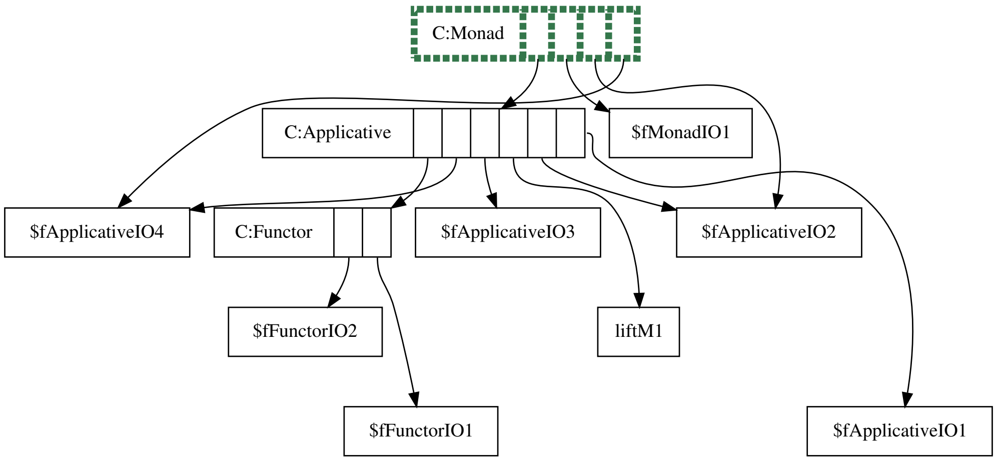


Front-end of a Debugger for Compiled Programs in Haskell

Visualising the evaluation of compiled Haskell programs

Bachelor's thesis in Computer science and engineering

Andreas Olsson
Carl Bergman
Brage Salhus Bunk
Elias Johansson
Krešimir Popović

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Bachelor's thesis 2022

Front-end of a Debugger for Compiled Programs in Haskell

Visualising the evaluation of compiled Haskell programs

Andreas Olsson
Carl Bergman
Brage Salhus Bunk
Elias Johansson
Krešimir Popović

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Front-end of a Debugger for Compiled Programs in Haskell
Visualising the evaluation of compiled Haskell programs
Andreas Olsson, Carl Bergman, Brage Salhus Bunk, Elias Johansson, Krešimir Popović

Supervisor: Krasimir Angelov, Department of Computer Science and Engineering
Examiner: Alex Gerdes, Department of Computer Science and Engineering

Bachelor's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: An example heap visualisation generated by the developed program.

Typeset in LaTeX
Gothenburg, Sweden 2022

Front-end of a Debugger for Compiled Programs in Haskell
Visualising the evaluation of compiled Haskell programs
Andreas Olsson, Carl Bergman, Brage Salhus Bunk, Elias Johansson, Krešimir Popović

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

There are debugging tools available for Haskell programs today, but despite providing help for developers, these tools are not widely used in the Haskell community. Current debugging tools for Haskell have different kinds of limitations. This thesis will discuss the implementation of a new debugger with a graphical interface. This program makes it possible for a user to debug a compiled Haskell program. It offers features such as step by step evaluation, setting breakpoints, and examining the heap through visualisation. We will also briefly explain the basic concepts of the Haskell programming language, parts of the Glasgow Haskell Compiler's run-time environment, and more thoroughly discuss the visualisation of the evaluation process for Haskell programs. The finished program is a user-friendly debugger that works for compiled Haskell code and can act as a useful tool for learning, profiling, and debugging purposes for both beginners and experienced Haskell developers.

Keywords: Haskell, debugger, debug, heap, lazy evaluation, functional programming, GHC

Sammandrag

Det finns tillgängliga felsökningsverktyg för Haskell-program idag, men trots att de kan hjälpa utvecklare används inte dessa verktyg i stor utsträckning bland Haskelutvecklare. De existerande felsökningsverktygen för Haskell har olika typer av begränsningar. Detta examensarbetet kommer att avhandla implementeringen av en ny debugger med ett grafiskt gränssnitt. Detta program gör det möjligt för användaren att felsöka ett kompilerat Haskell program. Programmet inkluderar funktionalitet som steg för steg evaluering, sätta brytpunkter och undersöka heapen genom att visualisera den. Vi kommer också kortfattat att förklara de grundläggande begreppen i Haskellprogrammeringsspråket, delar av Glasgow Haskell Compilerns exekveringsmiljö, och mer ingående diskutera visualiseringen av evalueringsprocessen för Haskell-program. Det färdiga programmet är en användarvänlig debugger som kan fungera som ett användbart verktyg för inlärning, profilering och felsökning av kompilerad Haskellkod för både nybörjare och erfarna Haskellutvecklare.

Nyckelord: Haskell, debugger, felsökningsverktyg, felsökning, heap, lat beräkning, funktionell programmering, GHC

Acknowledgements

We would like to send a big thanks to our supervisor, Krasimir Angelov, for answering our questions, providing help throughout the development, and creating the HDB debugging library.
Additionally, we are thankful for Dennis Felsing's work with ghc-vis, an inspiration for our own visualisation of Haskell's evaluation.

Carl Bergman, Brage Salhus Bunk, Elias Johansson, Krešimir Popović, Andreas Olsson, Gothenburg, June 2022

Contents

Glossary

Binary An executable.

Breakpoint A breakpoint is a specific position in the source code at which a debugged process should temporarily stop.

Bug A bug is a computer science term of an unwanted and unintended consequence of faulty source code.

Compile To compile a program means to generate an executable from source code.

Compiler A compiler is a program that can generate executables from source code.

Debug To debug something means to analyse the source code or the execution of a program at an attempt to remove bugs.

Executable An executable file, or just executable, is a file containing a program which can be loaded and executed by the operating system.

Function A function is a procedure that can use zero or more inputs to generate an output.

Functional programming A programming paradigm where functions take a central role.

Git Git is a version control tool that is used to collaborate on a shared source code base.

GTK A framework to create GUI applications.

GUI A GUI is a graphical user interface for an application.

Haskell Haskell is a functional programming language.

Heap A heap is a pool of memory used for dynamic memory allocation during a program's execution.

Interpreter An "interpreter" is a program capable of executing a program without compiling the source code to native machine code.

