Haskell Dynamic Tracing

Ondřej Kvapil

17th June, 2021

Czech Technical University in Prague, Faculty of Information Technology

Introduction

- Haskell is a lazy language

Introduction

- Haskell is a lazy language
- Delays evaluation of an expression until its value is needed

Introduction

- Haskell is a lazy language
- Delays evaluation of an expression until its value is needed

```
snd :: (Int, Int) -> Int
snd (x, y) = y
```

Introduction

- Haskell is a lazy language
- Delays evaluation of an expression until its value is needed

```haskell
snd :: (Int, Int) -> Int
snd (x, y) = y

foo  = snd (complexComputation, 3)
```

Introduction

- Haskell is a lazy language
- Delays evaluation of an expression until its value is needed

```haskell
snd :: (Int, Int) -> Int
snd (x, y) = y

foo  = snd (complexComputation, 3)
foo' = snd (error "oops!",      3)
```

Introduction

- Haskell is a lazy language
- Delays evaluation of an expression until its value is needed
- Shares evaluated subexpressions

```
snd :: (Int, Int) -> Int
snd (x, y) = y

foo  = snd (complexComputation, 3)
foo' = snd (error "oops!",      3)
```

Introduction

- Haskell is a lazy language
- Delays evaluation of an expression until its value is needed
- Shares evaluated subexpressions

```
snd :: (Int, Int) -> Int
snd (x, y) = y

foo  = snd (complexComputation, 3)
foo' = snd (error "oops!",      3)

let x = complexComputation
in  f (x, x)
```

Motivation

- Expressiveness comes at a cost

Motivation

- Expressiveness comes at a cost
- Memory overhead, garbage collector pressure, $\perp$, …

Motivation

- Expressiveness comes at a cost
- Memory overhead, garbage collector pressure, $\bot$, …
- *Is it worth it?*

Goals

- Key question: *how is laziness used in practice*?

Goals

- Key question: *how is laziness used in practice*?
- Develop a dynamic tracing framework

Goals

- Key question: *how is laziness used in practice*?
- Develop a dynamic tracing framework
- Collect and analyse real-world programs

Our solution

- Compiler plugin for the Glasgow Haskell Compiler

Our solution

- Compiler plugin for the Glasgow Haskell Compiler
- Transforms surface syntax of Haskell to add tracing calls

Our solution

- Compiler plugin for the Glasgow Haskell Compiler
- Transforms surface syntax of Haskell to add tracing calls
- Traces evaluation of function calls and function arguments

Our solution

```
snd (x, y) = y
main = print $ snd (error "oops!", 3)

-- rewrites to --
snd (x, y)
  = let call_number_0 = traceEntry "snd"
        in (traceArg "snd") "y" call_number_0 y
main
  = let call_number_1 = traceEntry "main"
        in print $ snd (error "oops!", 3)
```

Our solution

Timestamp	Thread ID	Trace type	Function	Call ID	Argument	Closure
437163913	ThreadId 1	EntryTrace	snd	1		
437066481	ThreadId 1	ArgTrace	snd	1	y	constr

Our solution

```
qsort [] = []
qsort (a:as) = qsort left ++ [a] ++ qsort right
        where (left, right) = (filter (<=a) as, filter (>a) as)

main = print $ qsort [1 + 8, 4, 0, 3, 1, 23, 11, 18]
```

Our solution

```
qsort (a : as) = let !call_number_1 = traceEntry "qsort"
                 in qsort left
                       ++ [(traceArg "qsort") "a" call_number_1 a]
                       ++ qsort right
  where
  (left, right) = let !call_number_2 = traceEntry "qsort"
                  in (filter
                        (<= (traceArg "qsort") "a" call_number_2 a)
                        (traceArg "qsort") "as" call_number_2 as,
                      filter
                        (> (traceArg "qsort") "a" call_number_2 a)
                        (traceArg "qsort") "as" call_number_2 as)
```

Our solution

Timestamp	Thread ID	Trace type	Function	Call ID	Argument	Closure
950778579	ThreadId 1	EntryTrace	qsort	1		
949631938	ThreadId 1	ArgTrace	qsort	1	as	constr
951867310	ThreadId 1	ArgTrace	qsort	1	a	thunk
952240416	ThreadId 1	EntryTrace	qsort	2		
952223429	ThreadId 1	ArgTrace	qsort	2	as	thunk
952269949	ThreadId 1	ArgTrace	qsort	2	a	constr
...						

Summary

- Compile-time rewriting of programs via a plugin for the Glasgow Haskell Compiler

Summary

- Compile-time rewriting of programs via a plugin for the Glasgow Haskell Compiler
- Annotation with side-effecting tracing functions

Summary

- Compile-time rewriting of programs via a plugin for the Glasgow Haskell Compiler
- Annotation with side-effecting tracing functions
- Compiled program records a trace of relevant events alongside regular output