Foundations for the Debugging of Functional Programs

Olaf Chitil, Yong Luo and Thomas Davie

University of Kent, UK
Supported by EPSRC grant EP/C516605/1

12th February 2008
Programs have Bugs

Even functional programs!

strong type system $\implies$ cannot corrupt run-time system

but

- wrong result
- abortion with run-time error
- non-termination
Why Debug Functional Programs Differently?

- No canonical execution model.
  - various reduction semantics (small step, big step)
  - interpreters with environments (explicit substitutions)
  - also denotational semantics

- No sequential execution of statements.
  - evaluation of expressions
  - evaluation of subexpressions is independent
    \[ f (g \ 3 \ 4) \ (h \ 1 \ 2) \ (i \ 5) \ (j \ 3 \ 9 \ 3) \]
Why Debug Functional Programs Differently?

- No canonical execution model.
  - various reduction semantics (small step, big step)
  - interpreters with environments (explicit substitutions)
  - also denotational semantics

- No sequential execution of statements.
  - evaluation of expressions
  - evaluation of subexpressions is independent

\[
f (g\ 3\ 4)\ (h\ 1\ 2)\ (i\ 5)\ (j\ 3\ 9\ 3)
\]

Conclusions

- Abstract from execution details: views for various semantics models.
- Take advantage of simple and compositional semantics.
- Liberate from sequentiality of computation.
Two-Phase Tracing: The trace as data structure.
- Liberates from the time arrow of computation.
- Enables many different views.

But where are formal definitions you can reason with?
Example: Insertion Sort

main :: String
main = sort "sort"

sort :: Ord a => [a] -> [a]
sort [] = []
sort (x:xs) = insert x (sort xs)

insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) = if x > y then y : (insert x ys) else x:ys

There is a bug: main = "os"!
Start with expression \texttt{sort (’t’:[])}
sort [] = []

sort (x:xs) = insert x (sort xs)

insert x [] = [x]

insert x (y:ys) = if x > y then y:(insert x ys) else x:ys
sort [] = []

sort (x:xs) = insert x (sort xs)

insert x [] = [x]

insert x (y:ys) = if x > y then y:(insert x ys) else x:ys
sort [] = []
sort (x:xs) = insert x (sort xs)

insert x [] = [x]
insert x (y:ys) = if x > y then y:(insert x ys) else x:ys
New nodes for right-hand-side, connected via result pointer.
Only add to graph, never remove.
Sharing ensures compact representation.
The Node Naming Scheme

Aim

- not distinguish isomorphic graphs
- avoid inconvenience of isomorphism classes
The Node Naming Scheme

Aim
- not distinguish isomorphic graphs
- avoid inconvenience of isomorphism classes

Solution
- standard representation with node describing path from root
- path at creation time (sharing later)
- path independent of evaluation order
The Node Labels

\[
\begin{align*}
n & \ ::= \ \{f, a, r\}^* \\
T & \ ::= \ a \quad \text{atom} \\
& \quad | \ nm \quad \text{application of nodes} \\
a & \ ::= \ f \ | \ C \ | \ 42 \ | \ldots \quad \text{defined variable, data constructor} \\
& \quad | \ \text{atomic literal, \ldots}
\end{align*}
\]

Reduction edge implicitly given through existence of node.

Olaf Chitil (Kent, UK)  Foundations for Debugging  12th February 2008  10 / 32
Projections

- Reduction edge implicitly given through existence of node.
- Every redex should be parent of at least one node.
  (otherwise reduction unreachable from computation result)

True && x = x
not True = False
Projections

- Reduction edge implicitly given through existence of node.
- Every redex should be parent of at least one node. (otherwise reduction unreachable from computation result)

⇒ A projection requires an indirection as result.

```
True && x = x
not True = False
```

```
label term  T  ::=  a                          atom
    |  n m                                      application of nodes
    |  n                                          indirection

atom  a  ::=  x | C | 42 | ... variable, data constructor, ...
```
Projections

- Reduction edge implicitly given through existence of node.
- Every redex should be parent of at least one node.
  (otherwise reduction unreachable from computation result)

⇒ A projection requires an **indirection** as result.

