FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Haskell Dynamic Tracing
Student:	Ondřej Kvapil
Supervisor:	Ing. Filip Křikava, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2020/2021

Instructions

Lazy evaluation is a strategy that delays expression evaluation until its value is needed. This allows one to avoid unnecessary computation and use of infinite data structures. Recently, Goel and Vitek looked into the use of laziness in R [1], which is one of the most widely used lazy programming languages. They found little evidence supporting that programmers use laziness to save on computation or use infinite data structures. It would be interesting to compare this to the use of laziness in Haskell. For this, we need a way to trace the execution of real-world Haskell programs.
The goal of this thesis is, therefore, to design and implement a dynamic tracing framework for Haskell. It shall be scalable in order to allow us to analyze a large corpus of Haskell code available on GitHub. The dynamic tracer should capture all interesting events such function call and argument order evaluation and present them in an easy to be queried form.

–

[1] DOI: 10.1145/3360579

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 4 February 2020 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Haskell Dynamic Tracing

Ondřej Kvapil

Programming Research Laboratory
Supervisor: Ing. Filip Křikava, Ph.D.

May 13, 2021

Acknowledgements

THANKS (remove entirely in case you do not wish to thank anyone)

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 13, 2021 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

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

Abstrakt

Líné vyhodnocování je potenciálně mocná implementační strategie pro non-strict programovací jazyky, která umožňuje programátorům soustředit se na to, co program znamená, aniž by byli rušeni způsobem jeho vyhodnocení. Lenost přináší možnost přirozeně vyjádřit uživatelem definované řídící konstrukce a vede k vyhodnocení jen potřebné části programu. Odklad vyhodnocování ale komplikuje analýzu složitosti a může vést k těžko předvídatelnému paměťovému chování. Pro lepší pochopení kompromisů spojených s leností a jejího využití v praktických situacích jsme navrhli zásuvný modul pro dynamické trasování do kompilátoru Glasgow Haskell Compiler, naimplementovali prototyp a ukázali jeho schopnost zachytit klíčové informace o využití lenosti v jednoduchých Haskell programech.

Klíčová slova Haskell, dynamické trasování, líné vyhodnocování, zásuvné moduly kompilátorů, generické programování

Abstract

Lazy evaluation is a potentially powerful implementation strategy for non-strict languages, freeing the programmer to focus on what a program means rather than on how it is computed. Laziness naturally accommodates user-defined control flow and evaluates only the required subset of a given program in a demand-driven manner. However, delayed evaluation makes complexity analysis challenging and can lead to hard-to-predict memory behaviour. To better understand the trade-offs laziness offers and how it is used in practical scenarios, we design a dynamic tracing plugin for the Glasgow Haskell Compiler, implement a proof of concept, and demonstrate its ability to record crucial information about the use of laziness in simple Haskell programs.

Keywords Haskell, dynamic tracing, lazy evaluation, compiler plugins, generic programming

Contents

List of Figures

List of Listings

List of Tables

Introduction

Conventional programming languages of all paradigms use – almost equivocally – eager evaluation strategies. Non-strict semantics has far-reaching implications on the design of a language [1] and comes with both benefits in expressiveness and implementation challenges.

```haskell
-- snd is non-strict in the first
-- component of the pair
snd :: (Int, Int) -> Int
snd (x, y) = y

-- purity and laziness: foo reduces to 3,
-- complexComputation is not evaluated
foo = snd (complexComputation, 3)

-- non-strict semantics can prevent
-- runtime errors
foo' = snd (error "oops!", 3)

-- computations are shared,
-- even across threads
bar = let x = complexComputation
      in x `par` f x
```

Listing 1: Example expressions where the semantics of Haskell notably differ from that of strict languages.

Lazy evaluation is a potentially powerful implementation strategy for non-strict languages, freeing the programmer to focus on what a program means rather than on how it is computed. Laziness naturally accommodates user-defined control flow and evaluates only the required subset of a given program in a demand-driven manner. The non-strict semantics of the Haskell language were a guiding principle which influenced or directly determined many of the

decisions made at its inception [2]. However, the implementation of non-strict features via laziness in GHC brings many pitfalls which Haskell programmers need to deal with. Automatic avoidance of unnecessary thunk allocations is conservative [3]: if GHC is unable to prove the strictness of a function in an argument by static strictness analysis, the function will remain lazy, possibly leading to pathological memory behaviour at runtime.

[[bridge this gap]]

This fight against the semantics is detrimental to the developer experience of the language. The question arises whether the benefits of laziness outweigh the toll it takes on the programmer. To answer this question, the runtime behaviour of lazy features needs to be understood. As a first step towards that understanding, we design a dynamic tracing tool capable of capturing information about the runtime behaviour of non-strict functions.

Chapter 1

State-of-the-art

Although there are many functional languages of the ML family which enjoy widespread use (F#, OCaml, SML), Haskell is the only non-strict language among them. The Glasgow Haskell Compiler (GHC) implements Haskell's non-strict semantics by lazy evaluation facilitated mainly by a runtime data structure called a *thunk*, which represents delayed computations.

Laziness leads to many issues with runtime behaviour of Haskell programs, although it is an efficient implementation of non-strict semantics as required by the Haskell spec [4]. The accumulation of thunks at runtime is a frequent cause of pathological memory behaviour and unpredictable performance. There is a number of libraries and tools which aim to help the Haskell programmer inspect the runtime state of the Haskell heap, force the evaluation of thunks known to be forced by the program at a later point anyway, and avoid their creation altogether for certain expressions.

We surveyed several debuggers and solutions for the inspection and management of thunks. We found no existing tools that would directly capture enough information to be suitable for dynamic tracing and strictness analysis, but two came close (Hat and `ghc-heap-view`).

1.1 The Glasgow Haskell Compiler

GHC is the most widespread Haskell distribution. Its plethora of language extensions [5], which range from simple syntactical utilities to complex type system add-ons, lets the programmer customise the set of features provided by the language. We briefly discuss the internal organisation of the project and in the process explain those basics of the Haskell language that are needed for the later chapters.

1.1.1 Architectural overview

Although a thorough and authoritative – if a little dated – description of the architecture of the compiler is available in the aptly named, freely accessible Architecture of Open Source Applications [6], we include a summary of the key points relevant to our work as well as to the discussed technicalities. GHC is an optimising compiler for the Haskell language. The project consists of three major components: (1) the compiler itself, (2) the boot libraries (a collection of core libraries GHC itself depends on), and (3) the Runtime System (RTS, a large library of C code linked into every compiled program). RTS provides low-overhead runtime support for facilities abstracted over by Haskell code such as garbage collection, exception handling, or concurrency primitives.

The compiler turns Haskell source code to object and interface files[1]. The process is organised in a pipeline that consists of the following phases:

Parsing constructs abstract syntax trees. Lexical and syntactical errors are reported here.

Renaming resolves identifiers into fully qualified names. Undefined references are reported here. The renaming phase reassociates operator applications in the AST formed during parsing, this is because Haskell allows specifying the precedence and associativity of infix operators, but their properties are only available after their references have been resolved.

Typechecking verifies the program's type-correctness. Type checking annotates all binders in the program with type signatures. Type errors are reported here.

Desugaring converts Haskell surface syntax to the much smaller intermediate language, Core.

Simplification performs optimisations on the Core language, including demand analysis, `let` floating, dead-code elimination, common subexpression elimination, constructor specialisation, and others.

Conversion to STG translates Core to the language of the Spineless Tagless G-machine, suitable for code generation.

Code generation produces machine code or LLVM bit code for further processing by the LLVM toolchain.

[1] These describe high-level information about a compiled module, including data type definitions and inlineable functions.

Compiler front end

Phases of the pipeline from parsing to desugaring form the compiler front end. It starts out with a textual representation and gradually transforms it into increasingly structured data before passing it on to the back end. During this process, the front end identifies and reports all errors in the user's code.

As the program flows through the pipeline, its invariants gradually change. The codebase reflects this by passing different data types from phase to phase. The types of the nodes of the surface syntax tree are indexed by an uninhabited type `GhcPass`, which is itself indexed by the `Pass` data type (i.e. `GhcPass` has kind `Pass -> *`), lifted to the type level. Together, the `GhcPass` types represent the various phases of the compiler front end, from parsing to renaming to type checking. The type level distinction between phases complicates the type signatures of almost all functions in the pipeline, but the choice comes with important benefits. First, indexing AST types by the `GhcPass` types provides a compile-time guarantee that nodes from different phases cannot be mixed unintentionally. Second, the phase type parameter allows one to use the Tress that Grow pattern [7], that enables easy extensions of both sum and product types at various phases.

Trees that Grow

Let us take a small detour to explain the concept of the design pattern, invented specifically to add extensibility to GHC's abstract syntax data types. The basic algorithm for making a data type extensible is as follows:

1. index the data type of choice by a type parameter ξ, called the *extension descriptor*,

$$\texttt{data D = } \ldots \rightarrow \texttt{data D } \xi \texttt{ = } \ldots$$

2. add one new constructor Extra to the data type,

$$\begin{array}{c} \texttt{data D } \xi \texttt{ = C}_1 \ \ldots \ \texttt{|} \ \ldots \ \texttt{|} \ \texttt{C}_n \ \ldots \\ \downarrow \\ \texttt{data D } \xi \texttt{ = C}_1 \ \ldots \ \texttt{|} \ \ldots \ \texttt{|} \ \texttt{C}_n \ \ldots \ \texttt{|} \ \texttt{Extra} \end{array}$$

3. create a *type family* X_{Con} – a function from types to types – for all constructors,

$$\begin{array}{l} \texttt{type family X}_{C_1} \quad \xi \\ \vdots \\ \texttt{type family X}_{C_n} \quad \xi \\ \texttt{type family X}_{Extra} \ \xi \end{array}$$

4. add one field of type $X_{\text{Con}}\ \xi$ to every constructor.

```
data D ξ = C1 ... | ... | Extra
                ↓
data D ξ = C1 (X_C1 ξ) ... | ... | Extra (X_Extra ξ)
```

This small refactoring enables the programmer to both restrict the use of certain constructors and introduce new constructors depending on the extension descriptor ξ. The programmer can apply these modifications by manipulating the definitions of the type families rather than the original data type itself. It also lets the programmer add new fields to the existing constructors, again depending on the particular type ξ.