Library A library is a collection of source code that can be utilised in other programs through a predefined interface.

Profiling A process where the performance of a program, for example speed or resource usage, is analysed.

Scrum Scrum is a way of structuring the development process that is commonly used in software development.

Stack For the Glasgow Haskell Compiler the stack is a part of the heap. It stores the address of the previous function call so that the current can return upon finished execution.

State A program's state represents its current execution status.

State variable A variable in a program that changes depending on a program's state.

List of Figures

List of Tables

List of Listings

1
Introduction

Ever since programming came into existence there have been “bugs”. Bugs are errors in programming which change the desired output of a program, sometimes with serious consequences [1].

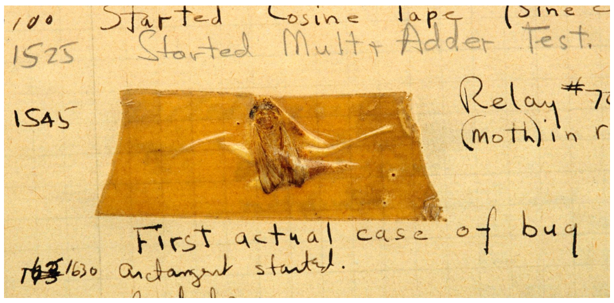


Figure 1.1: “In 1947, when technicians building the Mark II computer at Harvard discovered a moth in one of the relays, they saved it as the first actual case of a bug being found” [2]. *Image courtesy of Smithsonian Institution.*

We have taken on the project of constructing a graphical interface for a debugging library that operates on compiled programs in the functional programming language Haskell.

In Haskell, an expression can be partially evaluated during the execution. This means that the execution of the program will be harder to follow since a line of code can be evaluated partially at first and then fully evaluated at a later stage. This is different from imperative languages where the program operates and evaluates line by line. Therefore it is important to show the evaluation in a way that a user would understand and to avoid potential confusion that might arise due to existing experiences.

We visualise the Glasgow Haskell Compiler’s (GHC) heap and stack as a directed graph in a similar way to the heap visualisation tool ghc-vis [3]. However, ghc-vis only supports the GHC interactive environment (GHCi), which means that it does not support compiled programs.

To debug a compiled program a small debugging library used. This library is called HDB [4] and forms the complete “backend” of our debugger. HDB can parse files using the DWARF [5] format, which is a standard for storing debug information, and return the information in different forms to a user. HDB and DWARF are covered more in-depth in Chapter 2.

According to Angelov [6] there are currently no viable alternatives to debug compiled Haskell programs that work without changing the source code. Often compiled programs have higher performance than interpreted programs as they execute native machine code. When code compiles the compiler may modify the machine code so that it does not correlate exactly to the source code. For example, when debugging a performance-related issue, an interpreted version of the program may not reveal the problem. This project fulfils the niche of a debugger that works for compiled standard Haskell source code.

The finished program has the following features:

- A visualisation of the objects on the heap and on the stack as a directed graph.
- The depth of the graph can be limited.
- Automatic centring of the currently evaluating node on the screen.
- Functionality to control the graph by using the mouse to zoom and drag the image.
- Basic debug functionality such as adding and removing breakpoints, and stepping through the evaluation.
- A window where the source code files can be displayed and highlighting of currently evaluating code snippets.
- The ability to display the debugged program's output.
- A help dialog that informs the user about how to use the program.

1.1 Purpose

The purpose of this project is to develop a graphical debugger that uses HDB to work directly with compiled Haskell programs. In addition, our tool should visualise the heap and thereby act as a teaching tool as well as a debugger. It should also be possible to use for profiling. Memory allocation in Haskell can be hard to predict as it is allocated implicitly for unevaluated expressions. Being able to observe the evaluation directly makes the allocation visible, which helps the developer to improve the program.

The project is aimed at those who are familiar with working in the Haskell language and require a debugger that can debug compiled programs. Although the intention is to be a teaching tool as well, it requires the user to have knowledge of Haskell and its evaluation process. Without this knowledge, it will be difficult for a user to follow along with the graph representation of evaluation. But we believe that for an audience with some Haskell experience the debugger could be a useful tool to improve their understanding of Haskell evaluation, and of course, for debugging compiled programs.

1.2 Limitations

The debugger is dependent on software that does not support all operating systems. Specifically, the backend, HDB, is dependent on the library `libdw`, a part of `elfutils` [7], which is not supported on Windows nor macOS. The `libdw` library is used to parse DWARF debug information. The software will not support multi-threaded programs since that is not supported by HDB and it is also deemed out of scope for this project.

Another limitation is that the application in its current state is only available as a standalone GUI. There is no way for a user to be able to get the same debugging experience through, for example, a text editor plugin.

1.3 Challenges

The main challenges of this project are to produce an accurate graph from the heap and stack data reported from HDB as well as to create a GUI in a purely functional programming language. GUI applications are studied with object-oriented languages in several courses at the Department of Computer Science and Engineering and this is what the group has experience with. This means that working on a functional programming GUI project is new for the group and this might introduce difficulties during the development.

2

Background

2.1 The Haskell Programming Language

Haskell [8] is a general purpose, purely functional programming language. All the specifics of functional programming and Haskell itself are not necessary to understand the debugger so our explanation will be kept brief. A basic Haskell function can be seen in Listing 2.1.

Listing 2.1: Function to calculate the n:th Fibonacci number.

```
1 fib :: Int -> Int
2 fib 0 = 1
3 fib 1 = 1
4 fib n = fib (n-1) + fib (n-2)
```

The syntax is as follows; line 1 defines the type of the function `fib`, it has an integer (`Int`) argument and an integer (`Int`) return value. Lines 2 to 4 define the functionality of the function. Within this span there are three separate bindings of the function.

This way of writing a function is called pattern matching and it works so that the first line, line 2, matches when the first argument has the value of 0 and then returns 1. The second matches an argument of 1 and also returns 1. The last line matches an argument with any integer value and assigns it to a variable `n`, and is recursively defined to return the value of `fib (n-1) + fib (n-2)`.

Recursion is a mathematical concept where a function is defined in terms of itself, for example `fib n` is defined with the help of `fib (n-1)` and `fib (n-2)`. As Haskell is a pure functional language the concept of for-loops and while-loops does not exist, so code with “loop” functionality has to use recursion instead.