\[
\begin{align*}
\text{True} & \land x = x \\
\text{not True} & = \text{False}
\end{align*}
\]

\[
\begin{array}{c}
\text{label term} \\
T :=
\end{array}
\begin{array}{c}
a \\
n m \\
n
\end{array}
\begin{array}{c}
\text{atom} \\
\text{application of nodes} \\
\text{indirection}
\end{array}
\]

\[
\begin{array}{c}
\text{atom} a :=
\end{array}
\begin{array}{c}
x \mid C \mid 42 \mid \ldots
\end{array}
\begin{array}{c}
\text{variable, data constructor,} \\
\ldots
\end{array}
\]
A trace $G$ for initial term $M$ and program $P$ is a partial function from nodes to term constructors, $G : n \mapsto T$, defined by

- The unshared graph representation of $M$, $\text{graph}_G(\varepsilon, M)$, is a trace.
- If $G$ is a trace and
  - $L = R$ an equation of the program $P$,
  - $\sigma$ a substitution replacing argument variables by nodes,
  - $\text{match}_G(n, L\sigma)$,
  - $nr \notin \text{dom}(G)$,

then $G \cup \text{graph}_G(nr, R\sigma)$ is a trace.

No evaluation order is fixed.
A node represents many terms, in particular a most evaluated one.

\[
\text{mef}_G(\varepsilon) = (:) \ 't' \ []
\]

**Definition**

\[
\begin{align*}
  n \succ_G m & \iff m = nr \lor G(n) = m \\
  [n]_G = m & \iff n \succ^*_G m \land \forall o. m \succ_G o
\end{align*}
\]

**Definition**

\[
\begin{align*}
  \text{mef}_G(n) & = \text{mefT}_G(G([n]_G)) \\
  \text{mefT}_G(a) & = a \\
  \text{mefT}_G(n) & = \text{mef}_G(n) \\
  \text{mefT}_G(n \cdot m) & = \text{mef}_G(n) \cdot \text{mef}_G(m)
\end{align*}
\]
Redexes and Big-Step Reductions

\[
\text{redex}_G(r) = \text{insert } 't' \; []
\]

\[
\text{bigstep}_G(r) = \text{insert } 't' \; [] = (:) 't' \; []
\]

Definition

For any redex node \( n \), i.e., \( nr \in \text{dom}(G) \),

\[
\text{redex}_G(n) = \begin{cases} 
\text{mef}_G(m) \; \text{mef}_G(o), & \text{if } G(n) = m \; o \\
\text{a}, & \text{if } G(n) = a 
\end{cases}
\]

\[
\text{bigstep}_G(n) = \text{redex}_G(n) = \text{mef}_G(n)
\]
Every redex node $n$ yields a tree node $n$ labelled $\text{bigstep}_G(n)$.

Tree node $n$ is child of tree node $\text{parent}(n)$.

\begin{align*}
\text{parent}(nr) &= n \\
\text{parent}(nf) &= \text{parent}(n) \\
\text{parent}(na) &= \text{parent}(n) \\
\text{parent}(\varepsilon) &= \text{undefined} \\
\end{align*}
Algorithmic Debugging with the Computation Tree

main = "os"

sort "sort" = "os"

sort "ort" = "o"

insert 's' "o" = "os"

's' > 'o' = True

insert 's' "" = "s"

sort "rt" = "r"

insert 'o' "r" = "o"

'0' > 'r' = False

sort "t" = "t"

insert 'r' "t" = "r"

'0' > 'r' = False

sort "" = ""

insert 't' "" = "t"

'0' > 't' = False

Olaf Chitil (Kent, UK)
Algorithmic Debugging with the Computation Tree

```
main = "os"  ×

sort "sort" = "os"

sort "ort" = "o"
insert 's' "o" = "os"
's' > 'o' = True
insert 's' "" = "s"

sort "rt" = "r"
insert 'o' "r" = "o"

sort "t" = "t"
insert 'r' "t" = "r"
'o' > 'r' = False

sort "" = ""
insert 't' "" = "t"
'r' > 't' = False
```
Algorithmic Debugging with the Computation Tree

main = "os"

sort "sort" = "os"

sort "ort" = "o"

insert 's' "o" = "os"