To define the original data type without extensions in terms of its extensible variant, it suffices to fix the extension descriptor to some type, *e.g.* to `Void`, and omit any equations for the type families. Doing so leaves the type level applications of the shape X_{Con} Void irreducible and thus isomorphic to any empty type, with the only valid value being $\bot$ (such as `undefined`). In effect, the extension fields of constructors cannot be pattern-matched against, because they have no constructors. The extension constructor Extra can still be matched, but cannot hold any data. It can be hidden completely by not exporting it from the module of definition.

For extensions, it suffices to add type family instances – the analogy of function equations for type functions – which resolve a particular assignment of the extension descriptor to the desired type of the extension.

As presented, the Trees that Grow transformation leaves much to be desired from a usage perspective: we have to pass a `void` or `undefined` for unused extension fields during construction, these extensions also clutter the pattern matches, and matching on both constructors and multiple fields added via extensions is clunky at best. These grievances can be solved by the use of a convenient syntactical feature of Haskell called *pattern synonyms*[8]. These let the programmer abstract over patterns and so define reusable interfaces to the data types extended via the Trees that Grow transformation, hiding the structural complexity of the underlying flexible data type.

For an in-depth description of the design pattern, its generalisations to multiple type parameters, existentials, GADTs, hierarchies of extension descriptors, as well as relations to generic programming, typeclasses, and for many other practically useful details, we recommend the publication introducing the idea [7].

Although the Trees that Grow pattern is not used universally throughout the GHC project, its concepts play an important role in many of the core data types.

Compiler back end

The back end of the compiler starts with the desugaring phase, which translates the resolved and type checked surface syntax into an Intermediate Representation (IR) called Core. The Haskell language contains many redundancies and shorthands designed to make the syntax more user-friendly. The AST data types contains hundreds of constructors. In contrast, Core only has about 10 syntactical forms. Essentially, it is a variant of System F extended with type equality coercions [9].

Although Core is typed, the compiler only type checks Core programs if the user explicitly asks for it. Core types exist mostly to validate the internal consistency of the compiler – desugaring a Haskell program which passed typechecking into incorrectly typed Core would be a compiler bug.

Most of the optimisation passes GHC performs are local semantics-preserving transformations of Core (Core-to-Core passes) which are applied in many iterations during invocations of the simplifier [10]. These include *e.g.* constant folding, inlining, or fusion of nested case expressions. The local rewritings improve intermediate code between applications of heavier optimisations, such as specialisation (to eliminate overloading), demand analysis, `let` floating, and others.

The optimised Core is transformed to a slightly different representation which corresponds to programs of an abstract graph reduction machine, the Spineless Tagless G-machine (STG) ([11], later revised in [12]). It is translated again to the low-level imperative language `Cmm`, a dialect of C−−, before entering one of the final stages of the code generation phase. A successful run of the compiler typically terminates in GHC's built-in native code back end, but the `Cmm` representation can be translated to LLVM bit code and additionally processed by the LLVM pipeline.

A notable divergence in the compiler is the bytecode compilation pipeline. Bytecode is executed by the RTS interpreter, the backbone of GHC's interactive interface (GHCi). GHCi includes a debugger which can pause and resume the evaluation of an interpreted Haskell program and print the runtime values of local bindings. We will discuss GHCi in greater detail in Chapter 2. The bytecode pipeline does not involve optimisations, the conversion to STG, or any later passes. Instead, the separate generator translates Core directly to bytecode instructions, although this is about to change in GHC 9.2 [13].

Compiler plugins

Both the front end and the back end of the compiler can be modified or extended in a modular way using *compiler plugins*, which come in two flavours: core plugins on the back end and source plugins on the front end [14]. The former act on the Core language and are best suited for optimisations, while high level analyses, language extensions and code generation are better handled by the latter. We will discuss source plugins in more detail in chapter 2.

Runtime System

The runtime system consists of about 50,000 lines of C and C−− code. It implements all the functionality Haskell programs require that is not compiled into the programs themselves, much of which involves low-level interactions with abstractions provided by the operating system. The major components of the RTS are the following:

- A user-space scheduler which multiplexes lightweight Haskell threads onto heavy OS threads,
- a storage manager, including a block allocation layer, which abstracts over memory management, and a parallel generational garbage collector,
- primitives for exception handling, concurrency, and built-in operations,
- a bytecode interpreter and a dynamic linker for GHCi, and
- support for Software Transactional Memory (STM).

The scheduler is at the heart of the RTS. Haskell threads yield to the scheduler when their assigned slice of execution time expires, when they run out of heap or stack space, or when they need to switch between machine code execution and bytecode interpretation. Any foreign calls into or out of Haskell need to pass through the scheduler as well.

The storage manager defines the data structures which represent Haskell values at runtime. Since the understanding of these representations is crucial for the understanding of the implementation of laziness in GHC and the trade-offs involved, we will discuss the relevant parts of the storage manager here.

Closures (the runtime objects of programs compiled with GHC) share the same basic representation shown in Figure 1.1. The *header* contains primarily a pointer to the metadata of a closure, though it also includes a profiling header if profiling is enabled. The *payload* of a closure usually contains data not known at compile time. The *info table* identifies the type of the closure (data constructor, function, thunk, ...). It informs the garbage collector about the pointerhood of the payload. The *entry code* is the code executed when *entering*, *i.e.* evaluating the closure. For example, the entry code of functions represents the body of the function.

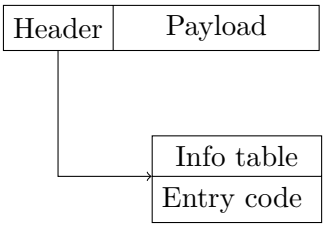

Figure 1.1: The memory layout of a generic closure.

The STG uses a number of registers, a heap, and a stack which stores function arguments and continuations. Closures can also reside statically in the compiled object code of a Haskell program. During execution, any heap allocations are preceded by a *heap check*, which invokes garbage collection if not enough space is left on the heap. Similarly, when code needs to push values onto the stack, it performs a *stack check* and grows the stack if necessary.

All the dynamic allocations are managed by the garbage collector, including stack frames and lightweight threads.

There are over 60 different types of closures. Here is a summary of the most important ones:

Function closures represent Haskell functions. When entered, functions assume that all their arguments are present at the top of the stack. This is known as the *eval/apply* evaluation model [15].

The payload of a function closure carries pointers to the free variables of the function's body.

Thunks represent unevaluated expressions. When entered, the corresponding expression is evaluated and the closure is replaced with an indirection to the resulting value. This ensures that thunks are not evaluated multiple times, as subsequent attempts at evaluation will instead enter the indirection which will simply return the existing value.

Indirections are proxies to other closures. Their payload is simply a single pointer to the target object. To reduce the overhead of sharing, indirections are removed by the garbage collector and never outlive the youngest generation.

Black holes are thunks under evaluation, with a layout identical to that of indirections. Since thunks are shared across threads, a thread entering a black hole blocks until it is overwritten with an indirection to the evaluated object.

Data constructors carry their arguments (fields) as payload, ordered such that pointers come first. Their entry code returns immediately to the

topmost stack frame (a constructor itself is always evaluated, although its arguments may not be).

Thread state objects represent lightweight Haskell threads, including their stacks. Since a TSO is simply a closure, it is managed by the garbage collector, just like any other heap object. The garbage collector sends exceptions to blocked threads which become unreachable.

1.1.2 Strictness features

GHC is not only used for production ready Haskell, but also serves as an incubator of new language features – including those directly related to managing the amount of laziness in a program. These allow programmers to aid the compiler in optimising by avoiding unnecessary non-strictness where its static analysis does not suffice.

A simple and robust method of preventing undesired laziness is the language extension `BangPatterns`, which introduces a new pattern syntax `!pat` for forcing an expression to WHNF before pattern-matching it against `pat`. For short functions and clear algorithms which do not benefit from pervasive laziness it is often very easy to simply annotate certain patterns in the program with exclamation marks and observe a reduction in memory consumption.

The language extension shares the exclamation mark syntax with the Haskell 2010 strictness flags feature [16]. While `BangPatterns` add optional strictness to pattern matching, strictness flags do the same for data types. Unfortunately, proper use of this flexibility hinges on the programmer's knowledge of how is the particular piece of code going to be used. While it is good practice to request the early evaluation of values which will have to be forced anyway, sprinkling strictness annotations throughout library code in attempts to prevent space leaks may lead to the unintentional sacrifice of the benefits of laziness, even preventing some usage patterns in subtle ways. Additionally, since these strict evaluation facilities only force thunks to WHNF, the evaluated objects may still retain large delayed expressions. The ability to excise thunks from a Haskell value completely was the core motivation for the development of the `deepseq` library.

