Declarative Debugging with Buddha

5th International Summer School on Advanced Functional Programming Tartu, Estonia

Bernie Pope

bjpop@cs.mu.oz.au

Department of Computer Science & Software Engineering

The University of Melbourne,

Victoria, Australia

Practicalities

For the past 50 years, software engineers have enjoyed tremendous productivity increases as more and more tasks have become automated. Unfortunately, debugging — the process of identifying and correcting a failure's root cause — seems to be the exception, remaining as labor intensive and painful as it was five decades ago.

A. ZELLER

Issues

- scalability

Issues

- scalability
- usability

Issues

- scalability
- usability

This third talk is a survey of features that will hopefully address the above two issues. Some are already implemented and others are “future work”.

Issues

- scalability
- usability

This third talk is a survey of features that will hopefully address the above two issues. Some are already implemented and others are “future work”.

If you are thinking of researching declarative debugging this might be a good place to start.

Trust

- long running programs produce VERY LARGE EDTs

Trust

- long running programs produce VERY LARGE EDTs
- large EDTs take up a lot of space

Trust

- long running programs produce VERY LARGE EDTs
- large EDTs take up a lot of space
- large EDTs take longer to search

Trust

- long running programs produce VERY LARGE EDTs
- large EDTs take up a lot of space
- large EDTs take longer to search
- a program can usually be divided into two parts: *suspicious* and *trusted* code

Trust

- long running programs produce VERY LARGE EDTs
- large EDTs take up a lot of space
- large EDTs take longer to search
- a program can usually be divided into two parts: *suspicious* and *trusted* code
- standard libraries are trusted

Trust

- long running programs produce VERY LARGE EDTs
- large EDTs take up a lot of space
- large EDTs take longer to search
- a program can usually be divided into two parts: *suspicious* and *trusted* code
- standard libraries are trusted
- trusted functions do not record nodes in the EDT

Trust

- long running programs produce VERY LARGE EDTs
- large EDTs take up a lot of space
- large EDTs take longer to search
- a program can usually be divided into two parts: *suspicious* and *trusted* code
- standard libraries are trusted
- trusted functions do not record nodes in the EDT
- trusted functions might have suspicious children

Trust

- long running programs produce VERY LARGE EDTs
- large EDTs take up a lot of space
- large EDTs take longer to search
- a program can usually be divided into two parts: *suspicious* and *trusted* code
- standard libraries are trusted
- trusted functions do not record nodes in the EDT
- trusted functions might have suspicious children
- trust and suspicion can be declared statically in `.opt` files (one per module)

Trust

- long running programs produce VERY LARGE EDTs
- large EDTs take up a lot of space
- large EDTs take longer to search
- a program can usually be divided into two parts: *suspicious* and *trusted* code
- standard libraries are trusted
- trusted functions do not record nodes in the EDT
- trusted functions might have suspicious children
- trust and suspicion can be declared statically in `.opt` files (one per module)
- static declarations can be made cheap by changing the transformation of trusted functions

Judgement

Buddha provides four judgement values:

- correct
- erroneous
- inadmissible
- unknown

Inadmissible

```
[32] Main 12 merge
   arg 1  = [3,1,2]
   arg 2  = [5,6]
   result = [3,1,2,5,6]
```

A derivation is inadmissible when the function was applied to arguments for which it has no intended meaning.

It has the same effect as answering correct, but it is closer to intuition.

Unknown

Some derivations are too hard to judge.

Unknown

Some derivations are too hard to judge.

- the arguments and/or result might be very large

Unknown

Some derivations are too hard to judge.

- the arguments and/or result might be very large
- higher-order code can add complexity

Unknown

Some derivations are too hard to judge.

- the arguments and/or result might be very large
- higher-order code can add complexity
- there might be a lot of thunks

Unknown

Some derivations are too hard to judge.

- the arguments and/or result might be very large
- higher-order code can add complexity
- there might be a lot of thunks

It is best to defer difficult derivations. However, deferal might only postpone the inevitable need to make a judgement.

Unknown

Some derivations are too hard to judge.

- the arguments and/or result might be very large
- higher-order code can add complexity
- there might be a lot of thunks

It is best to defer difficult derivations. However, deferal might only postpone the inevitable need to make a judgement.