Taking a function as an argument is an important aspect of functional programming. This concept is called higher-order functions and means that functions are treated like any other value and can thus be used as an argument and return values among other things. The argument `f` for the function `appl` in Listing 2.3 is an example of this.

In Haskell it is possible to declare infinite lists. This works because of lazy evaluation which will be covered in depth in Section 2.2. It is possible to declare infinite lists in multiple ways and the example in Listing 2.2 shows two.

Listing 2.2: Examples of infinite lists.

```haskell
-- an infinite list of 1's, [1,1,1,1,1,1,...]
ones = [1,1..]

-- a function which generates an infinite amount
-- of Fibonacci numbers
fibs n = fib n : fibs (n+1)

fibs 0 -- -> [1,1,2,3,...]
```

The first is a list declared with `..` at the end. This syntax makes the compiler try to figure out the interval between the elements, for example in `ones` the interval is 0. The second example recursively generates a list of Fibonacci numbers from a specified input.

Later in the thesis "closures" will be referred to. A closure is a way to contain local variables outside their original scope in programming languages with higher-order functions. Such a variable is commonly referred to as a "free" variable. An example can be seen in Listing 2.3.

Listing 2.3: A basic closure.

```haskell
appl :: (Int -> Int -> Int) -> Int -> Int -> Int
appl f = \a b -> f a b

add :: Int -> Int -> Int
add = appl (+)

var = add 4 5 -- -> 9
```

To construct a closure you need a function with, or without, arguments that returns another function that uses the previously mentioned arguments in its computation. In the example, the function `appl` has an argument `f`, which through `appl` can be used outside of its scope as can be seen for `add` in `var`.

2.2 Lazy Evaluation

Lazy evaluation is a technique of evaluation which does not require the arguments to evaluate fully before evaluating the rest of a function. By utilising lazy evaluation it is possible to use infinite data structures, such as lists, in a program as they will only be evaluated when needed by the program. For example `ones` and `fibs 0` in 2.2.

The opposite of a "lazy" program is an "eager" program. An eager program evaluates the arguments before continuing with the evaluation of a function. Lazy evaluation exists in Haskell and is implemented using "thunks". A thunk is a type of object on the heap.

A thunk represents an expression which has not been evaluated yet. When it is needed the thunk partially evaluates and it is replaced with the result. This result is later returned on each occasion the thunk would have been called. In Haskell a function may also reuse a value if it has already been evaluated within that function. This is often referred to as "sharing".

An example could be a function which calculates the n:th Fibonacci number by describing an infinite list of all Fibonacci numbers and then finds the n:th number in that list. The rest of the list after the highest evaluated number will be represented by an unevaluated thunk in memory, but when trying to access number n again the previously evaluated thunk will return its value. This can be seen by evaluating the `fibs` function shown in Figure 2.2.

2.3 DWARF

DWARF [5] is a standard for debug information. It was introduced to standardise debug information in executable files so that any debugger with DWARF support could work on an executable with DWARF information. When a program is compiled with the instructions to generate DWARF information, debug information is generated and embedded into the executable file. This data contains everything needed to debug the program, including functions, types, and variables.

The way this data is added to the executable file is defined by the DWARF standard and enabled during the compilation process by invoking GHC with the option "`-g`". DWARF can be used with any format of executable but is traditionally used together with binaries of ELF format [7]. In our case, as we use a GNU/Linux operating system, the DWARF data is used in the ELF object format.

To find out what data exactly is embedded with DWARF data, or check the integrity of the data in object files, it is possible to read it through several tools described in [9], such as: `readelf`, GDB, `dwarfdump`, `llvm-dwarfdump`, and `objdump`.

Because only recent versions of the GHC have made it possible to generate DWARF information, our debugger is going to be one of the first debuggers for executable files compiled by GHC.

2.4 HDB

HDB [4] is a debugging library created by our supervisor Krasimir Angelov. The library provides debugging capabilities for compiled Haskell programs as well as an interface to interact with the library. HDB provides functionality such as providing information about a closure that is being evaluated, returning stack and heap information, and finding information from a file name and a source code location. The interface of HDB is visible in listing 2.4 where the functions with `Debugger` represent the functionality described above.

The interface of operations is utilised in our debugger with the help of `findFunction` to get the `LinkerName` of a breakpoint. The `LinkerName` represents the name of an

Listing 2.4: HDB's interface, from [4].

```haskell
peekClosure ::
  Debugger ->
  HeapPtr ->
  IO (Maybe (LinkerName,GenClosure HeapPtr)),
getStack ::
  Debugger ->
  IO [(LinkerName,GenClosure HeapPtr)],
getSourceFiles ::
  Debugger ->
  IO [FilePath],
findFunction ::
  Debugger ->
  FilePath ->
  (Int,Int,Int,Int) ->
  IO [LinkerName],
findSource ::
  Debugger ->
  LinkerName ->
  IO (Maybe SourceSpans)
```

object generated by the compiler, in this case, a closure generated by GHC. The function `peekClosure` takes a `HeapPtr`, which is an address in memory, and returns a pair of the name and closure associated with it. This makes it possible to chain together different closures since many closures have `HeapPtrs` in them. When the function `getStack` is called the different elements on the call stack are returned.

HDB provides the function `startDebugger`, seen in listing 2.5, which is used to start the debugger. It takes in a list containing a binary to be executed, and command line arguments for that program. It also takes in a callback function. This callback function is where the debugger gets access to the `Debugger` argument and is called every time the debugging process is halted by an event.

Listing 2.5: HDB's `startDebugger` function and callback.

```haskell
startDebugger ::
  [String] ->
  ( Debugger ->
    LinkerName ->
    Maybe SourceSpans ->
    [HeapPtr] ->
    IO DebuggerAction ) ->
  IO ()
```

The callback handles the event and returns one of the actions defined on Listing 2.6. Using these actions one can `Step`, `Stop`, or `Continue` the debug process, where the different operations step to the next evaluation, stop the evaluation, or continues to one of the `LinkerName`:s in the list respectively.

