

# Debugging Lazy Functional Programs by Asking the Oracle

Bernd Braßel, Holger Siegel

Christian-Albrechts-University of Kiel

Freiburg, Germany September 29th, 2007

# Debugging for Lazy Languages

The most basic approach to debugging

Look at program execution step by step.

But:

Lazy step-by-step is often *difficult to comprehend*.

Approach works much better for *strict* languages.

## A program with a bug

```
notSoQuickSort (x:xs) = smalls ++ x:larges
  where smalls = notSoQuickSort (filter (<=x) xs)
        larges = notSoQuickSort (filter (>=x) xs)
```

### Lazy steps

```
notSoQuickSort [3,7,4,2,7]
... unfold notSoQuickSort
  filter (<=3) [3,7,4,2,7]
  3:filter (<=3) [7,4,2,7]
... calls to other functions
  filter (<=3) [2,7]
  2:filter (<=3) [7]
... calls to other functions
[2,3,4,7,7,7]
```

Sub computations are **shallow**, many **context switches**.

## A program with a bug

```
notSoQuickSort (x:xs) = smalls ++ x:larges
  where smalls = notSoQuickSort (filter (<=x) xs)
        larges = notSoQuickSort (filter (>=x) xs)
```

### Strict Steps

```
notSoQuickSort [3,7,4,2,7]
... unfold notSoQuickSort
  filter (<=3) [3,7,4,2,7]
  ... full evaluation of filter
    [3,2]
  ... calls to other functions
    [2,3,4,7,7,7]
```

Sub computations are **deep**, context switches **only at the end**.

Many sophisticated debugging techniques make use of such a strict sub structure, e.g., declarative debugging.

## How do tools like Hat manage to display strict information?

They record detailed information about all reductions made during program execution.

For the previous example about 8kb in Hat.

## Oracles are small

**Oracles** are a way to store much less data still enabling sophisticated debugging techniques in an efficient way.

This is “the Oracle” that our approach stores for the example program:

## How do tools like Hat manage to display strict information?

They record detailed information about all reductions made during program execution.

For the previous example about 8kb in Hat.

## Oracles are small

**Oracles** are a way to store much less data still enabling sophisticated debugging techniques in an efficient way.

This is “the Oracle” that our approach stores for the example program:

[37]

# What the Numbers Mean

## The Oracle has all the information about laziness

- Reexecute program with **strict** semantics
- Oracle tells which expressions to discard, i.e., **not** to evaluate
- Replace discarded expression by special value **underscore**
- The Oracle: number of steps between discards

## Example with many discards

```
fibs n = fib n : fibs (n+1)
main = length (take 2 (fibs 0))
```

The recorded Oracle is: [2,0,1,0,0,23]

# What the Numbers Mean

## The Oracle has all the information about laziness

- Reexecute program with **strict** semantics
- Oracle tells which expressions to discard, i.e., **not** to evaluate
- Replace discarded expression by special value **underscore**
- The Oracle: number of steps between discards

```
main                                     [2,0,1,0,0,23]
length (take 2 (fibs 0))                 [1,0,1,0,0,23]
```

## Example with many discards

```
fibs n = fib n : fibs (n+1)
main = length (take 2 (fibs 0))
```

The recorded Oracle is: [2,0,1,0,0,23]

# What the Numbers Mean

## The Oracle has all the information about laziness

- Reexecute program with **strict** semantics
- Oracle tells which expressions to discard, i.e., **not** to evaluate
- Replace discarded expression by special value underscore
- The Oracle: number of steps between discards

```
length (take 2 (fibs 0))           [1,0,1,0,0,23]
length (take 2 (fib 0 : fibs (2+1))) [0,0,1,0,0,23]
```

## Example with many discards

```
fibs n = fib n : fibs (n+1)
main = length (take 2 (fibs 0))
```

The recorded Oracle is: [2,0,1,0,0,23]

# What the Numbers Mean

## The Oracle has all the information about laziness

- Reexecute program with **strict** semantics
- Oracle tells which expressions to discard, i.e., **not** to evaluate
- Replace discarded expression by special value underscore
- The Oracle: number of steps between discards

```
length (take 2 (fib 0 : fibs (2+1)))      [0,0,1,0,0,23]
length (take 2 (_      : fibs (2+1)))    [0,1,0,0,23]
```