As a last resort you can declare a derivation to be unknown. This has the same effect as “correct” except buddha will remind you when a bug diagnosis depends on unknown derivations.

I/O

```
main
   = do c <- getChar
        d <- foo c
        print (c, d)

foo x
   = do putChar x
        getChar
```

The value of an I/O function has two parts:

- the “returned” result (main returns unit, foo returns a character)
- a sequence of primitive I/O actions

I/O tabling

```
type IO a = World -> (World, a)
```

I/O tabling

```
type IO a = World -> (World, a)

type World = Int
```

I/O tabling

```
type IO a = World -> (World, a)
type World = Int
```

Action No.	Value
1	GetChar 'a'
2	PutChar 'a'
3	GetChar 'b'
4	PutStr "('a', 'b')"

I/O tabling

```
type IO a = World -> (World, a)
```

```
type World = Int
```

Action No.	Value
1	GetChar 'a'
2	PutChar 'a'
3	GetChar 'b'
4	PutStr "('a', 'b')"

```
main = { 1 -> (4, ()) }
```

I/O tabling

```
type IO a = World -> (World, a)

type World = Int
```

Action No.	Value
1	GetChar ’a’
2	PutChar ’a’
3	GetChar ’b’
4	PutStr "(’a’, ’b’)"

```
main = { 1 -> (4, ()) }

foo 'a' = { 2 -> (3, 'b') }
```

I/O tabling

```
type IO a = World -> (World, a)
```

```
type World = Int
```

Action No.	Value
1	GetChar ’a’
2	PutChar ’a’
3	GetChar ’b’
4	PutStr "(’a’, ’b’)"

```
main = { 1 -> (4, ()) }
```

```
foo 'a' = { 2 -> (3, 'b') }
```

Question: how to handle threaded execution?

Re-evaluation

- perhaps we can't fit all of the EDT in memory at one time

Re-evaluation

- perhaps we can't fit all of the EDT in memory at one time
- trade space for execution time

Re-evaluation

- perhaps we can't fit all of the EDT in memory at one time
- trade space for execution time
- only store the top few levels of the EDT in memory

Re-evaluation

- perhaps we can’t fit all of the EDT in memory at one time
- trade space for execution time
- only store the top few levels of the EDT in memory
- when EDT traversal reaches the fringe, the tree is expanded a number of levels from that node down

Re-evaluation

- perhaps we can’t fit all of the EDT in memory at one time
- trade space for execution time
- only store the top few levels of the EDT in memory
- when EDT traversal reaches the fringe, the tree is expanded a number of levels from that node down
- the expansion of the tree is done by re-executing the call at the fringe node

Re-evaluation

- perhaps we can’t fit all of the EDT in memory at one time
- trade space for execution time
- only store the top few levels of the EDT in memory
- when EDT traversal reaches the fringe, the tree is expanded a number of levels from that node down
- the expansion of the tree is done by re-executing the call at the fringe node
- a difficulty is with nodes whose result is only partially computed