The lack of programmer insight into how a piece of code is used in a program and what strictness properties it has is a major developer experience issue [17, 18, 19, 20, 21]. Some of the discussed debugging tools help ameliorate the problem, but GHC itself includes features especially suited to doing so. The compiler supports two profiling modes, cost-centre profiling and "ticky-ticky" profiling, which the GHC User's Guide dedicates a chapter to [22]. While the "ticky-ticky" mode is only of interest to GHC developers, the cost-centre profiling functionality is an easy-to-use tool for understanding the time and space behaviour of Haskell programs. All it requires of the programmer is a recompilation of the modules of interest with a few specific compiler options.

Cost-centre profiling assigns the so-called "cost-centres" to certain sections of code. The RTS records any time spent and allocations performed during the evaluation of code associated with a cost-centre. These recordings are summarised by a time and allocation profiling report, which the profiled program generates. The report indicates the time and space requirements of each cost centre in proportion to the entire program. GHC is able to introduce cost centres automatically by adding them to all non-inlined bindings, but the user also has the option to annotate terms with a pragma to fine-tune the placement of cost centres.

GHC's implementation of profiling can shed some light on the use of call-by-need in a Haskell program. The compiler can also provide certain deeper insights about the program's strictness, although it presents them in a substantially less user-friendly manner. In particular, GHC can output the translation of surface syntax to its internal language, Core. Being a fairly small λ calculus, Core has a clearer semantics including a strict pattern-matching operator `case e of arms...`, which indicates obviously strict subexpressions. Furthermore, the Core output features *demand signatures*, inferred by GHC's demand analysis [3], which classify binders depending on how strict they are in their arguments and to what extent do they use the components of arguments of product types. The results of demand analysis are crucial for subsequent optimisation. Understanding the demand signatures of a program can equip the programmer with the information necessary to determine which patterns would most benefit from the `BangPatterns` extension, which data types could be annotated with strictness flags, and which parts of the program should be refactored in other ways in order to improve the native code generated by the compiler.

The GHC-provided tooling outlined above – particularly the option to dump Core code during compilation and analyse demand signatures – is rather obscure. It is reasonable to expect the average Haskell programmer to only reach for the profiling tools in a time of dire need, when writing high-performance code or dealing with unacceptable space leaks. It is further reasonable not to expect the average Haskell programmer to know the internals of the compiler well enough to ask it for the Core representation of their program, or indeed to be aware at all of the existence of demand signatures, which are only described in the GHC Commentary[2]. Perhaps it would be interesting to include the strictness information inferred by the compiler in interfaces programmers often interact with, such as the various widgets provided by the Haskell Language Server (HLS) [23], but to our knowledge no such tool exists at the time of writing.

In theory, the Glasgow Haskell Compiler's optimisations are advanced

[2]The commentary is intended for GHC developers and is hosted on a GitLab instance (online), unlike the User's Guide which is bundled with the GHC distribution and revised for every release.

enough to compile the majority of Haskell code fairly efficiently, without space leaks or allocation slow-downs, while enabling the greater flexibility, code reuse, and abstraction of a non-strict language. However, inefficiencies introduced to support unnecessary laziness which are small enough not to cause substantial problems could hide in the compiled program. It is a part of the motivation behind this thesis to lay the groundwork necessary for their detection.

1.2 Existing tools

Apart from functionality implemented in the compiler itself, the Haskell environment includes a number of practical solutions to help with debugging. A few of these deal specifically with the issues with laziness that Haskell programmers have to face.

Hoed

Hoed [24] is a tracer and debugger for Haskell. Unlike the built-in debugger of GHCi, Hoed is implemented as a regular Haskell library. Users of Hoed manually annotate functions of interest to make the tracer capture relevant information during execution. The annotations are simply calls to the provided debugging function `observe` with a signature similar to that of the `trace` function from the `Debug.Trace` module of Haskell's standard library. Both `trace` and `observe` circumvent the guarantees of the type system and are in fact impure. `observe` has type `Observable a => Text -> a -> a`, its `Text` argument has to equal the name of the function being annotated. The `Observable` constraint on `a` is used by Hoed internally, the typeclass has a default implementation. The resulting trace of the debugging session is exposed via a web-based interface, to which the users connect with a regular web browser. Hoed's traces include information about which functions have been called during the execution of the annotated program and what were their arguments. It only collects information about annotated functions.

Hoed features several tools to help users analyse problems with their code and find the culprits of test failures. One of these is *algorithmic debugging*, an interactive trace browser which uses an algorithm similar to binary search to locate the deepest incorrect function in the recorded call tree. It does so by asking the user questions about whether certain evaluations were correct, working its way gradually deeper into the tree. The "algorithmic debugger" ultimately reports the faults it located.

While Hoed's approach to debugging is certainly interesting and quite far removed from the concept of debuggers in other languages, it lacks any kind of awareness of the low-level details of non-strictness. Hoed is thus intended for use with property testers like QuickCheck [25], and not as a tool for the

identification and resolution of language implementation -dependent issues, such as memory leaks.

`nothunks`

`nothunks` is a recently released Haskell package which helps in writing thunk-free code. It defines a new typeclass, `NoThunks`, along with instances for common Haskell types. Any type with a `NoThunks` instance can be inspected for unexpected thunks. The library also implements a number of alternatives to common functions from the prelude. These reimplementations check for unexpected thunks introduced during execution, throwing an exception whenever a thunk is detected.

The exceptions of `nothunks` contain helpful information about the context of the thunk which the library function detected, guiding the programmer in locating the unexpectedly lazy code or data structure. The library also allows various relaxations to the strictness of its inspection policy, such as the `OnlyCheckWhnf` and `AllowThunk` `newtype`s. Thanks to `GHC.Generics`[26], `nothunks` also offers the convenient `deriving (Generic, NoThunks)` syntax to add instances of the necessary typeclasses for custom data structures automatically.

The `nothunks` package can greatly help fix serious memory leaks caused by thunk accumulation. However, it is intended primarily for the complete removal of thunks from the runtime state of a program, and does not help with careful strictness analysis.

Hat

The Haskell Tracer Hat [27] is a source-level tracer. It works by compiling Haskell source files to annotated – but still textual – Haskell source files. After this source-to-source translation, the user compiles the annotated source code and runs it to produce a Hat trace.

The trace is a rich recording which contains high-level information about each reduction the program performed. Hat comes with a number of utilities for exploring the trace files, including some forms of forward and backward debugging, filtering utilities which show all arguments passed to top-level functions, virtual stack traces, and even an interactive tool for locating errors in a program, similar to one of the features of Hoed.

Hat was initially developed for the `nhc` Haskell compiler [28]. It centred around the idea of using a single, rich trace of a program's execution to support several different kinds of debugging. Despite its advanced features, it did not seem to attract many users [28], possibly due to feature disparities between the supported syntax and new language extensions.

Hat's source-to-source translation makes it portable between different compilers. The project uses the `haskell-src-exts` package to parse the language,

rather than relying *e.g.* on the GHC API. While Hat cannot directly answer questions about the strictness of debugged functions, its approach to rewriting the source language is interesting. Tracing necessarily leads to runtime overhead, but the code produced by Hat is subject to compiler optimisations. Hat therefore does not need to worry about low-level details of the optimiser and how it reorders, splits and combines expressions. The connection between the tracing code and the original source is maintained thanks to the semantics-preserving nature of optimisations.

`htrace`

`htrace` [29] is a simple package which exports a single function: `htrace :: String -> a -> a`. As the name and function signature suggest, this function mirrors the behaviour of the standard `trace`, except that `htrace` hierarchically indents the tracing messages based on the current call depth. It works simply by manipulating a global mutable variable and hiding this fact from the user with `unsafePerformIO`.

Although very simple and oblivious to any laziness implementation details, this approach is still useful for debugging purposes. The indented tracing messages suggest the depth to which various thunks are evaluated at different points of the program's operation.

`ghc-heap-view`

`ghc-heap-view` is a Haskell package which enables advanced introspection of the Haskell heap from within pure Haskell code. It relies on the `ghc-heap` library which comes bundled with GHC.

The library's notable high-level features include a function which attempts to recreate readable Haskell source code from a runtime value, using `let` bindings to express sharing. There are also tree and graph data structures for heap mapping and a high-level algebraic data type for all Haskell closures, complete with their info tables.

Haskell Program Coverage

Haskell Program Coverage[30] is (unsurprisingly) a code coverage tool for Haskell. Unlike the other tools in this section, HPC is not directly related to laziness control and debugging. Similarly to Hat, HPC has a source-to-source mode of operation but additionally offers tight integration with GHC and comes bundled with modern releases of the compiler. It supports all GHC language extensions.

HPC allows easy instrumentation of arbitrarily complex Haskell programs without source annotations. It wraps subexpressions in the program with an unsafe side-effecting function which records its evaluation by mutating

Tool	Source changes	Order of evaluation	Thunks	Memory awareness[a]
Hoed	Required	Recorded	Transparent	None
`nothunks`	Required	Ignored	Detected	Limited
Hat	Unnecessary	Recorded	Transparent	None
`htrace`	Required	Illustrated	Transparent	None
`ghc-heap-view`	Unnecessary[b]	Ignored	Reified	Full

Table 1.1: An overview of existing solutions to thunk discovery and laziness debugging.

a module-wide array of integer counters. The final state of the per-module arrays forms the HPC trace. This architecture is wired into the GHC compiler pipeline in all the major data structures (the surface syntax, Core language, and STG), which makes it both robust and performant. The tool comes bundled with utilities for displaying the original source code with colourful mark-up, highlighting interesting subexpressions based on the information extracted from the trace. Notably, HPC supports traces of the boolean values of pattern guards, which are added to the visualisation.

