Lightweight Computation Tree Tracing for Lazy Functional Languages

Maarten Faddegon & Olaf Chitil

University of Kent | Computing

PLDI 2016

Shapiro’s algorithmic debugging method:

- ✓ Locates defect in code by asking judgements from oracle
- ✓ Is particularly suitable for pure computations
- ✓ Works from a computation tree

This presentation is about lightweight construction of computation tree

```
isOdd n = isEven (plusOne n)
isEven n = modTwo n == 0
plusOne n = n + 1
modTwo n = div n 2

prop_notBothOdd :: Int -> Bool
prop_notBothOdd x =
  isOdd x /= isOdd (x+1)
isOdd n = isEven (plusOne n)
isEven n = modTwo n == 0
plusOne n = n + 1
modTwo n = div n 2

prop_notBothOdd :: Int -> Bool
prop_notBothOdd x =
  isOdd x /= isOdd (x+1)

> quickCheck prop_notBothOdd
*** Failed! Falsifiable: 2
```

Example algorithmic debugging session:

```
isOdd 2 = False ?
```

Example algorithmic debugging session:

```
isOdd 2 = False ? right
isOdd 3 = False ?
```

Example algorithmic debugging session:

```
isOdd 2 = False ? right
isOdd 3 = False ? wrong
plusOne 3 = 4 ?
```

Example algorithmic debugging session:

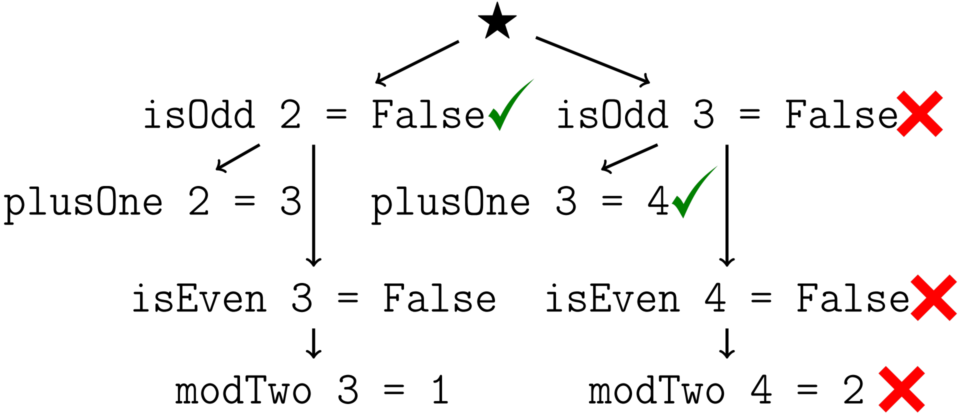
```
isOdd 2 = False ? right
isOdd 3 = False ? wrong
plusOne 3 = 4 ? right
isEven 4 = False ?
```

Example algorithmic debugging session:

```
isOdd 2 = False ? right
isOdd 3 = False ? wrong
plusOne 3 = 4 ? right
isEven 4 = False ? wrong
modTwo 4 = 2 ?
```

Example algorithmic debugging session:

```
isOdd 2 = False ? right
isOdd 3 = False ? wrong
plusOne 3 = 4 ? right
isEven 4 = False ? wrong
modTwo 4 = 2 ? wrong
Defect located in modTwo!
```


★
isOdd 2 = False ✓
isOdd 3 = False ✗
plusOne 2 = 3
plusOne 3 = 4 ✓
isEven 3 = False
isEven 4 = False ✗
modTwo 3 = 1
modTwo 4 = 2 ✗