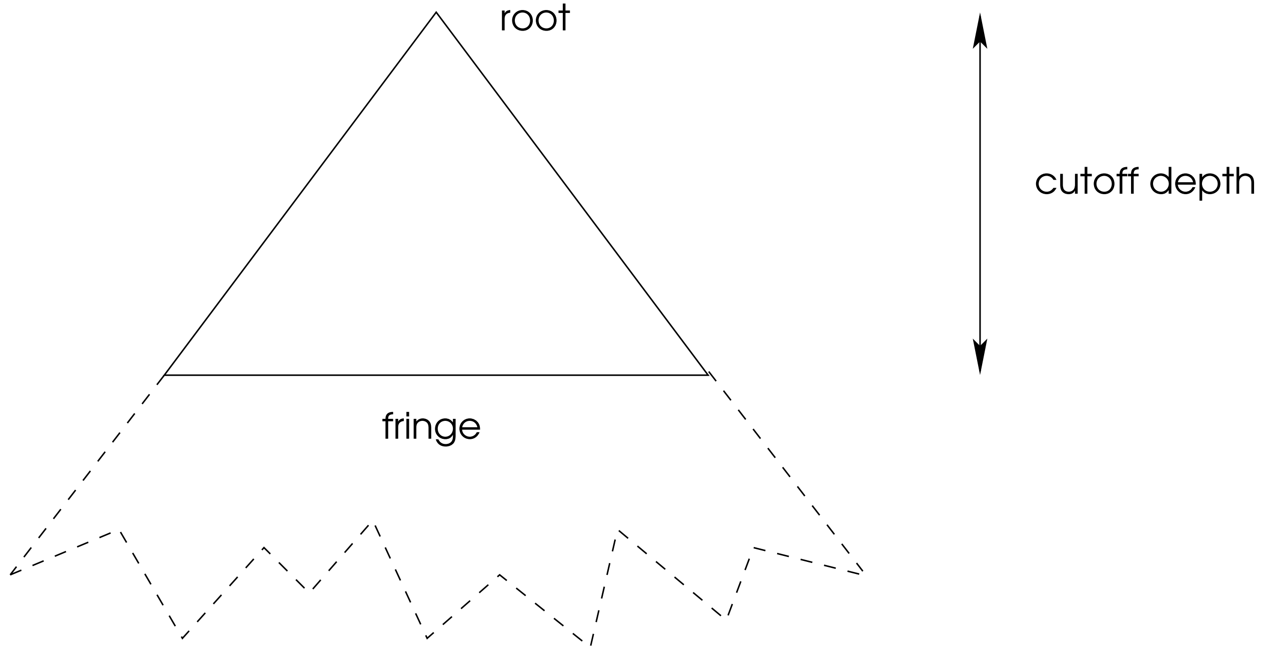
Re-evaluation

- perhaps we can’t fit all of the EDT in memory at one time
- trade space for execution time
- only store the top few levels of the EDT in memory
- when EDT traversal reaches the fringe, the tree is expanded a number of levels from that node down
- the expansion of the tree is done by re-executing the call at the fringe node
- a difficulty is with nodes whose result is only partially computed
- another difficulty is re-evaluation of I/O calls - we want idempotency (ala Mercury debugger)