Listing 2.6: HDB's `DebuggerAction` datatype.

```haskell
data DebuggerAction
  = Step
  | Stop
  | Continue [LinkerName]
```

HDB uses libdw to extract DWARF debug information and reads line information which has source code information. This can be used to highlight the relevant part of the source code. It can also reconstruct the stack state with the help of the information that GHC stores in the binary for the garbage collector and analyse type data and read closures in heap and stack with appropriate names and memory addresses. The returned closures are of the type `GenClosure` which is defined in the library `ghc-heap` [10]. Our debugger uses this to create a graph representation of objects in heap and stack for every execution step [11].

2.5 GHC Compilation

Compiling is a complex task and it is not necessary to know all the details to understand this report. An outline of the GHC compilation process is given in Figure 2.1. The image shows that the source code is first parsed and typechecked, then simplified and translated through three intermediary languages, Core, STG, and finally Cmm. These intermediate languages transform the code in different ways, for example all pattern matching is translated to case expressions in core and all functions are expressed in terms of lambda functions in STG code. The Cmm code can then be used to generate machine code, LLVM code, or C code [12].

Certain GHC optimisations on generated code can change the program, for example by inlining functions. For a function to be inlined means that through the compiler's optimisation process any call to the function is replaced directly with the definition of the function. Compiler flags such as `-ddump-simpl-stats` give useful information about optimisations done and `-ddump-inlinings` about which functions were inlined. Unwanted inlining can be avoided directly in code with instructions to the compiler using the pragma `NOINLINE` [13].

An example of this pragma is shown in Listing 2.7. Even compiling source code without any optimisation may optimise small functions, another way to circumvent this is by specifying a function as an exported function from a module, also as seen in listing 2.7.

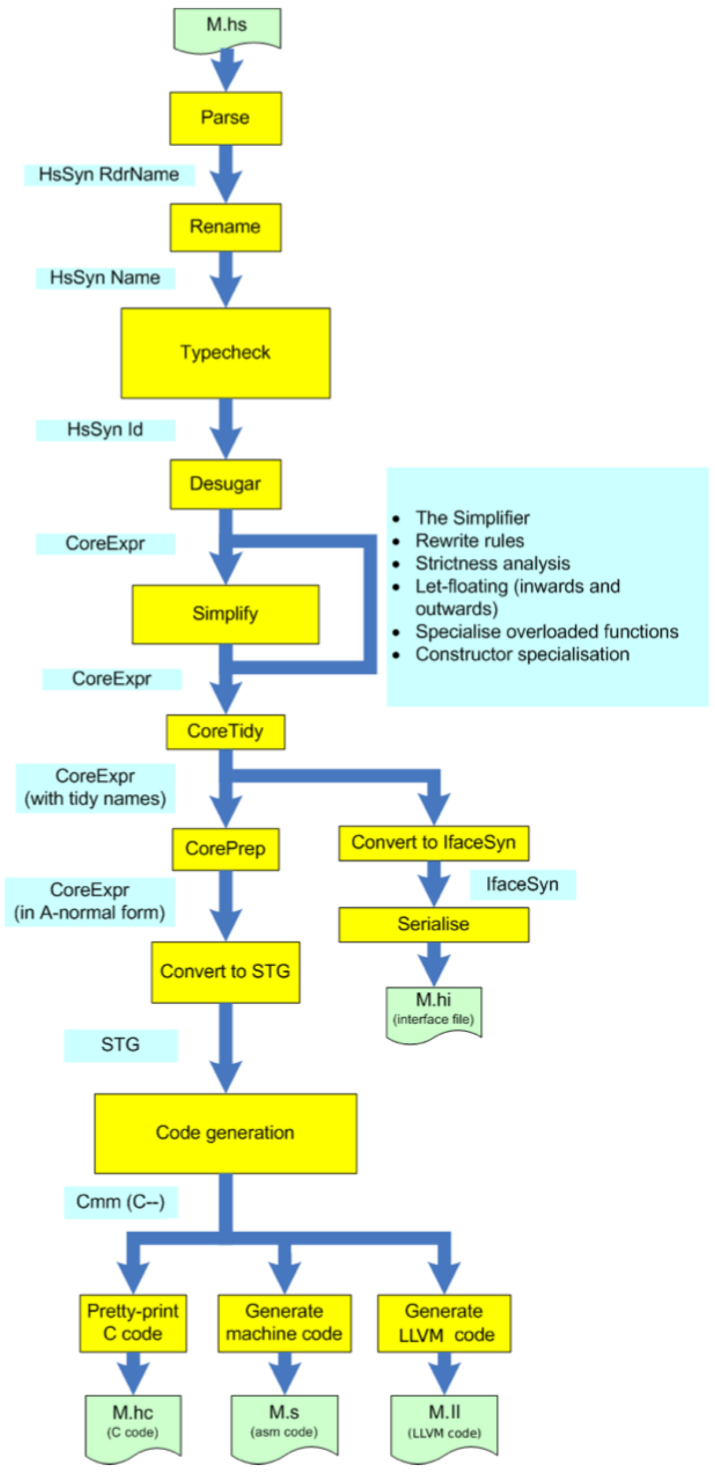


Figure 2.1: The GHC pipeline, image from figure 5.2 of [12], licensed under CC BY 3.0.

Listing 2.7: Pragma example and exporting of functions.

```haskell
module ModuleName (fib, add) where

fib = ...
{-# NOINLINE fib #-}

add = ...
```

The module is specified with the name `ModuleName` and the names within the parenthesis separated with comma are the functions which are exported from the module. This can be useful to know for a user which wants to test our debugger and has some issue with a minor function in a smaller program.

2.6 The GHC Heap

The GHC heap consist of entities called heap objects. Each of these objects contains an info pointer which points to an info table. The info table holds the layout information of the object as well as the entry code. The entry code consists of the code that will evaluate the closure, if the closure can be evaluated. The layout information describes the type of closure, data for the garbage collector, closure flags and other fields for profiling, parallel computing, and debugging fields [14]. The heap object also contains the payload. The payload can consist of both pointers and non-pointers. The payload of a function closure for instance contains pointers to the free variables of that function [15].