## Example with many discards

```
fibs n = fib n : fibs (n+1)
main = length (take 2 (fibs 0))
```

The recorded Oracle is: [2,0,1,0,0,23]

# What the Numbers Mean

## The Oracle has all the information about laziness

- Reexecute program with **strict** semantics
- Oracle tells which expressions to discard, i.e., **not** to evaluate
- Replace discarded expression by special value underscore
- The Oracle: number of steps between discards

```
length (take 2 ( _: fibs (2+1) ))           [0,1,0,0,23]
length (take 2 ( _: fibs _      ))         [1,0,0,0,23]
```

## Example with many discards

```
fib n = fib n : fibs (n+1)
main = length (take 2 (fibs 0))
```

The recorded Oracle is: [2,0,1,0,0,23]

# What the Numbers Mean

## The Oracle has all the information about laziness

- Reexecute program with **strict** semantics
- Oracle tells which expressions to discard, i.e., **not** to evaluate
- Replace discarded expression by special value underscore
- The Oracle: number of steps between discards

```
length (take 2 ( _:fibs _))           [1,0,0,23]
length (take 2 ( _:fib _:fibs (_+1))) [0,0,0,23]
```

## Example with many discards

```
fibs n = fib n : fibs (n+1)
main = length (take 2 (fibs 0))
```

The recorded Oracle is: [2,0,1,0,0,23]

# What the Numbers Mean

## The Oracle has all the information about laziness

- Reexecute program with **strict** semantics
- Oracle tells which expressions to discard, i.e., **not** to evaluate
- Replace discarded expression by special value underscore
- The Oracle: number of steps between discards

```
length (take 2 ( _:fib _:fibs (_+1)))      [0,0,0,23]
length (take 2 ( _: _      :fibs (_+1)))    [0,0,23]
```

## Example with many discards

```
fibs n = fib n : fibs (n+1)
main = length (take 2 (fibs 0))
```

The recorded Oracle is: [2,0,1,0,0,23]

# What the Numbers Mean

## The Oracle has all the information about laziness

- Reexecute program with **strict** semantics
- Oracle tells which expressions to discard, i.e., **not** to evaluate
- Replace discarded expression by special value underscore
- The Oracle: number of steps between discards

```
length (take 2 (::_: fibs (+1)))      [0,0,23]
length (take 2 (::_: fibs _))        [0,23]
```

## Example with many discards

```
fib n = fib n : fibs (n+1)
main = length (take 2 (fibs 0))
```

The recorded Oracle is: [2,0,1,0,0,23]

# What the Numbers Mean

## The Oracle has all the information about laziness

- Reexecute program with **strict** semantics
- Oracle tells which expressions to discard, i.e., **not** to evaluate
- Replace discarded expression by special value underscore
- The Oracle: number of steps between discards

```
length (take 2 ( _:_:fibs _))           [0,23]
length (take 2 ( _:_:_   ))           [23]
```

## Example with many discards

```
fibs n = fib n : fibs (n+1)
main = length (take 2 (fibs 0))
```

The recorded Oracle is: [2,0,1,0,0,23]

# What the Numbers Mean

## The Oracle has all the information about laziness

- Reexecute program with **strict** semantics
- Oracle tells which expressions to discard, i.e., **not** to evaluate
- Replace discarded expression by special value underscore
- The Oracle: number of steps between discards

```
length (take 2 (_:_:_))           [23]
length [_,_]                       [6]
```

## Example with many discards

```
fibs n = fib n : fibs (n+1)
main = length (take 2 (fibs 0))
```

The recorded Oracle is: [2,0,1,0,0,23]

# What the Numbers Mean

## The Oracle has all the information about laziness

- Reexecute program with **strict** semantics
- Oracle tells which expressions to discard, i.e., **not** to evaluate
- Replace discarded expression by special value underscore
- The Oracle: number of steps between discards

```
length [_,_]           [6]
2                       [0]
```

## Example with many discards

```
fibs n = fib n : fibs (n+1)
main = length (take 2 (fibs 0))
```

The recorded Oracle is: [2,0,1,0,0,23]

# Trading Time for Space: Why is it good?

## Main Disadvantage

**Slower data access** for such data that has been recorded explicitly

Future Work: Try to move access time to user interaction time.