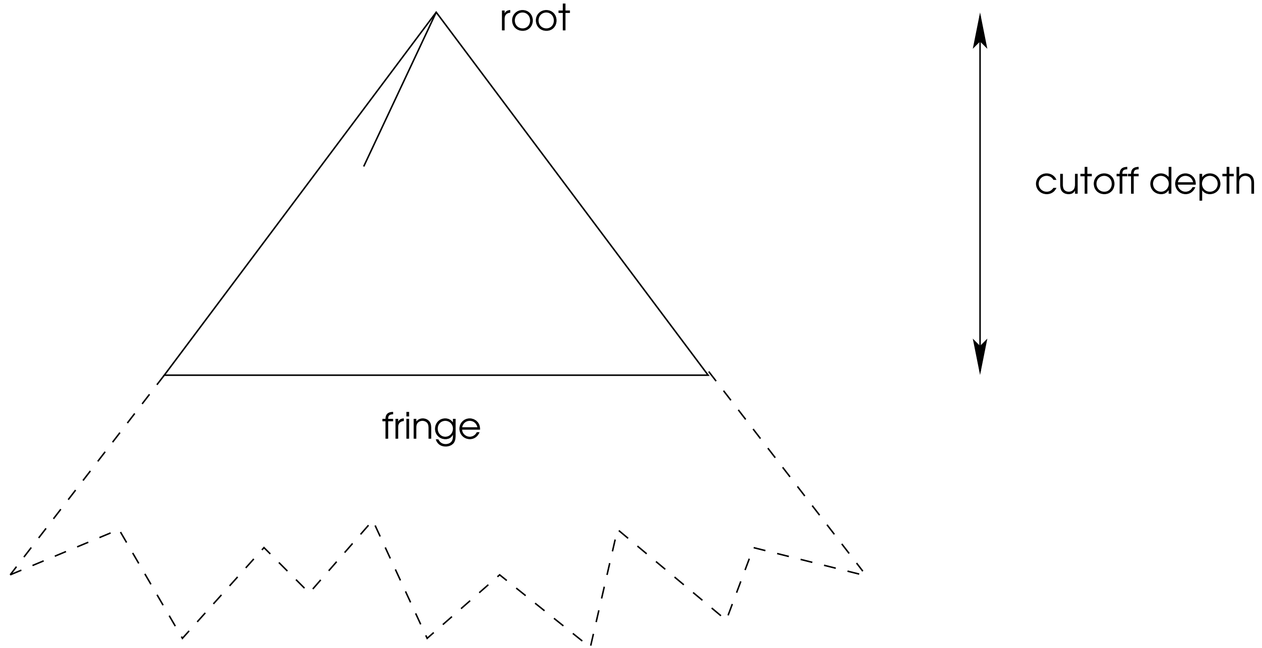
Re-evaluation

- perhaps we can't fit all of the EDT in memory at one time
- trade space for execution time
- only store the top few levels of the EDT in memory
- when EDT traversal reaches the fringe, the tree is expanded a number of levels from that node down
- the expansion of the tree is done by re-executing the call at the fringe node
- a difficulty is with nodes whose result is only partially computed
- another difficulty is re-evaluation of I/O calls - we want idempotency (ala Mercury debugger)
- idempotency can be implemented on top of tabled I/O