HPC's feature set can be of tremendous help to the Haskell programmer, especially when combined with tools like QuickCheck [25]. However, its traces are tuned specifically for code coverage and do not contain enough information to be useful for any kind of dynamic strictness analysis. While the HPC traces are sufficiently granular, the subexpression counters lack necessary information about their execution context and timing.

Summary

Table 1.1 summarizes the surveyed tooling. The *Memory awareness* column suggests to what extent is the particular package or program aware of runtime representations. Independence of the structures underlying Haskell values leads to better portability and a clean interaction with regular Haskell code. On the other hand, more low-level approaches such as `ghc-heap-view` give a much clearer view of the runtime state.

Despite Haskell users' considerable interest in avoiding the implicit delaying of computations which the language is notorious for, there are no records of a large-scale study of the use of laziness in practice akin to [31]. The tool with a feature set closest to what is necessary for a comprehensive analysis of the practical use of laziness is likely `ghc-heap-view`, which allows the user to interactively inspect the heap objects and look inside thunks using GHCi. However, the package primarily provides a rich library interface. It does not implement a tracing mode, which would facilitate collection of laziness-relevant information during the execution of entire programs.

Chapter 2

Analysis and design

2.1 Overview

- add a proper problem statement
- summarise possible approaches
- detail GHCi
- detail compiler plugins
- build on `Tickish`

2.2 Approach

The goal of this work is to design and implement a tool suitable for understanding how is laziness used in real-life Haskell programs. To analyse the practical implications of GHC's implementation of non-strict semantics, we have to understand the strictness properties of functions. For example, some arguments may be evaluated if and only if others are. The tool must capture these dependencies and usage patterns, as they may uncover both use cases where laziness is essential and places where it could be safely avoided, even though static analysis cannot determine so.

Dynamically inferring the strictness properties of functions requires a peek under the hood of Haskell's runtime machinery. Typical Haskell code is oblivious to the underlying representation of the values it manipulates, as reification of the heap objects underneath the abstractions would weaken equational reasoning and parametricity.

Once we have the power to inspect the runtime representations of values, we need to use it to determine the strictness of functions. A function `f` is strict in an argument `a` if `a` has to be evaluated whenever `f a` is evaluated.

There is a number of possible approaches to this problem. As discussed in chapter 1, there already exist related projects which we could build on, at various levels of abstraction. At the lowest level, we could modify the RTS and extract information about heap objects there. We could also modify the compiler in various ways, since it already includes support for HPC and profiling, which is similar to the tracing we would like to implement. Another option is to follow the path of Hat, rewriting the textual source code of traced programs. In this work, we explore two design directions: extending GHCi and tracing with compiler plugins.

2.3 Using GHCi

The bytecode compilation pipeline and the interpreter offer a refreshing break from the comparative complexity of GHC's back end. Unlike the RTS linked to a program compiled to object code[[**unclear**]], the interpreter is at the perfect level of abstraction to directly track evaluations of thunks and the stream of control flow in a program.

One of the issues with this approach that is clear from the outset is the range of supported language extensions. The bytecode compiler and the interpreter lack support for unboxed tuples and sums, shrinking the set of programs the tracer would be able to analyse. In our design of a GHCi-based solution, we took inspiration from [31], which modified the virtual machine of the R language.

Before we describe the GHCi tracing modifications, let us take a closer look at how this part of the compiler project works. GHCi is an interactive interface built on GHC's bytecode compilation pipeline and the bytecode interpreter of the RTS. It offers a read-eval-print loop popular in other functional programming languages.

GHCi consists of several key components: the UI, the GHCi library code, the debugger, the bytecode generator, and the bytecode interpreter. The following sections will introduce each of the building blocks from which GHCi is composed, starting with an overview of how they fit together.

The life of an interpreted expression

GHCi can serve either as a REPL interface, processing expressions one by one, or as an alternative compilation and execution environment for entire modules. These two modes can be freely mixed. The backbone of GHCi is a modified GHC pipeline which culminates in bytecode generation, producing a collection of bytecode objects together with high-level information about breakpoints, pointers to allocated string literals, and other data.

Compiled[3] bytecode objects and their metadata form an analogy of the

[3]That is, compiled to bytecode instructions.

compiler's module abstraction called `CompiledByteCode`. GHCi includes a dynamic linker capable of resolving references between bytecode objects as well as between BCOs and object code.

These features are transparent to the user, who manipulates GHCi via its user interface. The UI is implemented separately from the core functionality and communicates with the GHCi library code via message-passing. This separation allows the UI and the library code, which is in charge of interpreted evaluation, to run in different processes. GHCi features a mode of operation called "Remote GHCi," in which the UI and the interpreter communicate over a pipe. Remote GHCi is useful for situations where the capabilities and heap object definitions of the runtime systems of the compiler and the interpreter differ, e.g. when the compiler and GHCi UI were built regularly with optimisations, but the interpreter was built with profiling.

When evaluating an expression, the library code forks a new thread to perform evaluation independently of the interpreter server. This ensures that exceptions raised during evaluation of an expression do not crash GHCi. The server forwards exception handlers appropriately to ensure this is the case. The two threads communicate via *mutable variables*, or `MVar`s. These are concurrency primitives from the `Control.Concurrent.MVar` module which effectively implement concurrent, mutable `Maybe`s [32]. A mutable variable of type `MVar a` contains either no values or a single value of type `a`. It can be safely shared across threads and supports operations `takeMVar` and `putMVar`. The former operation extracts the value stored in an `MVar`, leaving the variable empty if a value is present. If the variable is empty, the operation blocks. The complementary operation `putMVar` blocks on a full variable and fills it with a value as soon as it is empty.

Two `MVar`s play an important role in the design of GHCi, `statusMVar` and `breakMVar`. These variables form a communication channel between the server thread and the thread responsible for the evaluation of an expression, which we will call the `eval` thread.

When the server thread forks into the `eval` thread to begin expression evaluation, it waits on the `statusMVar`. The `eval` thread keeps running, eventually either producing a result, throwing an exception, or hitting a breakpoint. In the former two cases, it simply fills the `statusMVar` with the appropriate information (either the result of evaluation or the exception) and exits. The server thread resumes execution, passing the result from the `eval` thread to the UI.

The case when the `eval` thread hits a breakpoint is more interesting. First, the `eval` thread fills the `statusMVar` to wake the server thread, notifying it of the breakpoint. Then it waits until the server thread fills the `breakMVar`, pausing evaluation. The server thread notifies the UI, passing along an identifier of the breakpoint that the `eval` thread hit. At this point, the UI notifies the user that evaluation paused on a breakpoint. The user can continue to enter expressions into the GHCi prompt, these will be evaluated indepen-

dently by newly forked threads. These can also hit breakpoints and wake the server thread, which notifies the UI again. None of this interferes with the initial `eval` thread, because every forked `eval` thread gets a new pair of status and breakpoint mutable variables. The user may resume execution in the UI, which messages the server thread, which in turn fills the appropriate `breakMVar`, waking the `eval` thread and blocking on `statusMVar` once again.

Bytecode generation

The bytecode facilities of GHC involve a detour from the typical sequence of steps performed to transform Haskell sources all the way to a form suitable for linking or execution. After desugaring, the program is transformed directly into bytecode instructions[4]. Optimisations implemented in the simplifier are not performed. GHCi is intended for interactive evaluation and favours fast, iterative development over runtime performance, making the naive code generation approach a reasonable choice.

Every top-level definition, every scrutinee of a `case` expression, and every right-hand side of a non-trivial `let` expression are compiled to a Byte Code Object (BCO). Such an object contains an array of bytecode instructions together with the data typically associated with a heap object: the arity of the BCO, a bitmap indicating which of its arguments are pointers, the literals it refers to, and pointers to various objects it refers to (symbols, primitive operations, other BCOs, or the object's array of breakpoint information).

The bytecode format comprises 67 instructions in total, 35 of which only exist to provide various ways of pushing values to the stack. The rest of the virtual instruction set consists of a few instructions for heap allocation, various less-than and equality tests, two instructions for invoking the C FFI, an explicit stack check instruction, and others. The wide variety of instructions of a shared or similar purpose, particularly in the case of stack pushes, is the consequence of distinguishing between the representation of their arguments. These can be pointers subject to garbage collection, word-sized integers, 64bit integers, floating point and double precision numbers, etc.

There is one particular instruction that catches the eye: `BRK_FUN`. The bytecode generator places `BRK_FUN` instructions at the very beginning of every bytecode object. These instructions correspond to breakpoints, though they are only relevant when the user has "placed" a breakpoint at a position in the source code. Alas, "placing" breakpoints is something of an illusion, the instructions are pervasive and every breakpoint has a numeric identifier assigned at compilation time. The introduction of a new breakpoint in the GHCi UI simply sets a flag in a breakpoint bitmap, enabling the corresponding breakpoint instruction from the perspective of the interpreter.

[4] As previously mentioned, this approach will soon be replaced by a new bytecode pipeline which follows the usual compilation process all the way to STG[13].

The bytecode interpreter

The interpreter which GHCi relies on is a part of the RTS. Its primary workhorse is the `interpretBCO` function which handles closure evaluation, unboxed returns, function application, and interpretation of bytecode instructions. For tasks it is unable to deal with, such as application of machine-code functions, it returns to the scheduler.

The interpreter looks at the top of the stack to decide what to do. If it finds a closure, it inspects its type. Most closures are evaluated by *entering*, that is, execution jumps to their entry code. For some types of closures (such as indirections), the interpreter includes shortcuts to avoid the overhead of returning from the interpretation loop to the scheduler and entering the closure. To handle other types of closures, the interpreter returns to the scheduler, setting a field in the TSO which indicates that execution should proceed with machine code evaluation when the thread is woken again.

