Haskell Dynamic Tracing

Ondřej Kvapil, supervised by Ing. Filip Křikava, PhD.

28th April, 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 :: Int
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' :: Int
foo' = snd (error "oops!", 3)
```

Introduction

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

Introduction

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

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

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
- Tool capable of analysing real-world Haskell programs

Motivation

- Interesting landscape – Haskell is heavily used in academia

Motivation

- Interesting landscape – Haskell is heavily used in academia
- Lack of empirical results despite extensive theoretical study

Motivation

- Interesting landscape – Haskell is heavily used in academia
- Lack of empirical results despite extensive theoretical study
- Laziness has advantages as well as significant downsides

State-of-the-art

- Unnecessary laziness leads to large memory overhead

State-of-the-art

- Unnecessary laziness leads to large memory overhead
- Some tools designed to avoid too much laziness (`nothunks`, `BangPatterns`)

State-of-the-art

- Unnecessary laziness leads to large memory overhead
- Some tools designed to avoid too much laziness (`nothunks`, `BangPatterns`)
- Black-box problem: developers lack insight about the runtime state

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

```
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

Thank you for listening.

Kvapil, Ondřej. *Haskell Dynamic Tracing*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

`https://gitlab.fit.cvut.cz/kvapiond/bachelors-thesis`
`https://github.com/viluon/bachelors-thesis`