evaluation of function known if application is saturated.
Haskell source module
\[ \downarrow \]
leaver & parser
\[ \downarrow \]
abstract syntax
\[ \downarrow \]
interface files of imported modules
\[ \rightarrow \]
import resolver
\[ \downarrow \]
interface file of this module
\[ \rightarrow \]
annotated abstract syntax
\[ \downarrow \]
instance derive
\[ \downarrow \]
annotated abstract syntax
\[ \downarrow \]
transformation
\[ \downarrow \]
abstract syntax
\[ \downarrow \]
pretty printer
\[ \downarrow \]
Haskell source module

hat-trans
Transformation

\[
f :: \text{RefExp} \to \text{Fun} \to \text{Bool} \\
f x = \text{g} x \\
f \text{True} \\
\]

\[
f p = R \left( \text{Fun} \left( (\lambda x \Rightarrow \text{ap} \ 	ext{r} \ (\text{g} r) \ x) \right) \right) \\
\text{let } p = \text{mkRoot} \\
in \text{ap} \ p \ (\text{fp}) \ (R \ \text{True} \ \text{mkAt} \ p \ \text{"True"}) \\
of \]

\[
\text{case} \ \\n\]

\[
x \mapsto x \\
\]

\[
\rightarrow \ \\
\text{augmented expressions} \\
\]

\[
data \ R \ a = R \ a \ \text{RefExp} \\
\]

\[
\text{newtype} \ \text{Fun} \ a \ b = \text{Fun} \ (R \ a) \rightarrow R \ b \\
\]

\[
ap :: \text{RefExp} \rightarrow R \ (\text{Fun} \ a \ b) \rightarrow R \ a \rightarrow R \ b \\
\]

\[
\text{let } r = \text{mkAp} \ p \ (\text{r} a) \\
in \text{R (contRedex r seq) case far of R y ry updResult ry seq x} \\
\]
Transformation: Types

```haskell
data Point = P Integer Integer

\rightarrow data \text{Point} = P (R \text{Integer}) (R \text{Integer})
```

```haskell
\text{sort} :: \text{Ord} \ a \Rightarrow [a] \rightarrow [a]
```

```haskell
\rightarrow \text{gsort} :: \text{Ord} \ a \Rightarrow \text{RefSrcPos} \rightarrow \text{RefExp} \rightarrow R (\text{Fun} (\text{List} a) (\text{List} a))
```

\text{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) j = ...

• k and n+k ~ explicit ==, fromInteger, ...
• irrefutable pattern ~ local pattern binding
• guard ~ continuation style
How does the transformation affect 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 10-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/untrusted code?

Wrapping does not work

* \((++): [a] \rightarrow [a] \rightarrow [a]\) \Rightarrow \text{... fromList (toList x ++ toList y)...}

* \(\text{elem}: \text{Eq } a \Rightarrow a \Rightarrow [a] \Rightarrow \text{Bool}\)

\(\text{gelem}: \text{Eq } a \Rightarrow \text{Ref SrcPos} \Rightarrow \text{Ref Exp} \Rightarrow R (\text{Fun } a (\text{Fun } (\text{List } a) \text{ Bool}))\)

Combinators for Trusting

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

- Tracing through Program Transformation
  - Flood: library + manual transformation.
  - Freja: a special compiler
  - a Haskell interpreter

- Transforming Haskell (powerful but large with irregularities)

- Compiler independent

at http://www.cs.york.ac.uk/~fp/hat