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.

April 25, 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 April 25, 2021 ....

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 Tables

Introduction

Conventional programming languages of all paradigms use – almost equivocally – eager evaluation strategies. Non-strict semantics have far-reaching implications on the design of a language[1] and pose many implementation challenges.

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.

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. Its most popular compiler, 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.

Although necessary for non-strictness as required by the Haskell spec, laziness leads to many issues with runtime behaviour of Haskell programs. 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.

Among the surveyed approaches to the inspection and management of thunks were the following:

- Hoed
- `nothunks`
- Hat
- `htrace`
- `ghc-heap-view`

1.1 Existing tools

Several tools related to tracing are available.

Hoed

Hoed[2] is a tracer and a 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, hiding unsafe IO. `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. This is perhaps due to the fact that it was implemented at a time when it was generally believed that competing implementations of Haskell will emerge. Hoed is thus intended for use with property testers like QuickCheck, 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, `nothunks` also offers the convenient `deriving (Generic, NoThunks)` syntax to add instances of the necessary typeclasses for custom data structures automatically.

Hat

The Haskell Tracer Hat[3] 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.

The architectural decisions of Hat reflect the environment it originated in, which unfortunately differs substantially from the current status quo. Its source-to-source model of operation makes it compatible with various Haskell compilers,

The Glasgow Haskell Compiler is the most widely used Haskell compiler with many language extensions beyond Haskell 2010. In 2009, GHC became the official compiler of the Haskell Platform[4], further cementing its monopoly as the primary implementation of the language.

Hat uses the `haskell-src-exts` package to parse the source language.

`htrace`

`htrace` 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 when displaying the tracing messages, `htrace` shows them hierarchically indented 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 makes advanced introspection of the Haskell heap a possibility 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.

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 [5]. 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.

1.1.1 Summary

Table 1.1 summarizes the surveyed tooling.

Tool	Source changes	Order of evaluation	Thunks	Memory awareness[1]
Hoed	Required	Recorded	Transparent	None
`nothunks`	Required	Ignored	Detected	Limited
Hat	Unnecessary	Recorded	Transparent	None
`htrace`	Required	Illustrated	Transparent	None
`ghc-heap-view`	Unnecessary[2]	Ignored	Reified	Full

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

1.2 Existing profilers

1.2.1 Haskell Program Coverage

Haskell Program Coverage[6] is (unsurprisingly) a code coverage tool for Haskell. 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 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[7]. 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.

1.3 The Glasgow Haskell Compiler

...

1.3.1 Compiler plugins

...

Chapter 2

Analysis and design

...

2.1 Analysis

- the core question: is laziness worth the hassle?
- original approach by modifying GHCi
- reuse the implementation of breakpoints, effectively implementing fairly clean hooks (bearing a slight resemblance to how the R tracer works)
- build on `Tickish`

2.1.1 The problem

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[8]. 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: if GHC is unable to prove the strictness of a function in an argument by static strictness analysis, the function will remain lazy, often leading to pathological memory behaviour at runtime.

A simple and popular method of dealing with undesired laziness is the language extension `BangPatterns`. `BangPatterns` introduce a new syntax for forcing an expression to WHNF when pattern-matching on it.

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.

2.1.2 Approach

- core question: how is laziness used in practice?
- to understand that, we have to find out whether functions are strict and to what extent, discover potential strictness dependencies between their arguments, etc
- to do that, we need to determine whether an argument has been evaluated during function application
- to do *that*, we have to look at runtime values
- to put the observations of runtime representations into context, we have to somehow keep track of function calls

The goal of this work is to design and implement a tool suitable for understanding how is laziness utilised in real-life Haskell programs. To analyse the practical implications of GHC's implementation of non-strictness, 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 thunks would weaken equational reasoning and parametricity.

There are two general approaches one could take to capture the information about runtime structures over the execution of a Haskell program: modify the program, or modify the compiler. The former would involve rewriting the source code, while the latter

The purpose of the project thus dictates use of features which violate some of the abstractions provided by the language.

Understanding the use of laziness at runtime requires insight about runtime structures that are otherwise transparent to the Haskell programmer. A key feature of the language is its support for equational reasoning, which would be broken if thunks were directly observable. To determine whether certain values have been evaluated or not, we need to observe state that is typically hidden from a Haskell program.

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.

2.1.3 The GHCi approach

Taking inspiration from [5], the original implementation plan was to work with GHC interpreter (GHCi). The bytecode compiler and interpreter lack support for certain GHC language extensions, namely unboxed tuples and sums, but the supported subset of the language was considered large enough to contain interesting examples. The relative simplicity of the bytecode compilation pipeline and the fairly straightforward evaluator were considered to provide a foundation amenable to low-level tweaks deemed necessary for the extraction of crucial tracing information.

The framework of the interpreter would ease the implementation of certain features. GHCi already implements breakpoints, which pause the execution of a Haskell program running in a separate thread and pass messages to the controlling Haskell thread.