## Advantages

- Less Disk Space** obvious
- Faster Recording** because less data is written
- Simple** easy to implement, easy to extend.  
Example: Virtual I/O
- Flexible** Future work: collect different kinds of data by instantiating “event hooks” of interpreter monad

# Virtual I/O

Since the program is reexecuted, we can execute actions at the right times.

```
module IOExample where
```

```
import Prelude hiding (getLine)
```

```
getLine :: IO String
```

```
getLine = getChar >>= testEOL
```

```
testEOL :: Char -> IO String
```

```
testEOL c = if c=='\n'
```

```
    then return []
```

```
    else getLine >>= \ cs -> return (c:cs)
```

```
main = getLine >>= writeFile "userInput"
```

## Three aspects of implementation

- 1 How to obtain the oracle
- 2 How programs are compiled for interpreter
- 3 How the lazy call-by-value interpreter works

## 1) Oracle production at ICFP

- How to instrument a lazy machine to get oracle numbers
- Proof of soundness:  
call-by-value interpreter computes the same results

## 2) Compilation for strict interpretation

You already know what these things look like.

# The Lazy Call-By-Value Interpreter

## The debug monad: a combination of three monads

**State** to manage oracle information, debugger options, socket to virtual I/O gui, etc.

**Error** to terminate debugger when bug is identified, resp. user quitted

**IO** for user interaction

```
type DebugMonad a =  
  StateT DebuggerState (ErrorT (Maybe BugReport) IO) a
```

# The Lazy Call-By-Value Interpreter

## The debug monad: a combination of three monads

**State** to manage oracle information, debugger options, socket to virtual I/O gui, etc.

**Error** to terminate debugger when bug is identified, resp. user quitted

**IO** for user interaction

```
type DebugMonad a =  
  StateT DebuggerState (ErrorT (Maybe BugReport) IO) a
```

# The Lazy Call-By-Value Interpreter

## The debug monad: a combination of three monads

**State** to manage oracle information, debugger options, socket to virtual I/O gui, etc.

**Error** to terminate debugger when bug is identified, resp. user quitted

**IO** for user interaction

```
type DebugMonad a =  
  StateT DebuggerState (ErrorT (Maybe BugReport) IO) a
```

# The Lazy Call-By-Value Interpreter

## The debug monad: a combination of three monads

**State** to manage oracle information, debugger options, socket to virtual I/O gui, etc.

**Error** to terminate debugger when bug is identified, resp. user quitted

**IO** for user interaction

```
type DebugMonad a =  
  StateT DebuggerState (ErrorT (Maybe BugReport) IO) a
```

## Basic function eval

Decide whether given action is evaluated or underscore is returned.

```
eval :: DebugMonad a -> DebugMonad a
```

## The value underscore

### The Problem:

- non evaluated expressions are replaced by underscore
- Extend data declarations with new constructor?
  - Need to convert types at run time for external functions
  - Keep two versions of types around

## The value underscore

### The Problem:

- non evaluated expressions are replaced by underscore
- Extend data declarations with new constructor?
  - Need to convert types at run time for external functions
  - Keep two versions of types around

### Basic Facts:

- underscore is only needed to display debug output for user
- output is part of I/O monad

## The value underscore

### The Problem:

- non evaluated expressions are replaced by underscore
- Extend data declarations with new constructor?
  - Need to convert types at run time for external functions
  - Keep two versions of types around

### Basic Facts:

- underscore is only needed to display debug output for user
- output is part of I/O monad

Clever idea (by Frank Huch): Exploit exceptions.

```
underscore :: a
```

```
underscore = throw NonTermination
```