If the stack is set up for a BCO application with a `RET_BCO` closure below a bytecode object with its arguments, the interpreter executes the BCO instructions. Interpretation of byte code works simply by case analysis on the current instruction.

The debugger

A notable feature of GHCi is its debugger, which allows the programmer to place breakpoints on certain expressions in their code. The interpreter then pauses execution when it is about to evaluate an expression marked by a breakpoint.

Due to laziness, the order in which breakpoints are hit depends on the order in which their respective thunks are forced to WHNF, not directly on the order in which functions are called. Breakpoints thus equip the Haskell programmer with a powerful tool for debugging order of evaluation issues caused by the language's non-strict semantics.

Internally, breakpoints rely on a special bytecode instruction called `BRK_FUN`. Upon encountering this instruction, the interpreter first checks whether it is already returning from a breakpoint (via a flag in the TSO). If it is not returning from a breakpoint and the associated breakpoint is enabled, the interpreter pauses execution at this point.

Pausing on a breakpoint is quite an involved action. The interpreter prepares to call a "breakpoint IO action," a Haskell function invoked to resume GHCi's server thread by filling the shared mutable variable. A pointer to this function is kept in a global variable in the RTS and updated from the Haskell side via FFI. The preparation for an IO action call saves the top stack frame to a new closure, a pointer to which is passed to the IO action. The stack is then set up to call the IO action, and the interpreter returns to the scheduler in order to perform the call.

At no point is the instruction pointer persisted – the progress of evaluation of the current BCO is lost whenever the interpreter stops at a breakpoint. This is acceptable, as the bytecode generator makes sure to only put `BRK_FUN` instructions at the very start of bytecode objects and the TSO flag ensures that a just-visited breakpoint is not stopped at again.

When stopped at a breakpoint, the user can still evaluate expressions at the GHCi prompt. It is thus possible to encounter another breakpoint while stopped at a breakpoint or to hit the same breakpoint multiple times, without ever resuming paused evaluation. The debugger maintains a stack of contexts which makes this possible.

Additionally, the REPL has access to the free variables in the paused expression. The `:print` command lets the user print the values of the free variables without forcing their evaluation, binding thunks to fresh variables. The `:force` command does the same, except that it forces the evaluation of the reference it is applied to. These commands are available even when not stopped at a breakpoint, but become especially useful when stepping through a program.

Tracepoints

Our design for dynamic tracing via GHCi builds on the existing functionality of breakpoints. We introduce *tracepoints*, a simpler variant of breakpoints which uses the communication channel between the server and `eval` threads to pause evaluation at every breakable expression.

The compiler features a generic approach to AST annotations via the `Tickish` type. Values of the `Tickish` data structure are called *ticks* and appear both in the surface syntax and in the Core language. Haskell Program Coverage functionality, profiling, and breakpoints are all implemented as ticks annotating Haskell expressions. The compiler includes scoping rules for ticks, these specify how closely should a particular tick stick to the expression it annotates, which is important for optimisations.

Profiling ticks and breakpoints are added to the Core program during the desugaring phase. Breakpoint ticks capture the free variables of the expression they annotate. When the interpreter pauses at a breakpoint and sets up the stack to call the GHCi breakpoint IO action, it saves the top stack frame[5] into a stack application closure. A pointer to the closure is passed to the IO action and then messaged to the UI. This lets the debugger print out the values of free variables at the site of a breakpoint.

Tracepoints are equivalent to breakpoints, except that they do not need to support nested hits. Our motivation for introducing a breakpoint analogy was to open room for future extensions while preserving the functionality of

[5]That is, the portion of the stack referenced by the just entered BCO. The size of the stack frame is determined from the BCO's bitmap, which indicates the pointerhood of its arguments / free variables.

breakpoints. Tracepoints share the `breakMVar` with breakpoints. Besides the communication channel between the server and the `eval` threads, most of the changes necessary for tracepoints are simple duplications of the existing breakpoint infrastructure. We add a new constructor for `Tickish` which represents a tracepoint, a new bytecode instruction `TRC_FUN`, a new IO action, a new status message for tracepoint hits, and the accompanying code throughout GHCi. Modifications to the module abstractions are not necessary because tracepoints do not need to be enabled or disabled – the interpreter should stop at each and every one.

We modify the UI such that when a tracepoint is hit, all the values of the free variables of the corresponding expressions are logged to a trace file. We also save accompanying timing information and the corresponding location in the source file. Evaluation then resumes without pause.

Summary

The tracepoint approach could be extended and improved in many ways. For example, we could tighten the loop between the interpreter and the reaction to a tracepoint encounter. There is no good reason why it should have to route through the UI. We could also replace the built-in `:print` command with `ghc-heap-view` and perform additional analysis of the runtime values while stopped at a tracepoint.

Nevertheless, our evaluation of the outlined solution points to significant problems. The resulting trace does not contain enough information to make useful inferences about the strictness properties of the code. The problem seems to be that with nested thunks, evaluation does not follow a single logical thread, but rather "skips around" depending on the demands made on the delayed computations. The hard-to-predict, gradual normalisation of values makes it prohibitively difficult to relate different trace entries. The context in which a trace record originated is not guaranteed to be the same for the previous or the following record.

These problems could be overcome by introducing state into the tracing framework. It would be enough to mirror the tree-like hierarchy of subexpressions in the source code with tokens which would relate them at runtime, similarly to how `htrace` uses a simple mutable counter to track nested calls. Unfortunately, the bytecode pipeline and interpreter offer no good point of extension for adding said state. Introducing impurities in the Core program is difficult and error-prone. The bytecode generator and interpreter are both too far removed from the source program and would require sizeable changes to accommodate the sort of functionality we would hope for.

The GHCi approach has other downsides as well. The current bytecode pipeline lacks support for some language extensions and interpretation is much slower than execution of machine code. These issues are not as problematic as the low utility of the tracing results, however.

2.4 Using compiler plugins

To produce useful tracing output, a dynamic tracing framework must capture interesting events during a program's evaluation and relate them to one another. In particular, the evaluation of function arguments must be clearly related to the respective function call to enable reasoning about the strictness of a function on a call-by-call basis. While retaining the order of evaluation is trivial in a call-by-value language, laziness introduces interleaving. This can only be dealt with by the introduction of state into the program (or into the tracing framework) in order to recover the dependencies between function calls and argument evaluations, which are no longer implicit in the order of the trace events.

It is this function-call-specific state that becomes difficult to express without high-level information about the program structure at hand, as is the case with the GHCi approach described in Section 2.3.

Adding state

Fortunately, introduction of function-call-specific state is rather trivial. The source program can simply be rewritten to store the state in local variables. It suffices to keep a unique identifier of the particular function call that the argument evaluation traces can refer to. Such a unique identifier necessarily needs to change with every function call. In clean Haskell code without unsafe features, this is impossible in general, as the language requires the use of the `IO` monad in order to perform side-effecting computations.

Since rewriting functions into a monadic form would be a difficult undertaking, we prefer the way of unsafe features. Integer counters are enough for call identification purposes, so we choose to keep one counter per function. All counters can be stored in a single mutable map indexed by function names. Unsafe functions in the standard prelude can also be used to persist tracing information to a file.

Equipped with a means of introducing benign side-effects into programs for tracing purposes, we are in search of a way of rewriting source code to put these side-effects to use. One plausible approach would be direct source code rewriting, akin to Hat. As described in Section 1.2, source-to-source transformations have the benefit of generality, but also the downside of additional complexity in both the rewriting process itself and the build process of the program, which the user of our tool would have to deal with. Furthermore, true implementation agnosticism of the tracing framework would require compiler-independent support for inspection of the Haskell heap, for which no solution seems to exist at the time of writing. A less general but more ergonomic way of rewriting source code is via GHC's *source plugins*, which hook directly into the compiler pipeline and can operate on the surface-level syntax at different stages.

Source plugins

Source plugins [14] are a relatively recently introduced feature of GHC. Compiler source plugins are Haskell packages which invoke the GHC API to hook into the compiler pipeline and modify the compiled program at various stages of the front end. Unlike Core plugins [33], which operate on the internal language, source plugins deal with the entirety of Haskell's surface syntax.

Rather than parsing, transforming, and serialising the source code separately to the compilation step, we can design a plugin that performs the required source transformations in the compiler pipeline directly. We introduce two tracing functions, `traceEntry` and `traceArg`. We then rewrite the source program to call `traceEntry` every time a function in the program is invoked and we thread every reference to a function's argument through `traceArg`. This introduces the opportunity to inspect the runtime representations of the arguments passed to a function when the result of the function is under scrutiny.

We can determine some of the strictness properties of a transformed function from the calls it makes to the tracing utilities. If we record a call to a (transformed) top-level function `f :: Int -> Int -> Int` defined as `f x y = ...` via `traceEntry` but no calls to `traceArg`, the function makes no use of any of its arguments, and is therefore non-strict in both of them. Examples of functions of this behaviour include `f x y = 3`, `f x y = undefined`, or `f x y = f x y`. Note that the latter example references the arguments on the RHS, but these references are never evaluated. If a call to `f` is followed by a call to `traceArg` for the `x` argument, but the program terminates and no calls to `traceArg` for the `y` argument occur, we say that `f` is strict in `x` and *potentially lazy* in `y`. `f` could be lazy in `y`, but it could also conditionally require `y` to be evaluated based on the value of `x`. The property of a multi-argument function being strict in one argument if another argument matches a predicate (and being non-strict in that argument otherwise) is what makes the interpretation of traces of nested functions tricky.