What makes tracing and debugging of lazily evaluated programs hard?

```
isEven (plusOne 3)
```

```
isEven n = modTwo n == 0
plusOne n = n + 1
modTwo n = div n 2
```

```
isEven (plusOne 3)
 ⇓  modTwo n == 0 where n = plusOne 3
```

```
isEven n = modTwo n == 0
plusOne n = n + 1
modTwo n = div n 2
```

```
isEven (plusOne 3)
 ⇓  modTwo n == 0 where n = plusOne 3
  modTwo n
```

```
isEven n = modTwo n == 0
plusOne n = n + 1
modTwo n = div n 2
```

```
isEven (plusOne 3)
 ⇓   modTwo n == 0 where n = plusOne 3
  modTwo n
   ⇓   div n 2       where n = plusOne 3
```

```
isEven n = modTwo n == 0
plusOne n = n + 1
modTwo n = div n 2
```

```
isEven (plusOne 3)
  ⇓   modTwo n == 0  where n = plusOne 3
   modTwo n
     ⇓   div n 2       where n = plusOne 3
      ↯ plusOne 3
```

```
isEven n = modTwo n == 0
plusOne n = n + 1
modTwo n = div n 2
```

```
isEven (plusOne 3)
 ⇓   modTwo n == 0 where n = plusOne 3
  modTwo n
   ⇓   div n 2     where n = plusOne 3
    ↯ plusOne 3
       ⇓  3 + 1
```

```
isEven n = modTwo n == 0
plusOne n = n + 1
modTwo n = div n 2
```

```
isEven (plusOne 3)
 ⇓   modTwo n == 0 where n = plusOne 3
  modTwo n
   ⇓   div n 2     where n = plusOne 3
    ↯ plusOne 3
       ⇓   3 + 1
       ⇓   4
    ↯
```

```
isEven n = modTwo n == 0
plusOne n = n + 1
modTwo n = div n 2
```

```
isEven (plusOne 3)
  ⇓  modTwo n == 0 where n = plusOne 3
   modTwo n
    ⇓  div n 2        where n = plusOne 3
     ↯ plusOne 3
         ⇓  3 + 1
         ⇓  4
     ↯
    ⇓  div 4 2
```

```
isEven n = modTwo n == 0
plusOne n = n + 1
modTwo n = div n 2
```

```
isEven (plusOne 3)
 ⇓   modTwo n == 0 where n = plusOne 3
  modTwo n
   ⇓   div n 2      where n = plusOne 3
    ↯ plusOne 3
       ⇓   3 + 1
       ⇓   4
    ↯
   ⇓   div 4 2
   ⇓   2
```

```
isEven n = modTwo n == 0
plusOne n = n + 1
modTwo n = div n 2
```

```
isEven (plusOne 3)
  ⇓  modTwo n == 0 where n = plusOne 3
   modTwo n
     ⇓  div n 2      where n = plusOne 3
      ↯ plusOne 3
          ⇓  3 + 1
          ⇓  4
      ↯
     ⇓  div 4 2
     ⇓  2
  ⇓  2 == 0
  ⇓  False
```

```
isEven n = modTwo n == 0
plusOne n = n + 1
modTwo n = div n 2
```

Debugging by value observation

- ✓ Gill's Haskell Object Observation Debugger is a small library
- ✓ Used to to reveal intermediate values (`printf`-style debugging)
- ✓ Only suspected code is annotated
- ✓ Does not change order of evaluation

```
isOdd = observe "isOdd" isOdd’
isOdd’ n = isEven (plusOne n)
isEven = observe "isEven" isEven’
isEven’ n = mod2 n == 0
plusOne = observe "plusOne" plusOne’
plusOne’ n = n + 1
mod2 = observe "mod2" mod2’
mod2’ n = div n 2
```

1:	req.	result of `isEven`	
2:	req.	result of `mod2`	
3:	req.	argument of `mod2`	
4:	req.	argument of `isEven`	
5:	req.	result of `plusOne`	
6:	req.	argument of `plusOne`	
7:	resp.	argument of `plusOne`	is `3`
8:	resp.	result of `plusOne`	is `4`
9:	resp.	argument of `isEven`	is `4`
10:	resp.	argument of `mod2`	is `4`
11:	resp.	result of `mod2`	is `2`
12:	resp.	result of `isEven`	is `False`