In order to discuss specific heap objects, it is important to clarify what unboxed and boxed types are. Boxed objects are represented as their own closures while unboxed types are stored directly in the data constructor of another heap object [3]. For example the constructor for an integer (`Integer`), is denoted I#. It constructs an integer by having an unboxed integer in its payload. Unboxed objects are preferable for optimisation reasons since they do not have to be allocated as their own closure.

There are different types of heap objects according to [14]. Pointed, unpointed and administrative objects. Pointed objects are heap objects that can be evaluated. Examples of these can be seen in Table 2.1.

Table 2.1: Some heap objects described in [14].

Thunks	Closures that are not yet evaluated.
Data constructors	—
Functions	—
Partial applications	Associated with the updating of a function valued thunk.
Indirections	Overwrites a thunk when it has been evaluated.
Black holes	Overwrites a thunk when it is under evaluation.

Unpointed objects are objects that will never take the form of a thunk since they are not evaluated. Examples of these are arrays and MVars [14], a type for shared state when concurrency is needed.

Lastly, there are administrative objects that are handled by the runtime system. An example of these are thread state objects, which each represent a thread. [14]

2.7 The GHC Stack

A stack is used to store previous and upcoming closures by the run-time system (rts). The stack is used with a current evaluation to be able to go back to where it was called from on finished evaluation.

In GHC the stack is represented by a stack, the data structure, of evaluation calls. In [16] the GHC stack is produced by using three theoretical stacks; An argument stack, a return stack, and an update stack. These different stacks contain closure addresses and primitive values, continuations for case expressions, and update frames respectively, these will be explained in detail later.

The actual implementation uses two stack-structures in the original runtime system by [16], but more modern versions of the GHC stack use a single stack-structure [14]. The elements contained in the different theoretical stacks mentioned above will be described in detail in the next section.

2.7.1 Stack Objects

The data contained in the stack is represented by stack objects. These are generally referred to as stack frames [17]. Stack objects have a similar structure to heap objects, but instead of the info pointer they have a return address, which is often referred to as the info pointer. The info pointer contains an entry code, which is executed when the program must return from the current closure while the payload contains everything else [18]. In GHC there are three kinds of stack objects: return addresses, update frames, and seq frames [17].

Return addresses are used for case alternatives on return from a constructor, and as all evaluations are performed by case expressions this also includes functions [16]. There are two types of these objects; `RET_XXX` which are addresses generated by the machine code compiler, `BCO_RET` are addresses generated by the byte-code compiler. Both return address object types contain a payload of the free variables of the case alternatives. Update frames trigger updates, once entered its payload is overwritten with the result and returns to the next stack object. Specifically, they contain a return address, a pointer to the next update frame, and a pointer to a thunk [18]. Seq frames are used for the `seq` primitive. `seq` is used to prevent unnecessary laziness by forcing evaluation of arguments and it is given this special update frame. There are also stop frames which are at the very bottom of the stack. When a stop frame is entered the thread is killed.

2.8 Previous Work

There exists a few debuggers for Haskell. However, none of these debuggers work for compiled Haskell without making changes to the source code or transforming the language in other ways according to [6].

2.8.1 ghc-vis

`ghc-vis` [3] is a tool to visualise Haskell evaluation in GHCi, meaning that it does not work for compiled Haskell code. It also is not a step by step debugger but it allows the visualisation of the heap. The graphs generated by `ghc-vis`, for example Figure 2.2, are an important inspiration for our visualisation of compiled Haskell program evaluation. The visualisation provided by `ghc-vis` makes it easier to understand lazy evaluation, which is also exactly what we want to achieve with our debugger. A concrete example of this inspiration is the layout of the nodes, with sections in the node for every outgoing edge, found in both programs.

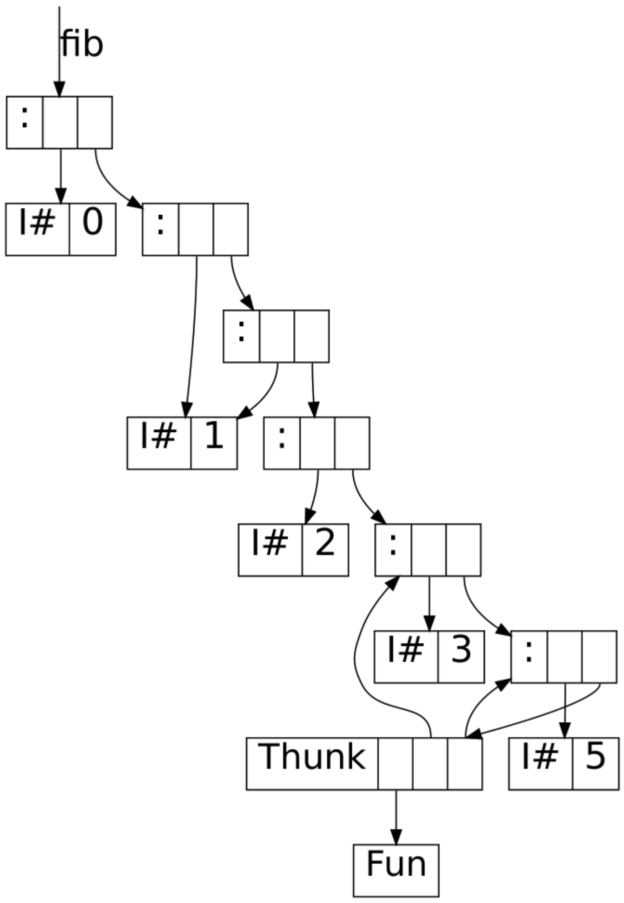


Figure 2.2: Image of `ghc-vis` Fibonacci example.

The example in Figure 2.2 shows the evaluation of a list of Fibonacci numbers generated by the function in Listing 2.8 evaluated to its fifth element. In this example you can see that some elements are used more than once, the first and second element, 1.

You can also see that the evaluation is not yet completed and the unevaluated part of the expression is shown as *Thunk* which points to *Fun*. *Fun* represents a function, in this case `fib`, which needs more input to evaluate. So when the evaluation progresses *Fun* will receive more input and take the evaluation further.