More interesting cases arise with types which can contain thunks themselves – their Weak Head Normal Form differs from their normal form. Consider `g :: Int -> [Int]` (defined as `g x = ...`). We could observe entries of `g` to be always followed by evaluations of `x`. This does not mean that `g` is necessarily strict in `x`, because the evaluations we have observed may simply happen to be immediately followed by evaluations of `g`'s return value beyond WHNF. This is the case in Listing 2.

```
g :: Int -> [Int]
g x = [x]

main = print . g $ 2 + 2
```

Listing 2: Deep evaluation of an applied lazy function.

However, a decisive analysis of the strictness of a function is not the point of dynamic tracing. We wish to understand the use of laziness in practice. We are interested not only in whether a function used its argument at all, but also in *how many* calls it did so. We would like to learn *e.g.* whether there are lazy functions which are only invoked in a strict manner (akin to the use of `g` in Listing 2), or invoked in a strict manner most of the time[6]. The relationships between the values of arguments in a single function call are instrumental in providing the context necessary for the interpretation of tracing results.

Rewriting the AST

Armed with the impure tracing functions and a plan on how to apply them, we move on to the problem of syntax tree transformation. The `GhcPlugins` module [34] of the GHC API includes the necessary functions to hook into the compiler pipeline. A source plugin can choose to modify the syntax tree at three different stages: right after parsing, between renaming and typechecking, or just after the typechecker has run. These hooks involve different trade-offs. Construction of new (sub)trees becomes more and more difficult further down the pipeline as the internal representation accumulates metadata from the various stages. On the other hand, the available metadata may be necessary for certain tasks and can help plugin authors write more robust implementations. For example, constructing parsed expressions is almost as easy as writing the surface syntax in a source file, using strings as identifiers, but it may result in accidental captures of bindings in scope. Because the renaming phase disambiguates identifiers, constructing renamed ASTs avoids this issue, at the expense of either working with abstract identifiers, or invoking a renaming phase manually.

[[**fix** `\citeauthor`]] As **(author?)**'s introduction to source plugins shows, the costs associated with the construction of syntax trees later in the pipeline are not prohibitive [35]. The GHC API exports high-level functions which let the plugin author take trees from parsed to renamed to typechecked in only a few lines of code. Moreover, the plugin author can use the quasiquoting features [36] of Template Haskell [37] to greatly simplify the construction of expressions. The quasiquoting facilities even manage references to definitions in the scope of the plugin's source code automatically. Common patterns in the expressions created by the plugin can be included as regular top-level definitions in the plugin's module or in a module the plugin depends on and spliced into the syntax tree. With these high-level features in mind, the suitable injection mechanisms for a dynamic tracing source plugin seem to be before and after typechecking. We only discuss the latter approach in the following text, even though a source plugin operating on the renamed AST

[6]For small functions like the one in our example, inlining will take care of eliminating unnecessary laziness. For larger functions, inlining may not help, or the strictness analysis may be too conservative to eliminate unnecessary overhead.

would likely be very similar. Note that the API makes no hard distinction between the different approaches to pipeline extensions. Indeed, a source plugin simply provides a value of the `Plugin` data type, overriding the appropriate fields of a default plugin implementation with monadic functions. A source plugin could run custom code after each of the front end stages.

The actual process of rewriting the right-hand sides of function definitions involves the data types for the surface syntax of Haskell, which has hundreds of constructs [6, Key Design Choices]. The general task of transforming hierarchies of deeply nested data types has many innovative Haskell solutions, including optics and generic programming. While we could use profunctor optics or novel generic approaches, we leverage a fairly simple, if a bit dated, generic programming technique via the Scrap Your Boilerplate (SYB) library [38]. SYB's built-in querying and transformation schemes empower the Haskell programmer with means of applying type-specific functions in all appropriately typed fields of a nested data structure. The library is built using powerful generalisations of folding and a number of combinators, making it easy to create new traversal schemes as compositions of existing building blocks.

Chapter 3

Implementation

In this chapter, we confront some of the development-related issues that tend to arise when working with the GHC project, in the hope of easing future endeavours. The description of the problems we encountered in our work relates to features of the compiler which did not fit in earlier chapters. Next, we discuss the implementation details of a compiler source plugin we developed for dynamic tracing and present the results of the implementation.

3.1 Working with GHC

While obtaining and compiling a local copy of GHC source code is unnecessary for compiler plugin development, a programmer inexperienced with the internals of the project may find it helpful to occasionally peek under the hood of its APIs. Obtaining a working copy is also a prerequisite for forking[[**term?**]] the project – we did so when evaluating the GHCi approach.

The GHC codebase is a large and complicated collection of source files written primarily in Haskell and C [6]. The ever-evolving project is supported by a custom build system called Hadrian [39], itself written in Haskell, introduced to replace GNU Make. Additionally, the build tool of the programmer's choice can be combined with a Docker or Nix -assisted set-up, simplifying the installation of other dependencies required for the build process. We do not recommend using Make: Hadrian was developed to address many issues with Make, including its excessive complexity and poor performance. Tools for simplifying library management (particularly Nix and Docker) are compelling choices for projects with larger teams. However, we found it significantly easier to set up a "native" development environment. Both Nix and Docker suffer from substantial disk usage overheads. Our experience in developing a Docker container for GHC 8.10.2 indicates that large portions of the native set-up have to be replicated in the container.

For example, correctly configuring HLS to load the GHC source code is quite an involved task, eased by the `hie-bios` project [40] which ships with

Hadrian (as a part of the GHC source tree). However, compiling GHC inside the container results in `hie-bios` reporting include paths specific to the container's filesystem to the language server, which in turn breaks all of its functionality. One possible solution is installing the HLS inside the Docker container instead and letting the code editor of choice talk to the containerised process. In comparison, a native set-up with `ghcup` is far easier and comes with fewer surprises along the way, which is why we would recommend it for a project similar in scale to ours.

Build process

The first step to working on the project after obtaining the source code is setting up the build system. Since specific releases of GHC require specific versions of it already installed as the project quickly adapts to use new language extensions, the management of GHC versions on a Unix-like system with a system-wide package manager can be difficult. To ease the management of installed versions and enable quick switching between them, there is the `ghcup` tool [41]. `ghcup` lets the GHC developer quickly install and switch between the releases of not only GHC itself, but also Cabal, the Haskell build system and dependency manager, and the Haskell Language Server (HLS), an LSP-compliant language server providing Haskell-specific editor integration features.

The build system bootstraps the self-hosting compiler in several steps. The GHC provided by the system is referred to as the **stage 0** compiler **[[fix the formatting of stages]]**. GHC comes with build scripts which use the **stage 0** compiler to build first the Hadrian build system. Once Hadrian has been built, the user invokes it to build the **stage 1** compiler, which is a GHC linked against the **stage 0** `base` library. The **stage 1** compiler is subsequently used to build the core libraries from scratch. It is then utilised again to build the **stage 2** compiler, which is linked against the freshly built `base`. The **stage 2** compiler constitutes a complete build of GHC from source code. There is an optional follow-up step, where the **stage 2** compiler builds a **stage 3** compiler, which is useful for profiling GHC while building GHC.

The Hadrian build system offers many configuration options defined as a Haskell module called `UserSettings`. Hadrian comes with a template settings file which explains the various options the user can tweak. The main abstraction is a *build flavour*, a collection of build settings which fully define a GHC build. For debugging the RTS, it is important to know about a lower-level abstraction called a *build way*. Libraries and the RTS can be built in multiple *ways*, which configure far-reaching GHC-specific features, such as profiling and debugging symbols. The RTS can be built in threaded and non-threaded ways, the latter variant of RTS runs in a single operating system thread.

After the initial build, the **stage 1** compiler can be *frozen* by passing a flag to the build system on subsequent invocations. This prevents rebuilding

the **stage 1** compiler every time a source file changes, which speeds up the edit-compile-run cycle tremendously.

[[...]]

3.2 Dynamic tracing with plugins

With a Haskell development environment ready, we have all we need to dive into the implementation of a compiler plugin. We take a look at the details first, then take a step back and cover building and applying the plugin.

Our goal is to transform the following code into a variant that produces a recording of laziness-related events during evaluation. Take the naive version of the QuickSort algorithm shown in Listing 3 as an example. We would like to log both calls to this function and evaluations of its arguments.

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

Listing 3: The QuickSort algorithm on linked lists.

A transformed version of `qsort` with dynamic tracing is shown in Listing 4. Every reference to an argument is replaced with a call to the helper function `traceArg`, patterns in function definitions are split into binding and pattern matching, and the right hand sides of functions are wrapped in `let` expressions which introduce `callNumber` variables.

```
qsort xs = let !callNumber = traceEntry "qsort"
           in case traceArg "qsort" "xs" callNumber xs of
             []     -> []
             (a:as) ->
               qsort left
                 ++ [traceArg "qsort" "a" callNumber a]
                 ++ qsort right
               where
               (left, right)
                = (filter
                     (<= traceArg "qsort" "a" callNumber a)
                     (traceArg "qsort" "as" callNumber as),
                   filter
                     (> traceArg "qsort" "a" callNumber a)
                     (traceArg "qsort" "as" callNumber as))
```

Listing 4: The QuickSort algorithm on linked lists, extended with impure tracing calls.

Anatomy of a plugin

Our source plugin consists of four modules.