's' > 'o' = True

insert 's' "" = "s"

sort "rt" = "r"

insert 'o' "r" = "o"

'o' > 'r' = False

sort "t" = "t"

insert 'r' "t" = "r"

'r' > 't' = False

sort "" = ""

insert 't' "" = "t"

'o' > 'r' = False

sort "" = ""
Algorithmic Debugging with the Computation Tree

main = "os"  ×

sort "sort" = "os"  ×

sort "ort" = "o"

insert 's' "o" = "os"  √

's' > 'o' = True

insert 's' "" = "s"

sort "rt" = "r"

insert 'o' "r" = "o"

'o' > 'r' = False

sort "t" = "t"

insert 'r' "t" = "r"

'r' > 't' = False

sort "" = ""

insert 't' "" = "t"
Algorithmic Debugging with the Computation Tree

main = "os" ×

sort "sort" = "os" ×

sort "ort" = "o" ×

insert 's' "o" = "os" ✓

's' > 'o' = True

insert 's' "" = "s"

sort "rt" = "r"

insert 'o' "r" = "o"

sort "t" = "t"

insert 'r' "t" = "r"

'o' > 'r' = False

sort "" = ""

insert 't' "" = "t"

'r' > 't' = False

Olaf Chitil (Kent, UK)
Algorithmic Debugging with the Computation Tree

```
main = "os"  
  sort "sort" = "os"  
    sort "ort" = "o"  
      sort "rt" = "r"  
        sort "t" = "t"  
          sort "" = ""  
            insert 't' "" = "t"  
              'r' > 't' = False
        insert 'r' "t" = "r"  
          'o' > 'r' = False
      insert 'o' "r" = "o"  
        's' > 'o' = True
    insert 's' "o" = "os"  
      's' > 'o' = True

Olaf Chitil (Kent, UK)
Foundations for Debugging
12th February 2008 19 / 32
```
Algorithmic Debugging with the Computation Tree

main = "os"  

sort "sort" = "os"  

sort "ort" = "o"  
insert 's' "o" = "os"  
's' > 'o' = True  
insert 's' "" = "s"  
sort "rt" = "r"  
insert 'o' "r" = "o"  
're' > 'r' = False  

sort "t" = "t"  
insert 'r' "t" = "r"  
'o' > 'r' = False  

sort "" = ""  
insert 't' "" = "t"  
'r' > 't' = False  

Olaf Chitil (Kent, UK)  
Foundations for Debugging  
12th February 2008  

19 / 32
main :: String
main = sort "sort"

sort :: Ord a => [a] -> [a]
sort [] = []
sort (x:xs) = insert x (sort xs)

insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) = if x > y then y : (insert x ys) else x:ys

Faulty computation: insert 'o' "r" = "o"
Correctness of Algorithmic Debugging: The Property

If node $n$ incorrect and all its children correct, then node $n$ faulty, i.e., its equation is faulty.

\[
\varepsilon \text{ sort } (\text{'t':[]}) = \text{'t':[]} \\
\text{ra sort } [] = [] \\
\text{r insert 't' [] = 't':[]} \\
\]

**Definition**

Tree node $n$ incorrect $\iff$ redex$_G(n) \not\equiv_I$ mef$_G(n)$.
Tree node $n$ faulty $\iff$ redex$_G(n) \not\equiv_I$ reduct$_G(n)$.

If tree node $n$ faulty, then for its program equation $L = R$ exists substitution $\sigma$ such that $L\sigma \not\equiv_I R\sigma$. 
**Theorem**

Let \( n \) be a redex node. If for all redex nodes \( m \) with \( \text{parent}(m) = n \) we have \( \text{redex}_G(m) \equiv_I \text{mef}_G(m) \), then \( \text{reduct}_G(n) \equiv_I \text{mef}_G(n) \).