Listing 2.8: Another function to generate a list of Fibonacci numbers. Function from [3].

```haskell
fib = 0 : 1 : zipWith (+) fib (tail fib)
```

2.8.2 GHCi Debugger

The GHCi debugger [19] is a debugger within the GHC's interactive environment. It has basic capabilities like settings break-points on function definitions or expressions, stepping through execution, and "tracing mode," which can display the history of the evaluation from a break-point. However, the GHCi debugger does not support debugging of compiled Haskell programs and visualising of the heap, which is the goal of our project.

2.8.3 Hat

The Hat program can generate trace information from a Haskell source code file. However, Hat only supports Haskell98 language standard together with some Haskell2010 libraries. This does not represent the current standard so it is not viable for modern programs.

2.8.4 Hood

Hood [20] is a debugger for Haskell which supports text-visualisation of data structures. There also exists a GUI for hood, GHood [21]. The tool is currently not under active development and requires the code to be changed for the data structure observations to be shown which means that it is not suited for debugging compiled programs according to [6].

2.8.5 GDB

Haskell code is transformed to native executable code through a multi-stage pipeline. This process is described in more detail in section 2.5 on GHC compilation. We can take advantage of DWARF debugging information created with the GHC and debug the executable using The GNU Project Debugger, GDB [22].

Debugging with GDB is only useful at a very low level and mainly comes down to analysis of assembler code along with limited symbolic information. As such, it is most useful for debugging the GHC run-time or for debugging C libraries linked with the Haskell code.

3
Method

3.1 Development Process

The intended way of working during this project was to use ideas from Scrum. The project did not have a product owner, but had an external stakeholder in the form of our supervisor. It would also use concepts such as sprints, user stories, and short, but frequent, meetings. The sprints were decided to be two weeks long at first, but later in the development it was decided to shorten them to one week. The reasoning for the usage of Scrum development was that it would ensure that there would be a prototype of the product at the end of every sprint and thereby ensure steady development of the product.

The tool Trello [23] was used to act as a scrum board and it was used to organise the tasks and user stories of the project. The version control system Git [24] was used to enable every member of the team to work on the codebase. The remote repository used was GitLab. The `main` branch hosts the completed parts of the program, and the goal was that this branch would always be able to compile and run without errors. Other work was contained in their own branches and was only to be merged into `main` if they were functional, and if applicable; tested.

3.2 Software Implementation

The main feature of the debugger is to show the current state of the heap and the call stack as a directed graph, relative to Haskell source code in a single execution step. The program `ghc-vis` was our main source of inspiration when making decisions concerning software implementation.

Before each step of software fabrication, we discussed components that would add functionality to the debugger, primarily focusing only on the most essential. Different options, their strengths and weaknesses regarding planned functionality requirements were further discussed and tasks were assigned to group members through an agile project management software when implementing them.

Haskell was used for development of the debugger as it provided a natural way of communicating with HDB, and we, the group members, wanted to improve our Haskell knowledge. As we prioritised integration with the operating system, we decided to use gtk2hs [25] which is based on the GTK3 GUI framework. An addition to the mature GTK framework, is also the rapid application development tool Glade, a part of GTK that after a short period of testing proved to be a valuable tool to adapt to our needs which involves frequent changes of the user interface.

For displaying the graphs we chose to use the graph creation software Graphviz [26]. It acts as an interface for Dot [27] that is a language that is used to generate graphs. This was done for two reasons: Firstly, since ghc-vis uses a Haskell binding for it [28], there was potential in reusing already written code. Secondly, Graphviz has a tool that enables further manipulation of the graph called `unflatten`.

3.3 Personas

At the beginning of the development, a target group for the project was determined. From this target group, two types of end-users were created and aimed to be designed for, known as personas. A persona is an abstraction of a user that serves to guide the development of a program [29]. There are two types of personas, one of them is called the primary persona and one is called the secondary persona. The primary persona is the main target for the design of the interface and the secondary persona is someone that agrees with the primary persona in most parts but with exceptions of additional needs, which does not upset the primary persona [29].

The project's primary persona is an individual with Haskell experience. They have good knowledge of the language itself and the Haskell evaluation process. They understand concepts such as lazy evaluation and have some knowledge of how the GHC run-time system works.

The secondary persona chosen for this product is an individual that is fairly new to Haskell, with basic knowledge of the language itself, such as pattern matching, recursive functions, higher-order functions, and so on, but lacks a deeper understanding. This type of user will require assistance in understanding the graph that is shown to the user, both in documentation as well as simplifications of the heap and stack representation.

3.4 Testing

3.4.1 User testing

To develop the application in the best possible way, user-testing was performed in the later stages of the project with people in our target audience.

To perform the testing we used the method of Usability testing [30]. This method involves creating tasks that the user walks through. By doing this, it is easier for a team to identify and fix the parts of the application that are confusing for the user. The test follows the format of the think-aloud protocol [30], this protocol means that the user describes what they think and how they feel while doing a given task. The tasks used in our testing are specified in Table 3.1.

The first task entailed loading the file that was to be debugged into the debugger. This forced the users to interact with both a file dialog and the decision to either compile a selected file or chose a pre-existing binary. To clarify the second task; the

Table 3.1: User testing tasks.

Task 1	Select the Haskell file named “test.hs” to be debugged.
Task 2	Set a break-point at the function “foo”. Step to that break-point and inspect the graph. Explain the different colouring of the nodes.
Task 3	Finish the execution of the program. Find the output of the debugged program.

main idea is to force the user to look for “help” within the application. The third task was to find the program output section of the debugger and to find the output of the program that was debugged in said section.

After the tasks were completed, some questions were asked to the user. Did you encounter any challenges during the tasks? If yes, explain what was challenging. On a scale of 1, which is, not very likely, to 5, which is very likely, how likely would you be to recommend this program? Was the help section enough to understand how to use the program? Any other thoughts?

While performing the test with the user, at least one group member participated and took notes. The results from the user-testing and how the findings improved the program, can be further read in Chapter 4, Section 4.4.

3.4.2 Software-testing

As development started the intention was to test all appropriate code using the QuickCheck [31] library. However, as most code written for the debugger is for the GUI a lot of it has side effects and this makes property-based testing considerably harder. Therefore, the program was mainly tested by performing a sequence of operations in the GUI, in order to find and fix bugs.

The integration testing was performed on each feature’s completion by the author and, if possible, by some other group members. Having someone other than the author test the same feature gives new perspectives and sequences of operations since they do not necessarily use the software in the exact same way.