- `TracingPlugin`, the entry point of execution and the only exposed module of the package,
- `Typechecking`, which contains utilities for typechecking expressions constructed by the plugin,
- `Logging`, which defines the tracing functions that we compile into source programs, and
- `Rewriting`, where the magic happens. [[perhaps unprofessional]]

The `TracingPlugin` module simply defines and exports a `Plugin` derived from the `defaultPlugin` implementation, overriding `typeCheckResultAction`, the function invoked after the typechecking phase. Neglecting command-line arguments, our action has the type `ModSummary -> TcGblEnv -> TcM TcGblEnv`. As the type indicates, it computes within the typechecking monad (`TcM`) with access to information about the current module (`ModSummary`), modifying its typechecking environment (`TcGblEnv`). The action is invoked once for each compiled module. The typechecking environment is a large data structure which describes the top level of a module with 58 fields. Of these, only `tcg_binds :: LHsBinds GhcTc` is interesting to us. The type constructor `LHsBinds` stands roughly for "located Haskell bindings" and represents a collection of all the top-level bindings of a module annotated with their source file locations. Our post-typechecking action simply threads this field through our rewriting function, which also computes in the typechecking monad, and returns the transformed bindings.

The rewriting function, shown in Listing 5, resides in the `Rewriting` module. It initiates a stateful computation which transforms the bindings in a generic manner using the SYB library.

```
rewrite :: LHsBinds GhcTc -> TcM (LHsBinds GhcTc)
rewrite binds = fst <$> (`runStateT` initialState)
                        (everywhereM' trans binds)
```

Listing 5: The top-level rewriting function, a sole export of the `Rewriting` module.

Since the GHC API abstracts over compiler state using (among other types) the `TcM` monad, the generic transformation involving any non-trivial compiler computations needs to be monadic as well. This transformation is implemented by the `trans` function (shown in fig. 6), which additionally carries a context from the roots of the top-level definitions down to their leaves. We combine the stateful traversal with the typechecking monad by way of the `mtl` package, itself inspired by [42], using the `StateT` monad transformer.

```
trans :: Typeable a => a -> StateT WrapperState TcM a
trans = mkM collectFunInfo `extM` wrapRef `extM` incrementCC
```

Listing 6: The generic transformation function.

`trans` is applied in a single, top-down traversal of the ASTs via a SYB scheme derived from `everywhereM`. Ultimately, the function pattern-matches on important structures in the syntax trees of top-level bindings in three different ways:

1. `collectFunInfo` adds information about the current function to the `WrapperState`,
2. `wrapRef` wraps argument references with a tracing function, and
3. `incrementCC` wraps the right-hand side of each function with a `let` binding, introducing a call counter variable into its scope.

Each of these building blocks of the complete transformation operates slightly differently.

`collectFunInfo :: Bind -> StateT WrapperState TcM Bind` pattern-matches on the various sorts of bindings that can appear in an AST and extracts the names of the named ones, saving them to the `WrapperState` context, thus providing the name of the innermost named function to the other transformations.

`incrementCC :: RHS -> StateT WrapperState TcM RHS` pattern-matches on right-hand sides of functions and introduces calls to the tracing function `traceEntry` using Template Haskell (TH). Calling `traceEntry` with a function name increments a global call counter for that function and returns the counter's current value. `incrementCC` has to introduce a new binding in the scope of the right-hand side so that tracing calls on the RHS can refer to the call ID. Since TH cannot lift the Haskell AST types, the binding has to be constructed in two steps.

First we read the `WrapperState` to find out the name of the function we are currently transforming. We construct a TH expression for the application of `traceEntry` to the function name and bind it via a `let` binding which assigns the result to a new call counter variable in the scope of a dummy expression (a proxy to `undefined`). Then we typecheck this expression and run a SYB transformation which replaces the dummy subexpression with the original right-hand side. Care must be taken when replacing a node in the typechecked AST because the typechecker inserts type applications for polymorphic terms such as `undefined`.

Finally, `incrementCC` also finds the `Id` of the call counter variable via a SYB query and saves it in the `WrapperState`.

`wrapRef :: LExpr -> StateT WrapperState TcM LExpr` pattern-matches on references to function arguments in function bodies. Its purpose is to transform every argument reference into a call to `traceArg`.

[[fix \hsCode overflows]]

To identify references to function arguments, `wrapRef` consults the `boundVars :: [Id]` collection. This collection is built independently of the `wrapRef` transformation, since it needs no function-specific information. We rely on the fact that while references to bindings are semantically valid only in local (lexically-scoped) contexts, they have globally unique identifiers. Collecting the identifiers of function arguments is thus a simple task of traversing all the syntactical pattern-matching structures which bind them. We once again leverage SYB to do this without having to pattern-match on the entirety of surface syntax.

When the reference wrapping transformation identifies a function argument, it constructs a partial application of the `traceArg` tracing function and applies the original binding reference to it. The partially-applied `traceArg` is an unsafe identity function which logs information about the argument's runtime representation to a file.

Since the overall rewriting operation proceeds in a top-down manner, the `wrapRef` transformation runs into the issue of producing subexpressions it could recursively match on again. This could be avoided by tagging the transformed expressions somehow. Unfortunately, this is difficult to achieve, because the AST datatypes lack useful typeclass instances for doing so. Crucially, there is no notion of equality on syntax trees and no hashing implementation which would let us store the transformed expressions in a hash set (or at least a set). We work around this limitation by stripping the source location tags from the AST nodes and checking for their presence before invoking `wrapRef`'s rewriting logic, but we are aware of the problems with this approach. However, issues with error reporting are largely mitigated by the fact that the plugin is invoked after the source program passed the typechecking phase.

Implementation of tracing utilities

The tracing functions inserted into the AST by the rewriting logic reside in the `Logging` module. They leverage the unsafe IO features of Haskell, specifically the standard `unsafePerformIO :: IO a -> a` from `System.IO.Unsafe`, to hide the side-effects of tracing from the type system. When invoked, these functions append a row of CSV-encoded data to a *trace file*, a log of interesting events that occurred during the evaluation of a Haskell program, which is suitable for further analysis.

`traceEntry :: String -> Int` marks the evaluation entry point of a function. Taking the function's name, it increments its call counter in the background and returns its new value. The call counters are stored in a global map called `functionEntries :: IORef (Map String Int)`. The `IORef` indirection makes `functionEntries` a mutable variable which can be manipulated in the `IO` monad. The map is explicitly marked with a *{-# NOINLINE #-}* pragma to ensure it is shared between the tracing calls. Since the `IORef` constructor returns a reference in the `IO` monad, we allocate the global variable via `unsafePerformIO`.

The call counter map is empty at first, individual counters are initialised on-demand. The initialisation of a new call counter and the increment of an existing one are both described concisely by the `insertWith` operation on `Map`s, which takes a binary function on values, a key, an initial value, and a map, and either initialises the key to the initial value or updates it by applying the binary function to its current value and the initial one. This operation is applied atomically via `atomicModifyIORef'` to accommodate concurrent updates.

`traceArg :: String -> String -> Int -> a -> a` indicates a reference to a function argument. Partially applying this function to the name of the enclosing function, the name of the referenced argument, and the number of the call to the enclosing function leaves an impure identity, which is applied to the actual argument. `traceArg` leverages the `ghc-heap-view` library to take a peek at the runtime representation of the argument to determine whether it has been evaluated or not.

`logt :: TraceSort -> [String] -> IO ()` persists a tracing message to the trace file. Calls to this function are not introduced during the rewriting process directly, but both `traceEntry` and `traceArg` call it internally. The function can thus stay in the safe realm of the language, as its type indicates. File system operations in Haskell require a value of type `Handle` which the RTS uses to manage IO with file system objects. Allocating a handle corresponds to opening a file. Since that is a potentially expensive operation, we store the handle in another non-inlineable `IORef`, again created globally with `unsafePerformIO`. `logt` then simply reads the `IORef`, appends tracing data to the file, and flushes the handle, to avoid problems with lazy IO and prevent data loss when the program exits.

The types `Bind`, `RHS`, and `LExpr` are aliases for the more verbose structures of the surface syntax.

Using the plugin

Applying a compiler plugin in the pipeline of GHC is straightforward: it suffices to pass a command-line option `-fplugin` set to the name of the module which exports the plugin definition. Multiple plugins can be specified at once by passing more than one `-fplugin` option, the compiler applies them in the order they are defined[7]. To make the module visible to the compiler, it has to be installed in GHC's package database. Installation is simply a matter of executing the Cabal command `cabal install --lib` in the plugin's directory.

[[...]]

Shortcomings

Our implementation work does not include all the transformations required to perform the perfect rewriting as illustrated in Listing 4. Instead, it performs a simplified transformation which neither separates variable binding from destructuring nor untangles nested pattern matches. Due in part to limitations in Template Haskell, `let` bindings of call counters do not include bang patterns. Several styles of pattern definitions are also unsupported. A notable omission is the support for the popular `do` notation, commonly used when working with monads.

The plugin unfortunately suffers from a bug caused by the traversal scheme used in the rewriting transformation. The error stems from the way the call counter variables are referred to in the stateful computation. When the traversal exits a lambda, it should restore the call counter reference to that of the surrounding context. This does not happen because SYB does not offer a way of detecting this step. In effect, the plugin inserts undefined references to code of certain shapes, crashing the compiler during desugaring. Fixing this bug in the scope of SYB is challenging, because the monadic transformation proceeds sequentially through the syntax tree.

Despite these issues, we feel that the design laid out and partially implemented in this work is suitable for dynamic tracing. All of the shortcomings listed above are implementation problems which can be addressed in future work. Handling of `do` notation can be implemented via source-level desugaring, mirroring the work of the desugaring phase of the compiler. This is in fact the technique already implemented by Hat. The traversal issue can similarly be fixed by the choice of another generic programming library better suited for monadic tree operations, or by replacing the generic programming approach with another. The remaining deficiencies have straightforward solutions and could be alleviated with more time.