With \( \text{redex}_G(n) \not\equiv_I \text{mef}_G(n) \) follows \( \text{redex}_G(n) \not\equiv_I \text{reduct}_G(n) \).
Higher-Order Insertion Sort

main :: String
main = sort "sort"

sort :: Ord a => [a] -> [a]
sort = foldr insert []

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)

insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) = if x > y then y : (insert x ys) else x:ys
main = "os"

foldr insert [] "sort" = "os"
foldr insert [] "ort" = "o"

sort = foldr insert []
insert 's' "o" = "os"

's' > 'o' = True
insert 's' "" = "s"

foldr insert [] "rt" = "r"

foldr insert [] "t" = "t"

foldr insert [] "" = ""
insert 't' "" = "t"

insert 'o' "r" = "o"
insert 'r' "t" = "r"
'o' > 'r' = False
'r' > 't' = False
Higher-Order Algorithmic Debugging

```
main = "os"

foldr insert [] "sort" = "os"
sort = foldr insert []

foldr insert [] "ort" = "o"
insert 's' "o" = "os"

's' > 'o' = True

foldr insert [] "rt" = "r"
insert 'o' "r" = "o"

'o' > 'r' = False

foldr insert [] "t" = "t"
insert 'r' "t" = "r"

'r' > 't' = False

foldr insert [] "" = ""
insert 't' "" = "t"

'o' > 'r' = False
```

Olaf Chitil (Kent, UK)
Main function:

```
main = "os" ×
```

Fold function:

```
foldr insert [] "sort" = "os"
```

Sort function:

```
sort = foldr insert [] ✓
```

Insert function:

```
insert 's' "o" = "os"
```

Character comparisons:

- `'s' > 'o' = True`
- `'o' > 'r' = False`
- `'r' > 't' = False`

Foldr insert: empty list inputs:

```
foldr insert [] "ort" = "o"
foldr insert [] "rt" = "r"
foldr insert [] "t" = "t"
foldr insert [] "" = ""
```

Insert examples:

```
insert 's' "o" = "os"
insert 'o' "r" = "o"
insert 'r' "t" = "r"
insert 't' "" = "t"
```

Auditing:

```
'ō' > 'r' = False
```

Foundations for Debugging

12th February 2008 27 / 32
Higher-Order Algorithmic Debugging

main = "os" ×

foldr insert [] "sort" = "os" ×

foldr insert [] "ort" = "o"

foldr insert [] "rt" = "r"

foldr insert [] "t" = "t"

foldr insert [] "" = ""

sort = foldr insert [] √

insert 's' "o" = "os"

insert 's' "" = "s"

'.' > 'o' = True

insert 'o' "r" = "o"

insert 'r' "t" = "r"

'.' > 'r' = False

insert 't' "" = "t"

'.' > 't' = False

Olaf Chitil (Kent, UK) Foundations for Debugging 12th February 2008
Higher-Order Algorithmic Debugging

main = "os" ×

foldr insert [] "sort" = "os" ×
sort = foldr insert [] ✓

foldr insert [] "ort" = "o"

insert 's' "o" = "os" ✓

's' > 'o' = True

foldr insert [] "rt" = "r"

insert 'o' "r" = "o"

foldr insert [] "t" = "t"

insert 'r' "t" = "r"

'o' > 'r' = False

foldr insert [] "" = ""

insert 't' "" = "t"

'r' > 't' = False

Olaf Chitil (Kent, UK)  Foundations for Debugging  12th February 2008  27 / 32
Higher-Order Algorithmic Debugging

```
main = "os" ×
foldr insert [] "sort" = "os" ×
sort = foldr insert [] √
foldr insert [] "ort" = "o" ×
insert 's' "o" = "os" √
's' > 'o' = True
insert 's' "" = "s"
foldr insert [] "rt" = "r"
insert 'o' "r" = "o"
foldr insert [] "t" = "t"
insert 'r' "t" = "r"
'o' > 'r' = False
foldr insert [] "" = ""
insert 't' "" = "t"
'r' > 't' = False
```

Olaf Chitil (Kent, UK)  Foundations for Debugging  12th February 2008
Higher-Order Algorithmic Debugging

main = "os" ×

foldr insert [] "sort" = "os" ×

foldr insert [] "ort" = "o" ×

foldr insert [] "rt" = "r"

foldr insert [] "t" = "t"

foldr insert [] "" = ""

sort = foldr insert [] √

insert 's' "o" = "os" √

insert 's' "" = "s"

' s' > ' o' = True

insert 'o' "r" = "o" ×

insert 'r' "t" = "r"

insert 'r' "" = "t"