4 Results

4.1 Software Architecture

The majority of students at the Department of Computer Science and Engineering have not encountered the development of more advanced applications using the functional programming paradigm. The applications that are studied and constructed in the mandatory project-based courses for the Software Engineering and Computer Science bachelor's degrees are exclusively based on object-oriented programming languages unless the students themselves decided to work in some other paradigm.

None of the members of this group had previous experience with functional programming on a larger scale than the development of smaller programs from courses at the Department. Despite this, there has been an attempt at separating the functionalities and capabilities as much as possible to allow for reuse. For example, all GTK parts are excluded from the part which communicates with HDB and generates the graphs to enable further development.

A basic diagram of the application structure is visible in figure 4.1. In the figure, GUI, HDB, Graph, and Debug correspond to parts of our application where GUI, Debug and Graph are parts that we have constructed.

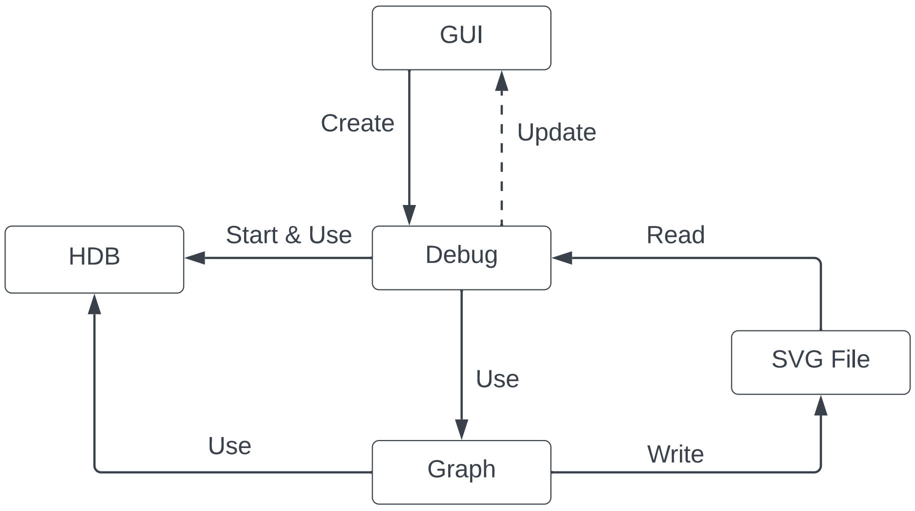


Figure 4.1: The software architecture.

GUI starts a thread with Debug that starts the debug process in HDB and responds to data sent to the callback function. The reason why there is a need for concurrency is that this process needs to run independently of the GUI. Otherwise, the GUI thread would halt execution until the debugging done by HDB is done.

Graph generates a graph which is put into an SVG File, which is then read by Debug for updating GUI so it can be displayed.

The callback function in Debug is executed when HDB steps. When continue is selected, HDB will not call back until one of the linker names specified in the call appears as the currently evaluated heap object. The main thread communicates with this process by using shared thread-safe variables called `MVar`:s.

An example of such a variable is the one that keeps track of the most recent executed command, Stop, Step or Continue. Whenever a step is executed, or a breakpoint is hit, the graph is generated. The graph is then saved as an SVG file when the debug process executes a callback to the main GUI thread. The graph file needs to be read when HDB has finished a step but this notification creates a weak dependency. The reason for this is the use of higher-order functions.

Because GTK must be updated from its main thread it is not feasible to update it concurrently. To be able to notify the GUI from the Debug thread a function is passed to Debug which enables running operations in other threads with the GTK application's main loop. The type of the function is general enough to be a function of another graphical framework.

By having this function as an argument, there are no GUI dependencies in the Debug module. By doing this it becomes dependency-wise independent from the GUI. This results in the possibility of using the Debug module and graph generating capabilities with other interfaces in the future.

4.2 Graph Representation

The decision was made to show the heap and stack to the users as a directed graph, where every edge represents a pointer from one node to another, and every node represents a closure. The reason for this was that a natural way to visualise pointers is to view them as edges in a graph between different closures. The ability to view what is on the stack also gives further information about the current state of the program by showing upcoming evaluations. In some cases, the information that the node that is currently evaluated and associated nodes represent will be difficult to decipher by the user. In those cases, the stack can bring context to the evaluation.

4.2.1 The visual representation

The node representation is heavily inspired by ghc-vis, and the nodes themselves are divided into sections based on the number of outgoing edges as well as the data that is held by the node. The reason for this specific layout was that it led to a clearer separation between the different outgoing edges from a node since every edge comes from a designated part of a node. This means that arguments will be ordered. It was also thought to be an intuitive way of displaying the concept of a heap since the representation is reminiscent of how memory addresses often are shown. The layout of the nodes is illustrated in Figure 4.2.

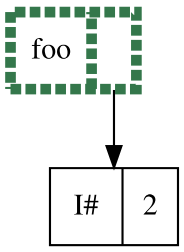


Figure 4.2: The function `foo` points to a boxed integer (`I#`) as an argument.

Two nodes can be seen in the image. `foo`, that holds a pointer to another node that is a constructor for an integer. The fields to the right of the name can hold a value directly or house a pointer to another node.

Since the graph can be wide it was important to alert the user's attention to nodes of importance. Therefore different colours were used. The node that is currently being evaluated is differentiated from the rest of the nodes by being coloured green and by having a striped border, see the node labelled `foo` in Figure 4.2.

The nodes that are only shown when the stack trace is present in the graph are coloured yellow. The exception to this is when a closure present in the stack trace is returned directly by HDB as a part of the current evaluation, in that case it is also coloured yellow. If an object present in the stack trace is currently being evaluated it will be coloured green. See Figure 4.3 for an example where every node is derived from the stack trace and `stg_enter` is the node that is currently being evaluated. Since objects from the stack trace can be on the heap it means that some heap objects will be coloured yellow.

Any node that does not meet the above mentioned criteria are coloured black.