2.1.3.1 GHCi: a primer

GHCi is an interactive interface built on GHC's bytecode compilation pipeline and the bytecode interpreter of the RTS. GHCi offers a read-eval-print loop popular in other programming languages.

GHCi consists of several key components: the GHCi UI, the GHCi debugger, the bytecode generator, and the bytecode interpreter. The former two are a part of the front end of GHC, while the bytecode-centric parts fit into the back end of the compiler pipeline and the RTS, respectively.

The following sections will introduce each of the building blocks from which GHCi is composed, starting with an overview of how they fit together.

2.1.3.2 The life of an interpreted expression

The user's expression entered at the REPL's prompt is fed through a modified GHC pipeline, as GHCi expects Haskell expressions, not top-level definitions. This modified pipeline 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.

The bytecode objects form, together with other information, a compiled module. That module is loaded by the compiler instance and

When evaluating an expression, the UI forks a new thread to perform evaluation independently of the interface. This ensures that exceptions raised during evaluation of an expression don't crash GHCi. The UI 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[9]. 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 converse 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 UI thread and the thread responsible for the evaluation of an expression.

They are greeted with the interpreter's UI and can begin writing Haskell expressions directly or first invoking various GHCi commands to load modules, print types, kinds, and documentation, browse the contents of modules and perform other tasks.

2.1.3.3 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[3]. 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 bytecode format comprises 67 instructions

2.1.3.4 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.

Interpretation works simply by case analysis on the current instruction.

2.1.3.5 The debugger

A notable feature of the tool is the GHCi 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.

[3]Note that this approach will soon be replaced by a new bytecode pipeline which follows the usual compilation process all the way to STG.

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 an "IO action," which is a Haskell function invoked to resume GHCi's UI thread by filling the shared mutable variable. This preparation 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.

2.1.3.6 The user interface

...

2.1.4 The compiler plugin approach

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 was the case with The GHCi approach.

Adding state

Fortunately, function-call-specific state can be easily introduced into the source program, simply as 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, which associates

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 1.1, 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[10] 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, 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`, into the current module. 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 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 non-strict* in `y`. `f` could be non-strict 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.

Rewriting the AST

Armed with the necessary tracing functions and a plan on how to apply them, we move on to the problem of syntax tree transformation. The `GhcPlugins` module[11] 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.

As **(author?)**'s introduction to source plugins shows, the costs associated with the construction of syntax trees later in the pipeline are not prohibitive[12]. 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[13] features of Template Haskell[14] 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 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 frontend 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[15, 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[16]. 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.

Anatomy of a plugin

The structure of our source plugin, which hooks into the pipeline after the typechecking phase, is as follows. The field of interest in the `Plugin` type is `typeCheckResultAction`, Neglecting command-line arguments, it has the type `ModSummary -> TcGblEnv -> TcM TcGblEnv`.

Chapter 3

Realisation

...

3.1 Development environment

...

3.2 Building GHC

The GHC codebase is a large and complicated collection of source files written in a number of programming languages, primarily Haskell and C[15]. The ever-evolving project is supported by a custom build system called Hadrian, itself written in Haskell, which bootstraps the self-hosting compiler in several steps. To build GHC, an appropriate version of GHC has to be installed already. The installed compiler is referred to as the **stage 0** compiler . Hadrian uses the **stage 0** compiler to build first the Hadrian build system and with it the **stage 1** compiler, which is a freshly built 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 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 **stage 0** compilers 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, the `ghcup` tool[17] has been developed. `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.

There are several supported approaches to building GHC, as the compiler previously used a build system based on GNU Make (before switching to Hadrian) and the old Make build system is still being phased out. 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.

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.

...

Conclusion

...

Bibliography

[1] Hudak, P.; Hughes, J.; et al. *A history of Haskell: being lazy with class*, chapter Haskell is pure. In [8], 2007, pp. 12–1.

[2] GitHub - MaartenFaddegon/Hoed: Hoed - A Lightweight Haskell Tracer and Debugger. Available from: `https://github.com/MaartenFaddegon/Hoed`

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

[4] Haskell Platform. Available from: `https://www.haskell.org/platform/`

[5] 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`

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

[7] 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.

[8] 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.

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

[10] 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.

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

[12] 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`

[13] 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`

[14] 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`

[15] 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.

[16] 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.

[17] ghcup - The Haskell (GHC) toolchain installer. Available from: `https://www.haskell.org/ghcup/`

[18] 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.

Appendix **A**

Acronyms

API Application Programming Interface.

AST Abstract Syntax Tree.

BCO Byte Code Object.

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.

LSP Language Server Protocol.

REPL Read-Eval-Print Loop.

RHS Right-Hand Side.

RTS GHC RunTime System.

STG Spineless Tagless G-machine, an abstract machine based on graph reduction[18]. GHC compiles the Core language to STG instructions, machine code is generated from the STG representation..

SYB Scrap Your Boilerplate.

TSO Thread State Object.

WHNF Weak Head Normal Form.

Appendix **B**

Contents of enclosed CD

```
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
```