## Type class ShowTerm with generated instances

Create term representation for flexible output

```
data Term = Term String [Term] | Underscore | Fail String
```

```
class ShowTerm a where
```

```
  showCons :: a -> DebugMonad Term
```

```
showTerm :: ShowTerm a => a -> DebugMonad Term
```

```
showTerm x = liftIO
```

```
  (catch (x 'seq' return Nothing) (return . Just)) >>=
```

```
  maybe (showCons x)
```

```
    (\ e -> case e of
```

```
      NonTermination -> return Underscore
```

```
      ErrorCall s     -> return (Fail s))
```

## Type class ShowTerm with generated instances

Create term representation for flexible output

```
data Term = Term String [Term] | Underscore | Fail String
```

```
class ShowTerm a where
```

```
  showCons :: a -> DebugMonad Term
```

```
instance ShowTerm a => ShowTerm [a] where
```

```
  showCons []          = return (Term "[]" [])
```

```
  showCons (x1:x2) = do sx1 <- showTerm x1
```

```
                        sx2 <- showTerm x2
```

```
                        return (Term ":" [sx1,sx2])
```

# Adding Declarative Debugging

Conceptually, Oracle is a list of boolean values

- True means “unfold”, False means “discard”.

[2,0,1,0,0,23]

TFFFTFFFTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT

# Adding Declarative Debugging

Conceptually, Oracle is a list of boolean values

- True means “unfold”, False means “discard”.
- there is a current position

[2,0,1,0,0,23]

TFFTFFFT→TTTTTTTTTTTTTTTTTTTTTTTT

# Adding Declarative Debugging

Conceptually, Oracle is a list of boolean values

- True means “unfold”, False means “discard”.
- there is a current position
- **Skip**: move current position to end of next sub computation

[2,0,1,0,0,23]

TTFFTFFFT→TTTTTTTTTTTTTTTTTTTTTTTTTTT

TTFFTFFFTTTTTT→TTTTTTTTTTTTTTTTTTTT

# Adding Declarative Debugging

Conceptually, Oracle is a list of boolean values

- True means “unfold”, False means “discard”.
- there is a current position
- **Skip**: move current position to end of next sub computation
- **Correct**: mark from current position to end of sub computation

[2,0,1,0,0,23]

TTFFTFFFT → TTTTTTTTTTTTTTTTTTTTTTTT

TTFFTFFFTTTTTT → TTTTTTTTTTTTTTTTTTTT

TTFFTFFFTTTTTT [TTTTTTTTTT] → TTTTTTTTTT

# Adding Declarative Debugging

Conceptually, Oracle is a list of boolean values

- True means “unfold”, False means “discard”.
- there is a current position
- **Skip**: move current position to end of next sub computation
- **Correct**: mark from current position to end of sub computation
- **Wrong**: cut chase to next sub computation

```
[2,0,1,0,0,23]
TTFFTFFFT→TTTTTTTTTTTTTTTTTTTTTTTTTTT
TTFFTFFFTTTTTT→TTTTTTTTTTTTTTTTTTTTTTT
TTFFTFFFTTTTTT [TTTTTTTTTT] →TTTTTTTTT
→TTTTTTTT
```

# Adding Declarative Debugging

Conceptually, Oracle is a list of boolean values

- True means “unfold”, False means “discard”.
- there is a current position
- **Skip**: move current position to end of next sub computation
- **Correct**: mark from current position to end of sub computation
- **Wrong**: cut chase to next sub computation
- **End**: whole remaining area marked correct

```
[2,0,1,0,0,23]
TTFFTFFFT→TTTTTTTTTTTTTTTTTTTTTTTTTTT
TTFFTFFFTTTTTT→TTTTTTTTTTTTTTTTTTTTTTT
TTFFTFFFTTTTTT [TTTTTTTTTT] →TTTTTTTTT
→TTTTTTTT
[TTTTTT]
```

## Debugging by Asking the Oracle

- program is transformed to two versions
- during execution of first version Oracle is recorded
- Oracle: list of numbers, the strict steps between discarding expressions
- the second version is a monadic left-most innermost program consuming the oracle
- execute second version for debugging

## Future Work

- adding flexibility by introducing event hooks
- shifting execution time to background during user interaction
- integrate more debugging views