'o' > 'r' = False

'r' > 't' = False
Higher-Order Algorithmic Debugging

\[
\text{main} = "\text{os}" \times
\]

\[
\text{foldr \ insert} \ [\ ] "\text{sort}" = "\text{os}" \times
\]

\[
\text{sort} = \text{foldr \ insert} \ [\ ] \checkmark
\]

\[
\text{foldr \ insert} \ [\ ] "\text{ort}" = "\text{o}" \times
\]

\[
\text{insert \ 's' \ 'o'}" = "\text{os}" \checkmark
\]

\[
\text{'s'} > \text{'o'} = \text{True}
\]

\[
\text{insert \ 's'} \ "" = "\text{s}"
\]

\[
\text{foldr \ insert} \ [\ ] "\text{rt}" = "\text{r}"
\]

\[
\text{insert \ 'o'} \ "\text{r}" = "\text{o}" \times
\]

\[
\text{'o'} > \text{'r'} = \text{False} \checkmark
\]

\[
\text{foldr \ insert} \ [\ ] "\text{t}" = "\text{t}"
\]

\[
\text{insert \ 'r'} \ "\text{t}" = "\text{r}"
\]

\[
\text{'o'} > \text{'r'} = \text{False}
\]

\[
\text{foldr \ insert} \ [\ ] "" = ""
\]

\[
\text{insert \ 't'} \ "" = "\text{t}"
\]

\[
\text{'r'} > \text{'t'} = \text{False}
\]
Higher-Order Algorithmic Debugging II

main = "os"

sort = {"sort" -> "os"}

foldr {‘s’ "o"->"os",’o’ "r"->"o","r’ "t"->"r","t’ ""->"t"} [] "sort" = "os"

foldr {‘s’ "o"->"os",’o’ "r"->"o","r’ "t"->"r","t’ ""->"t"} [] "ort" = "o"

foldr {‘s’ "o"->"os",’o’ "r"->"o","r’ "t"->"r","t’ ""->"t"} [] "rt" = "r"

foldr {‘s’ "o"->"os",’o’ "r"->"o","r’ "t"->"r","t’ ""->"t"} [] "t" = "t"

foldr {‘s’ "o"->"os",’o’ "r"->"o","r’ "t"->"r","t’ ""->"t"} [] "" = ""

insert ‘s’ "o" = "os"  insert ‘o’ "r" = "o"  insert ‘r’ "t" = "r"  insert ‘t’ "" = "t"

‘s’>’o’ = True  insert ‘s’ "" = "s"  ‘o’>’r’ = False  ‘r’>’t’ = False
Definition (Most evaluated form for finite maps)

\[
\text{mef}_G^M(n) = \begin{cases} 
\text{fMap}_G(n), & \text{if } M = f\ N_1\ldots\ N_k \land 0 \leq k < \text{arity}(f) \\
\emptyset, & \text{if } M = f\ N_1\ldots\ N_k \land k \geq \text{arity}(f) \\
M, & \text{otherwise}
\end{cases}
\]

where \( M = \text{mea}_G(n) \)

\[
\text{mea}_G(n) = \text{meaT}_G(G(\lceil n \rceil_G))
\]

\[
\text{meaT}_G(a) = a
\]

\[
\text{meaT}_G(m\ n) = \text{mea}_G(m) \ \text{mef}_G^M(n)
\]

\[
\text{fMap}_G(n) = \{ \text{mef}_G^M(o) \mapsto \text{mef}_G^M(m) \mid G(m) = n'\ o \land n' \succ^*_G\ n \land \text{mef}_G^M(m) \neq \emptyset \}
\]
Definition (Parent for finite maps)

\[ \text{parentFDT}_G = \text{parent} \cdot \text{fun}_G \]

\[ \text{fun}_G(n) = \begin{cases} 
  n, & \text{if } G(n) = a \\
  \text{fun}_G([m]_G), & \text{if } G(n) = m o 
\end{cases} \]
Conclusions

- Simple model amenable to proof.
- Contains a wealth of information about computation.
- Models real-world trace of Haskell tracer Hat.
- Proves soundness of algorithmic debugging.

http://www.haskell.org/hat
http://www.cs.kent.ac.uk/people/staff/oc/traceTheory.html