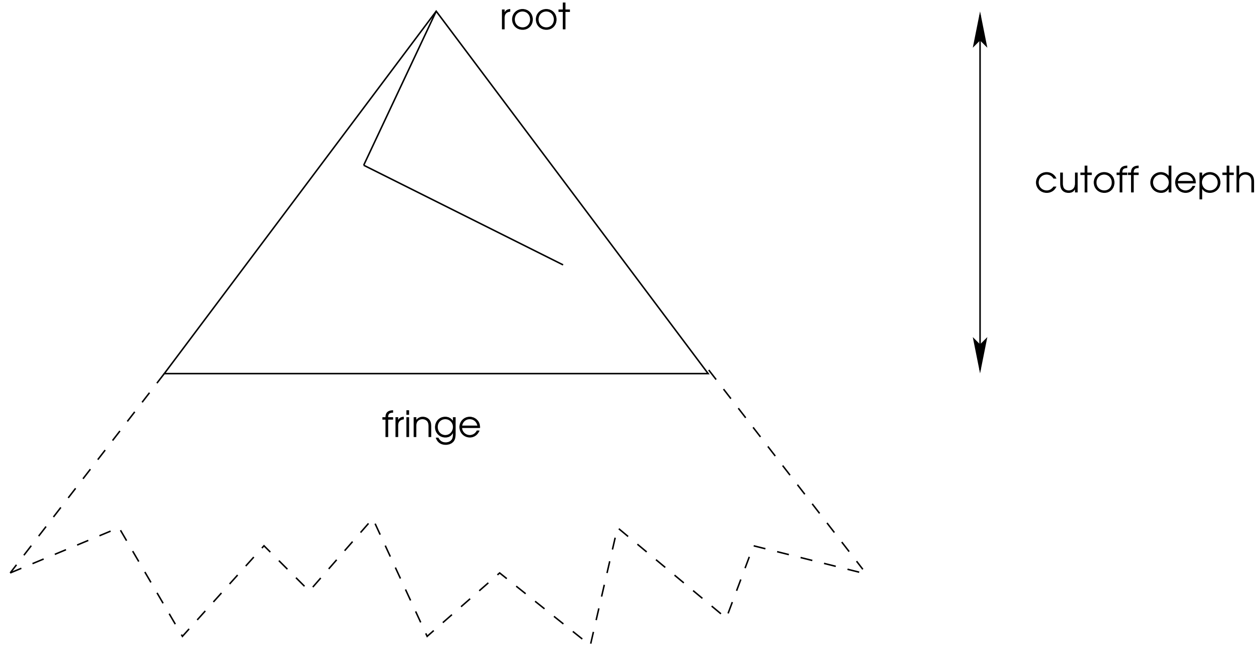
Re-evaluation



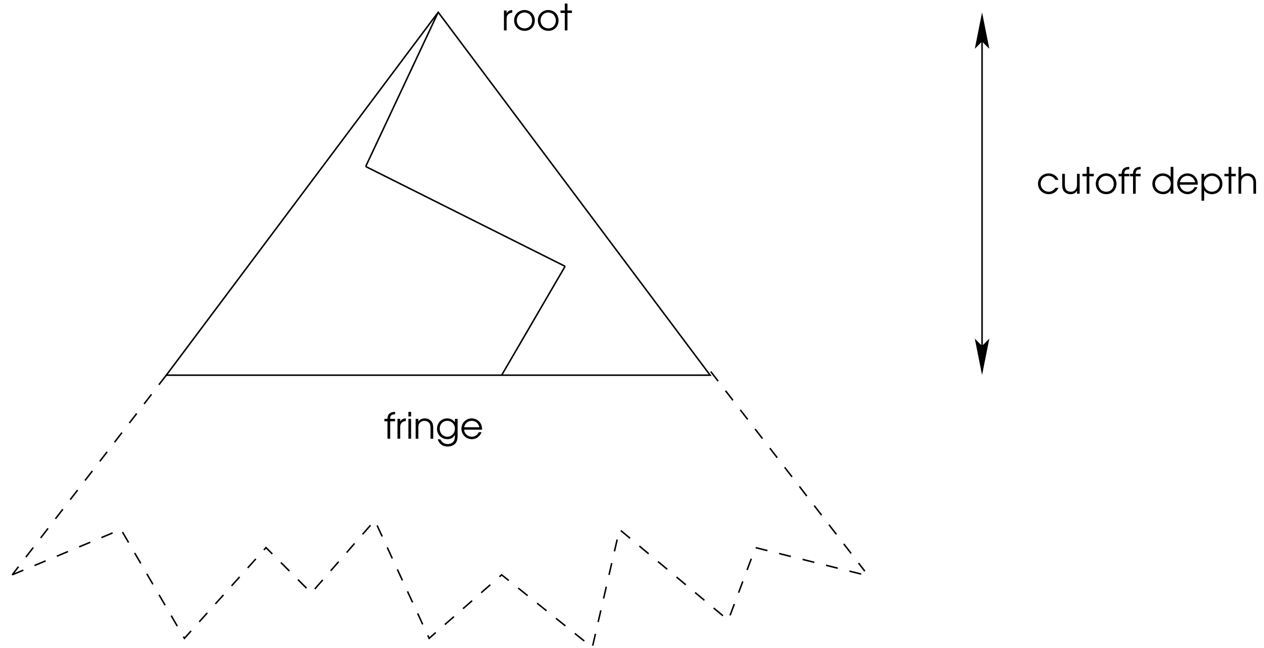
Re-evaluation



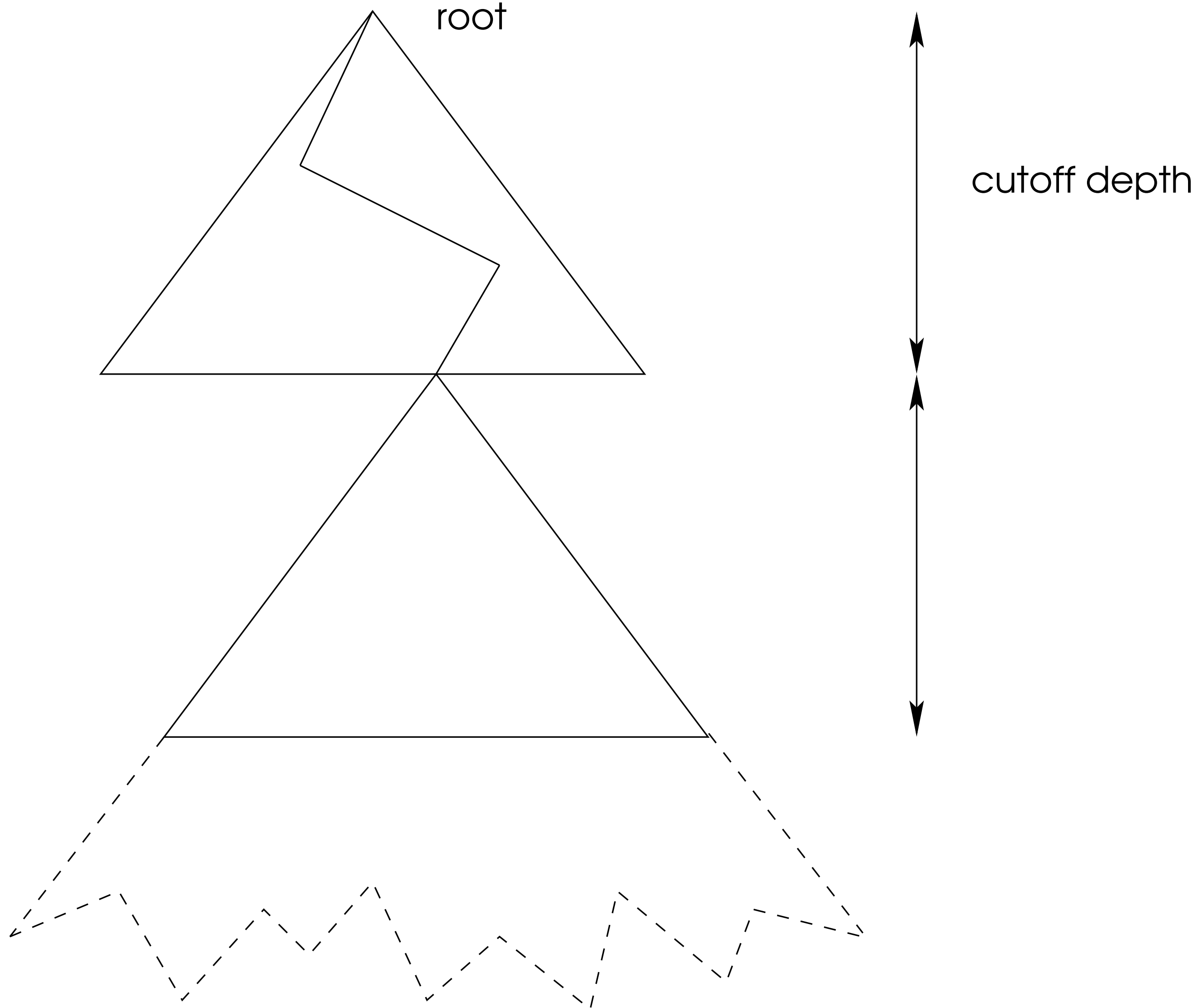
Re-evaluation



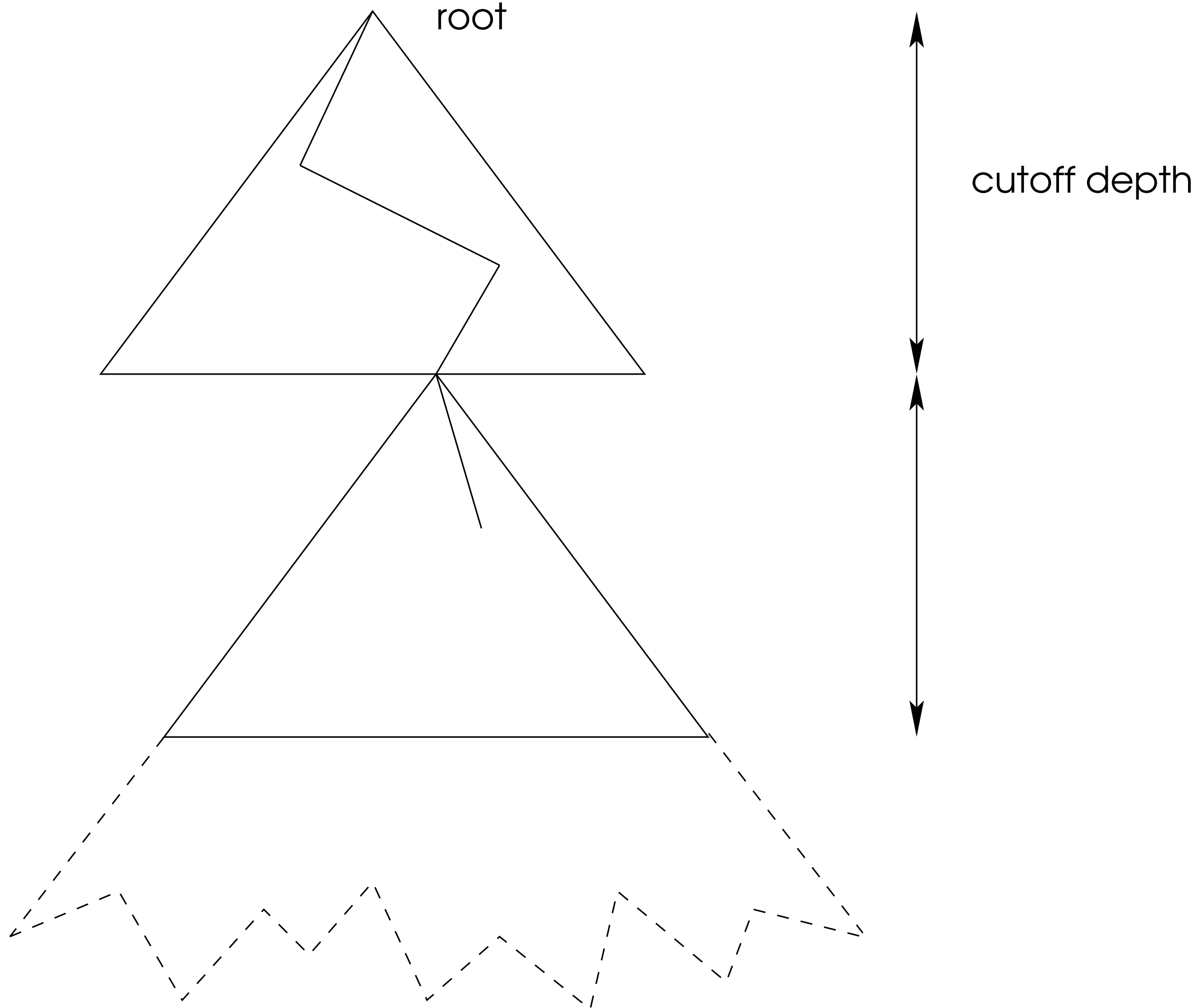
Re-evaluation



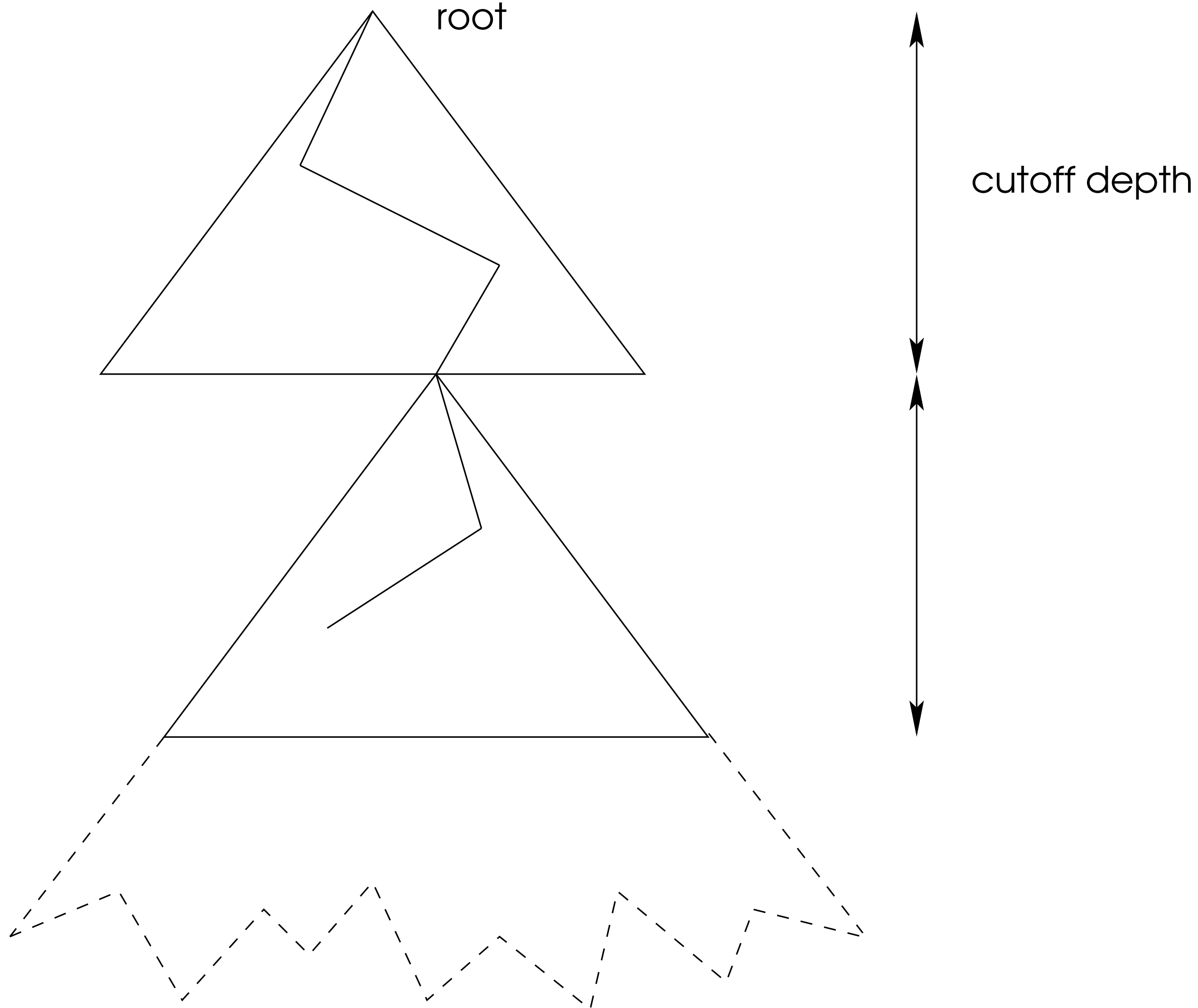
Re-evaluation



Re-evaluation



Re-evaluation



Threaded execution

- run the debugger and debuggee in seperate threads

Threaded execution

- run the debugger and debuggee in seperate threads
- interleave exeuction of the debuggee and traversal of the EDT

Threaded execution

- run the debugger and debuggee in seperate threads
- interleave exeuction of the debuggee and traversal of the EDT
- debugging is more “immediate”

Threaded execution

- run the debugger and debuggee in seperate threads
- interleave exeuction of the debuggee and traversal of the EDT
- debugging is more “immediate”
- diagnosis has to deal with a partial tree

Threaded execution

- run the debugger and debuggee in seperate threads
- interleave exeuction of the debuggee and traversal of the EDT
- debugging is more “immediate”