[7] Still subject primarily to the order of phases in the pipeline, but ordered secondarily according to the succession of command-line arguments to avoid ambiguities.

One simple but necessary improvement to the implementation is changing the representation of names in the trace. The examples and our implementation uses strings, which can lead to name clashes. However, GHC assigns globally unique identifiers to all names. During compilation, the plugin should produce a table associating fully qualified names to their numeric identifiers. The trace can then refer to functions and arguments with integers instead.

It remains to be seen whether the choice of rewriting after the typechecking phase supports the future extensions. It may be easier to perform some transformations without the annotations introduced by the typechecker right after the renaming phase, or even right after parsing.

Conclusion

This thesis aimed to design and implement a scalable dynamic tracing framework for Haskell. Although the implementation does not support all of Haskell syntax, the design allows for great scalability thanks to tight integration with the Glasgow Haskell Compiler. Tracing is added by rewriting the syntax trees of source programs. The traced programs produce recordings of function call and argument evaluations in CSV format, suitable for subsequent analysis.

Future work

Our work is only the beginning of the journey towards understanding the practical implications of laziness. The compiler plugin we developed needs to be finalised and extended before it can start recording data on unmodified real world programs. After incorporating these fixes, it can support large-scale collection of traces, providing the empirical evidence for use in a future study.

Bibliography

[1] Hudak, P.; Hughes, J.; et al. *A history of Haskell: being lazy with class*, chapter Haskell is pure. In [2], 2007, pp. 12–1.

[2] Hudak, P.; Hughes, J.; et al. A history of Haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, 2007, pp. 12–1.

[3] GHC Commentary – demand. Accessed: 2021-05-03 10:57:53. Available from: `https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/demand`

[4] Marlow, S.; et al. Haskell 2010 language report. *Available on: https://www.haskell.org/onlinereport/haskell2010*, 2010.

[5] GHC Team. *GHC User's Guide Documentation*, chapter 10. GHC Language Features. In [43]. Available from: `https://downloads.haskell.org/~ghc/8.10.2/docs/users_guide.pdf`

[6] Marlow, S.; Peyton Jones, S. In *The Architecture of Open Source Applications: Structure, Scale, and a Few More Fearless Hacks*, volume II, chapter The Glasgow Haskell Compiler.

[7] Najd, S.; Jones, S. P. Trees that Grow. *J. UCS*, volume 23, no. 1, 2017: pp. 42–62.

[8] Pickering, M.; Érdi, G.; et al. Pattern synonyms. In *Proceedings of the 9th International Symposium on Haskell*, 2016, pp. 80–91.

[9] Sulzmann, M.; Chakravarty, M. M.; et al. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, 2007, pp. 53–66.

[10] GHC Commentary – Core to Core pipeline. Accessed: 2021-05-05 02:55:00. Available from: https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/core-to-core-pipeline

[11] Peyton Jones, S. L.; Salkild, J. The spineless tagless G-machine. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, 1989, pp. 184–201.

[12] Jones, S. L. P. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine - Version 2.5. *Journal of Functional Programming*, volume 2, 1992: pp. 127–202.

[13] Stegeman, L. Support unboxed tuples and sums in GHCi. Accessed: 2021-05-11 09:44:26. Available from: https://gitlab.haskell.org/ghc/ghc/-/merge_requests/4412

[14] Pickering, M.; Wu, N.; et al. Working with source plugins. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, 2019, pp. 85–97.

[15] Marlow, S.; Jones, S. P. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *ACM SIGPLAN Notices*, volume 39, no. 9, 2004: pp. 4–15.

[16] Marlow, S.; et al. *Haskell 2010 language report*, chapter 4.2.1 Algebraic Datatype Declarations. In [4], 2010.

[17] Sewell, W. Memory profiling in Haskell. Accessed: 2021-05-11 10:00:07. Available from: https://making.pusher.com/memory-profiling-in-haskell/

[18] Yang, E. Z. Anatomy of a thunk leak. Accessed: 2021-05-11 09:53:19. Available from: http://blog.ezyang.com/2011/05/anatomy-of-a-thunk-leak/

[19] de Vries, E. Being lazy without getting bloated. Accessed: 2021-05-11 09:41:27. Available from: https://well-typed.com/blog/2020/09/nothunks/

[20] Kulal, S.; Ganvir, R.; et al. *Space leaks exploration in Haskell*. Dissertation thesis, Indian Institute of Technology Bombay Mumbai 400076, India.

[21] Mitchell, N. Detecting Space Leaks. Accessed: 2021-05-11 09:49:22. Available from: http://neilmitchell.blogspot.com/2015/09/detecting-space-leaks.html

[22] GHC Team. *GHC User's Guide Documentation*, chapter 8. Profiling. In [43]. Available from: https://downloads.haskell.org/~ghc/8.10.2/docs/users_guide.pdf

[23] Successor of ghcide & haskell-ide-engine. One IDE to rule them all. Accessed: 2021-05-03 12:24:18. Available from: https://github.com/haskell/haskell-language-server

[24] GitHub – MaartenFaddegon/Hoed: Hoed – A Lightweight Haskell Tracer and Debugger. Available from: https://github.com/MaartenFaddegon/Hoed

[25] Claessen, K.; Hughes, J. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, volume 46, no. 4, 2011: pp. 53–64.

[26] Universiteit Utrecht; University of Oxford. GHC.Generics – Hackage: The Haskell Package Repository. Accessed: 2021-05-12 16:29:53. Available from: https://hackage.haskell.org/package/base-4.15.0.0/docs/GHC-Generics.html

[27] The Haskell Tracer Hat. Available from: https://archives.haskell.org/projects.haskell.org/hat/

[28] Hudak, P.; Hughes, J.; et al. *A history of Haskell: being lazy with class*, chapter 10.4.2 Debugging via redex trails. In [2], 2007, pp. 12–1.

[29] Kirpichov, E. htrace: Hierarchical tracing for debugging of lazy evaluation. Accessed: 2021-05-12 11:49:26. Available from: https://hackage.haskell.org/package/htrace

[30] Gill, A.; Runciman, C. Haskell program coverage. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, 2007, pp. 1–12.

[31] Goel, A.; Vitek, J. On the Design, Implementation, and Use of Laziness in R. *Proc. ACM Program. Lang.*, volume 3, no. OOPSLA, Oct. 2019, doi: 10.1145/3360579. Available from: https://doi.org/10.1145/3360579

[32] Jones, S. P.; Gordon, A.; et al. Concurrent Haskell. In *POPL*, volume 96, 1996, pp. 295–308.

[33] GHC Team. *GHC User's Guide Documentation*, chapter 12.3 Compiler plugins. In [43]. Available from: https://downloads.haskell.org/~ghc/8.10.2/docs/users_guide.pdf

[34] GhcPlugins. Available from: http://hackage.haskell.org/package/ghc-8.10.2/docs/GhcPlugins.html

[35] Pickering, M. Source Plugins: Four ways to build a typechecked Haskell expression. Available from: https://mpickering.github.io/posts/2018-06-11-source-plugins.html

[36] Mainland, G. Why It's Nice to Be Quoted: Quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, New York, NY, USA: Association for Computing Machinery, 2007, ISBN 9781595936745, p. 73–82, doi:10.1145/1291201.1291211. Available from: https://doi.org/10.1145/1291201.1291211

[37] Sheard, T.; Jones, S. P. Template Meta-Programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, New York, NY, USA: Association for Computing Machinery, 2002, ISBN 1581136056, p. 1–16, doi:10.1145/581690.581691. Available from: https://doi.org/10.1145/581690.581691

[38] Lämmel, R.; Jones, S. P. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, volume 38, no. 3, 2003: pp. 26–37.

[39] Mokhov, A.; Mitchell, N.; et al. Non-recursive make considered harmful: build systems at scale. *ACM SIGPLAN Notices*, volume 51, no. 12, 2016: pp. 170–181.

[40] Pickering, M. GitHub – mpickering/hie-bios. Accessed: 2021-05-09 03:38:39. Available from: https://github.com/mpickering/hie-bios

[41] ghcup – The Haskell (GHC) toolchain installer. Available from: https://www.haskell.org/ghcup/

[42] Jones, M. P. Functional programming with overloading and higher-order polymorphism. In *International School on Advanced Functional Programming*, Springer, 1995, pp. 97–136.

[43] GHC Team. *GHC User's Guide Documentation*. Available from: https://downloads.haskell.org/~ghc/8.10.2/docs/users_guide.pdf

Appendix A

Acronyms

API Application Programming Interface.

AST Abstract Syntax Tree.

BCO Byte Code Object.

CSV Comma-Separated Values.

FFI Foreign Function Interface.

GADT Generalised Algebraic Data Type.

GHC Glasgow Haskell Compiler.

GHCi GHC interpreter.

GNU GNU's Not Unix, a Unix-like operating system.

HLS Haskell Language Server.

HPC Haskell Program Coverage.

IR Intermediate Representation.

LLVM Low-Level Virtual Machine.

LSP Language Server Protocol.

OS Operating System.

REPL Read-Eval-Print Loop.

RHS Right-Hand Side.

RTS Runtime System.

STG Spineless Tagless G-machine.

STM Software Transactional Memory.

SYB Scrap Your Boilerplate.

TH Template Haskell.

TSO Thread State Object.

UI User Interface.

WHNF Weak Head Normal Form.

Appendix B

Contents of enclosed CD

[[figure out what to do about this]]

- `readme.txt` ..... the file with CD contents description
- `exe` ..... the directory with executables
- `src` ..... the directory of source codes
  - `wbdcm` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format