1:	req.	result of `isEven`	
2:	req.	result of `mod2`	
3:	req.	argument of `mod2`	
4:	req.	argument of `isEven`	
5:	req.	result of `plusOne`	
6:	req.	argument of `plusOne`	
7:	resp.	argument of `plusOne`	is `3`
8:	resp.	result of `plusOne`	is `4`
9:	resp.	argument of `isEven`	is `4`
10:	resp.	argument of `mod2`	is `4`
11:	resp.	result of `mod2`	is `2`
12:	resp.	result of `isEven`	is `False`

$$\underbrace{\begin{array}{rlll} 1\text{:} & \text{req.} & \text{result of } \texttt{isEven} & \\ 4\text{:} & \text{req.} & \text{argument of } \texttt{isEven} & \\ 9\text{:} & \text{resp.} & \text{argument of } \texttt{isEven} & \text{is } \texttt{4} \\ 12\text{:} & \text{resp.} & \text{result of } \texttt{isEven} & \text{is } \texttt{False} \end{array}}_{\texttt{isEven 4 = False}}$$

```
isEven 4 = False ✗
isEven 3 = False ✓
modTwo 4 = 2 ✗
modTwo 3 = 1 ✓
plusOne 2 = 3 ✓
plusOne 3 = 4 ✓
```

`isEven 4 = False` ✗

`isEven 3 = False` ✓

`modTwo 4 = 2` ✗

`modTwo 3 = 1` ✓

`plusOne 2 = 3` ✓

`plusOne 3 = 4` ✓

No conclusion!

- ✓ Order of evaluation is unchanged
- ✓ User not exposed to messy and confusing output
- × Cannot directly use for algorithmic debugging because relation between computation statements unknown

Key insights:

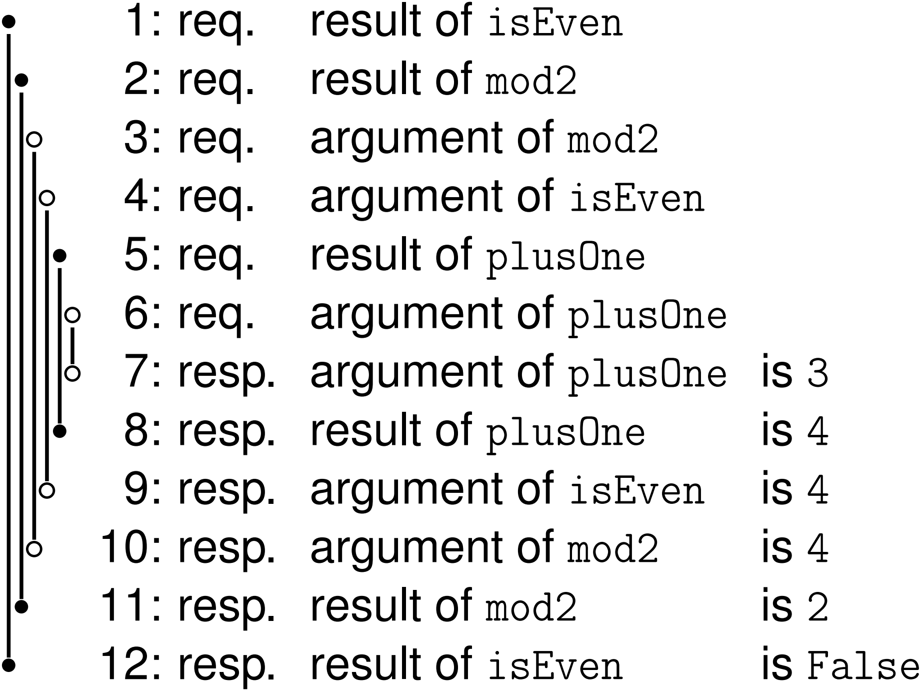
1. Value observation trace contains request and response events

Key insights:

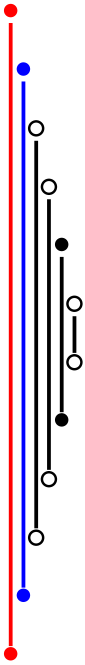
1. Value observation trace contains request and response events
2. Every request has corresponding response forming a span

Key insights:

