W-Hat?

a query language for Hat

PhD work by Tom Shackell
Trace is a graph

parent

result

fun

arg 1

arg 2

name (function)
constructor
basic value

2
Pure Trace Operations

parent :: Node -> Node
result :: Node -> Node
numArgs :: Node -> Int

fun :: Node -> Node
name :: Node -> String
arg :: Int -> Node -> Node
intValue :: Node -> Int
Trace Comprehensions

\{ n \mid \text{name (fun n) == "insert"} \\
\quad \text{&& numArgs n == 2} \} \\

\{ n \mid \text{intValue (result}^\wedge n) == 7 \}
Compiling queries

\[
\{ \ n \mid \text{name (fun n) == "insert"} \\
\&\& \text{numArgs n == 2} \}
\]

\[
[ \ n \mid \text{n <- trace,} \\
\quad \text{isJust (do f <- fun n} \\
\quad \quad \text{m <- name f} \\
\quad \quad \text{guard (m=="insert")} \\
\quad \quad \text{a <- numArgs n} \\
\quad \quad \text{return (a==2)} \\
\quad \text{)}} \\
\]

Meta-operations

(==) :: a -> a -> Bool
(@=) :: Node -> Node -> Bool
(*=) :: Node -> Node -> Bool
(~) :: Node -> Node -> Bool

(&&) :: Bool -> Bool -> Bool
(|||) :: Bool -> Bool -> Bool

(^) :: (a->Maybe a) -> a -> Maybe a
Queries are tedious

Find “insert 1 (3:_)”

\{
  n | name (fun n) == “insert”
  && numArgs n == 2
  && intValue (arg 1 n) == 1
  && fun (arg 2 n) == (:)
  && intValue (arg 1 (arg 2 n))
  == 3
\}
Pattern-matching

\{ (n,y) \mid \text{match } n \text{ of} \}
\begin{align*}
    &\left[ \text{insert 1 } (3:y:_) \right] \\
\end{align*}
\}

n = \text{insert 1 } [3,4] \\
y = 4

n = \text{insert 1 } (3:\perp:5:[]) \\
y = \perp
Embedding of Hat tools

Hat-observe

observe pat context = map display
{ (n,r) | match n of [[ pat ]
    && p == parent n
    && match p of [[ context ]
    && r == result\^ n
}
Embedding of Hat tools

Hat-stack

stack err = loop err

where loop n =

do display n

  [x] <- { p | p == parent n }  
loop x
Embedding of Hat tools

Hat-trail

trail err = loop err

where loop n =

do display n

sub <- interactive n

[x] <- { p | p == parent sub }

loop x
Embedding of Hat tools

Hat-detect

detect main = interactive (tree main)

where

tree n = EDT n (children n)

children n =

{ c | n == parent c }
Going further...
Computation over queries

\[
\{ \text{xs} \mid \text{length} \left[\text{sort} \text{xs}\right] \\
/\!/= \text{length} \text{xs}
\}
\]

\[
\text{xs} = [1,2]
\]

\[
\{ (\text{x,ys}) \mid \text{length} \left[\text{insert} \text{x ys}\right] \\
/\!/= \text{length} \text{ys} + 1
\}
\]

\[
\text{x} = 1, \text{ys} = [2]
\]
Alternatively...

\{ (x,ys) \mid \exists z . z `\text{elem}` (x:ys) 
    \quad \&\& z `\text{notElem}`
        \quad [ \text{insert } x \text{ ys } ]
\}
Meta-operations again

\{ x \mid x @ [ \text{insert} \_\_\_ ] \\
\quad \&\& [ \text{sort} \_\_ ] \ `\text{parent}` \_ x \\
\}

Implementation

• must take values from trace and convert to Haskell values at query time

• Yhc has a reflection API, allowing dynamic type inspection

• Hat trace contains types of constructors

• so Yhc can check that function application at query time is well-typed.
Summary of progress

- Query language has a design, based on extracting nodes from the trace, and either navigating, or testing assertions
- Parser exists
- Compilation rules in flux
- Difficulties lie in finding a minimal traversal of the trace