- diagnosis has to deal with a partial tree
- EDT can be pruned along the way to keep space usage down

Threaded execution

- run the debugger and debuggee in seperate threads
- interleave exeuction of the debuggee and traversal of the EDT
- debugging is more “immediate”
- diagnosis has to deal with a partial tree
- EDT can be pruned along the way to keep space usage down
- easy to implement on top of the tabled construction of the EDT

Large derivations

. . . are hard to judge.

Possible improvements:

Large derivations

. . . are hard to judge.

Possible improvements:

- a query language

Large derivations

. . . are hard to judge.

Possible improvements:

- a query language
- custom printing routines

Large derivations

. . . are hard to judge.

Possible improvements:

- a query language
- custom printing routines
- take advantage of type information

Large derivations

. . . are hard to judge.

Possible improvements:

- a query language
- custom printing routines
- take advantage of type information

```
reverse ::  [a] -> [a]
```

No need to completely show the values in the list, a (small) label will do.

Search strategies

The current simple search strategy:

```
debug :: Diagnosis -> [EDT] -> IO Diagnosis
debug diagnosis [] = return diagnosis
debug diagnosis (node:siblings)
   = do let d = derivation node
        judgement <- askOracle d
        case judgement of
           Correct
              -> debug diagnosis siblings
           Erroneous
              -> debug (Buggy d)
                       (children node)
```

Search strategies

The current search strategy is top-down left-to-right.

Can we do any better? This is one place we can "put the computer to work".

Search strategies

The current search strategy is top-down left-to-right.

Can we do any better? This is one place we can “put the computer to work”.

- deep recursive chains (binary search)

Search strategies

The current search strategy is top-down left-to-right.

Can we do any better? This is one place we can “put the computer to work”.

- deep recursive chains (binary search)
- probabilistic search — where are bugs more likely to occur, based on nodes we have already seen and other hueristics (Lee Naish)

Search strategies

The current search strategy is top-down left-to-right.

Can we do any better? This is one place we can "put the computer to work".

- deep recursive chains (binary search)
- probabilistic search — where are bugs more likely to occur, based on nodes we have already seen and other hueristics (Lee Naish)
- we want the search path to be comprehensible to the programmer (jumping all over the tree is a bad idea)

Related Work

- Hat
- Hood
- Freya
- Mercury debugger
- Curry debugger

Conclusion

When you can’t think of your own conclusion, steal someone else’s:

Conclusion

When you can't think of your own conclusion, steal someone else's:

> *... debugging can be a messy problem, if one creates a messy environment in which it has to be solved, but it is not inherently so. The theory of algorithmic debugging suggests that if the programming language is simple enough, then programs in it can be debugged semi-automatically on the basis of several simple principles.*

EHUD SHAPIRO