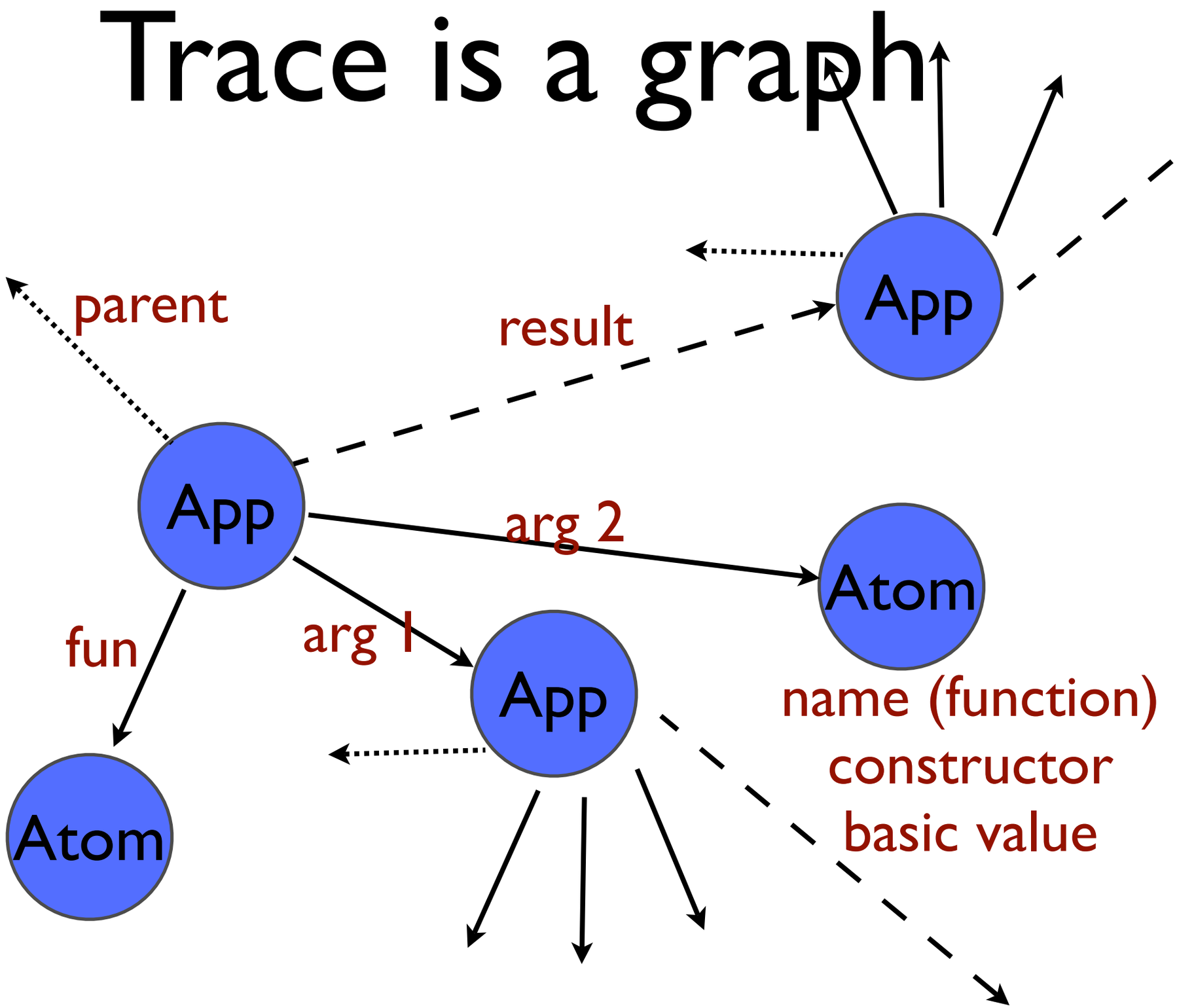
W-Hat?

a query language for Hat

PhD work by Tom Shackell

Trace is a graph



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 | name (fun n) == “insert”
    && numArgs n == 2 }
```

```
{ n | intValue (result^ n) == 7 }
```

Compiling queries

```
{ n | name (fun n) == “insert”
      && numArgs n == 2 }
```

```
[ n | n <- trace,
    , isJust (do f <- fun n
                 m <- name f
                 guard (m==“insert”)
                 a <- numArgs n
                 return (a==2)
             )
    ]
```

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) | match n of
            ⟦ insert 1 (3:y:_) ⟧
        }
```

```
n = insert 1 [3,4]
y = 4

n = insert 1 (3:⊥:5:[])
y = ⊥
```

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

```
{ xs | length ⟦ sort xs ⟧
       /= length xs
     }
```

```
xs = [1,2]
```

```
{ (x,ys) | length ⟦ insert x ys ⟧
           /= length ys + 1
         }
```

```
x = 1, ys = [2]
```

Alternatively...

```
{ (x,ys) | ∃z . z `elem` (x:ys)
           && z `notElem`
              ⟦ insert x ys ⟧
}
```

Mixes two time frames for computation:
trace time and query time.

Meta-operations again

```
{ x | x @ [[ insert _ _ ]]
      && [[ sort _ ]] `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