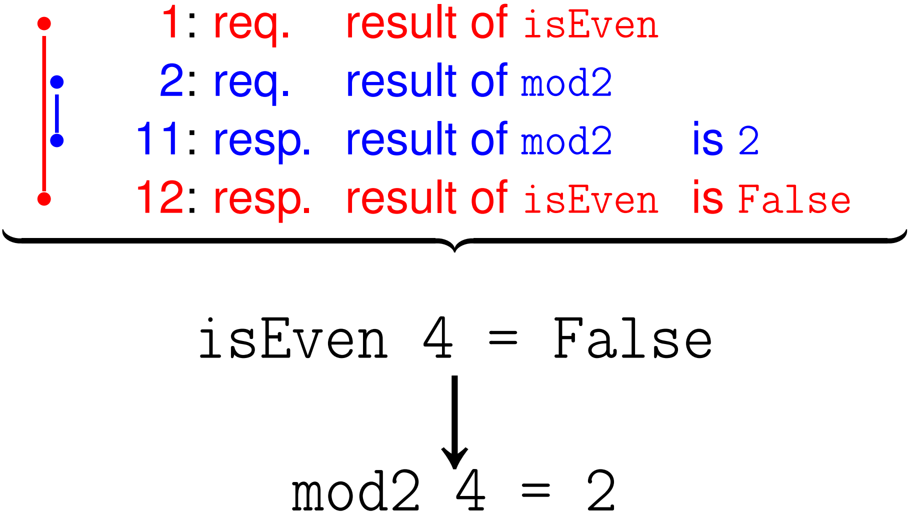
1. Value observation trace contains request and response events
2. Every request has corresponding response forming a span
3. Relation between statements derivable from nesting of spans



1:	req.	result of `isEven`	
2:	req.	result of `mod2`	
3:	req.	argument of `mod2`	
4:	req.	argument of `isEven`	
5:	req.	result of `plusOne`	
6:	req.	argument of `plusOne`	
7:	resp.	argument of `plusOne`	is `3`
8:	resp.	result of `plusOne`	is `4`
9:	resp.	argument of `isEven`	is `4`
10:	resp.	argument of `mod2`	is `4`
11:	resp.	result of `mod2`	is `2`
12:	resp.	result of `isEven`	is `False`



1:	req.	result of `isEven`	
2:	req.	result of `mod2`	
3:	req.	argument of `mod2`	
4:	req.	argument of `isEven`	
5:	req.	result of `plusOne`	
6:	req.	argument of `plusOne`	
7:	resp.	argument of `plusOne`	is `3`
8:	resp.	result of `plusOne`	is `4`
9:	resp.	argument of `isEven`	is `4`
10:	resp.	argument of `mod2`	is `4`
11:	resp.	result of `mod2`	is `2`
12:	resp.	result of `isEven`	is `False`



1: req. result of `isEven`

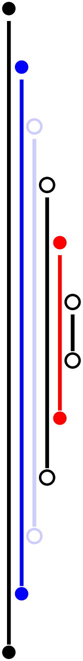
2: req. result of `mod2`

11: resp. result of `mod2` is `2`

12: resp. result of `isEven` is `False`

`isEven 4 = False`

`mod2 4 = 2`



1:	req.	result of `isEven`	
2:	req.	result of `mod2`	
3:	req.	argument of `mod2`	
4:	req.	argument of `isEven`	
5:	req.	result of `plusOne`	
6:	req.	argument of `plusOne`	
7:	resp.	argument of `plusOne`	is 3
8:	resp.	result of `plusOne`	is 4
9:	resp.	argument of `isEven`	is 4
10:	resp.	argument of `mod2`	is 4
11:	resp.	result of `mod2`	is 2
12:	resp.	result of `isEven`	is `False`

Tracing with Hoed Pure

PLDI * Artifact * Evaluated *

Consistent * Complete * Well Documented * Easy t

Hoed constructed computation trees for

- ✓ The pretty printing library FPretty
- ✓ The window manager XMonad
- ✓ The video game Raincat

Installation instructions and further reading:

`https://wiki.haskell.org/Hoed`

Practical Computation Tree Tracing

- ✓ Simple implementation that is easy to maintain
- ✓ Works with any run-time system
- ✓ Exact computation trees from request-response pairs
- ✓ Annotations in suspected code only: applicable to wide range of programs
- ✓ Handles higher-order functions and data constructors