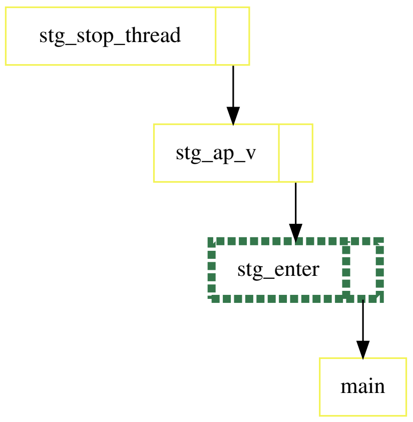


Figure 4.3: An edge case in the graph where all the nodes shown are derived from the stack trace.

There were several decisions made to limit the width of the graph. First of all the names of the closures were simplified. This was possible due to the fact that the names of some of the heap objects included information that was not deemed important for the end-user to know. The names also need to be decoded with the help of a library called zenc [32] to improve readability. For example the node name: `base_GHCziBase_zgzg_info` becomes `base_GHC.Base_>>_info` after zenc is used. After removing the unnecessary information, the package name `base`, module name `GHC.Base` and the suffix `_info` it simply becomes `>>`, which is the function's name.

Another consequence of the width of the graph was that a form of zooming functionality was necessary to implement.

Since there exists an option to set the desired depth of the graph, there needed to be a clear indicator of whenever a part of the graph is hidden. To differentiate the truncated nodes they have similar border colours to the normal nodes, but are greyed out. The content of a truncated node is "`...`". In Figure 4.4, a graph with limited node depth is shown and truncated nodes are displayed.

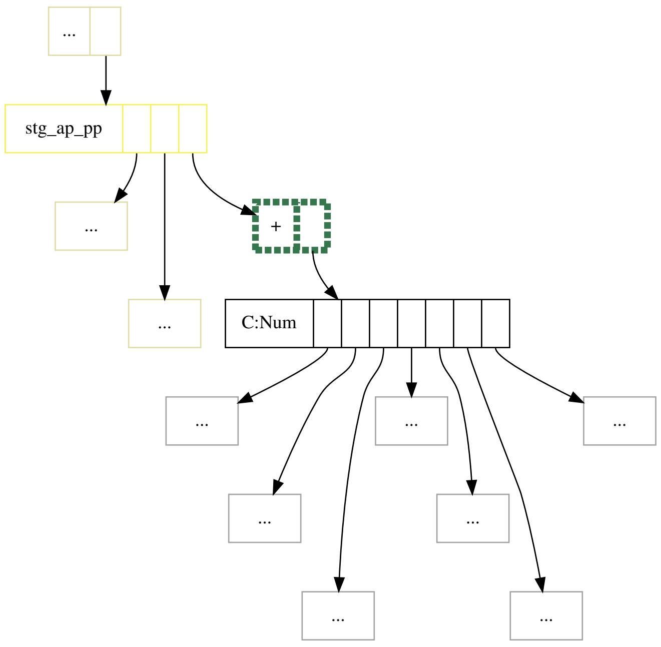


Figure 4.4: A graph where the depth is set to 1 and some nodes are truncated. The currently evaluated node has a depth of 0.

Finally, the functionality to limit the graph's depth was added. This was done in two ways. First of all it is possible to hide the call stack of the graph altogether with the exception of stack nodes that are a part of the current evaluation. For example, an update frame returned by HDB. Secondly, the option exists to set the desired depth of graph. This was done partially so that the user could determine the size of the graph but it was also implemented so that no graph would be impossible to view. In theory, and most probably in practice, a graph could grow big enough to be impossible to be represented by the SVG viewer.

The graph is also modified by the tool `unflatten`, which is included in Graphviz. This tool modifies the graph by modifying its layout and thereby decreasing its width. A graph modified by `unflatten` can be seen in Figure 4.5.

By default Graphviz tries to keep all nodes of the same level at the same height,

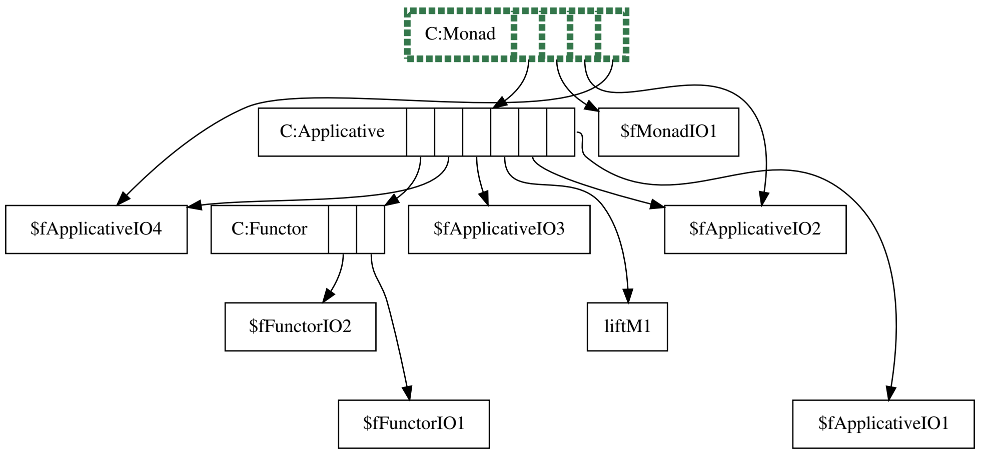


Figure 4.5: Here `unflatten` has modified the graph by displacing the nodes in order to decrease the width of the generated image.

resulting in an at times very wide graph. As shown in the example the nodes are no longer on the same level anymore due to the usage of `unflatten`. The way this process is done in our debugger is by saving the graph as a Dot file and using that as a parameter to `unflatten`. The output is then converted to an SVG file.

4.2.2 Example graphs

Here a few examples of how different Haskell concepts are visualised will be presented. An example will also be shown of how the program can be used for profiling purposes.

The first graph, Figure 4.6, shows the string `"UTF-8"`. The currently evaluated node is the cons operator, the operator that adds an element to the front of a list. The figure shows how a string is represented by a list of characters, represented by node name C#, which is a constructor for characters. All strings (`String`) values in Haskell are represented in this way.

Sharing, as previously mentioned, is a concept in Haskell where evaluated values can be reused in the context where they are evaluated. The graph in Figure 4.7 illustrates this concept where the list contains a sequence of two different integers. The heap objects which represent the integers are reused as elements.

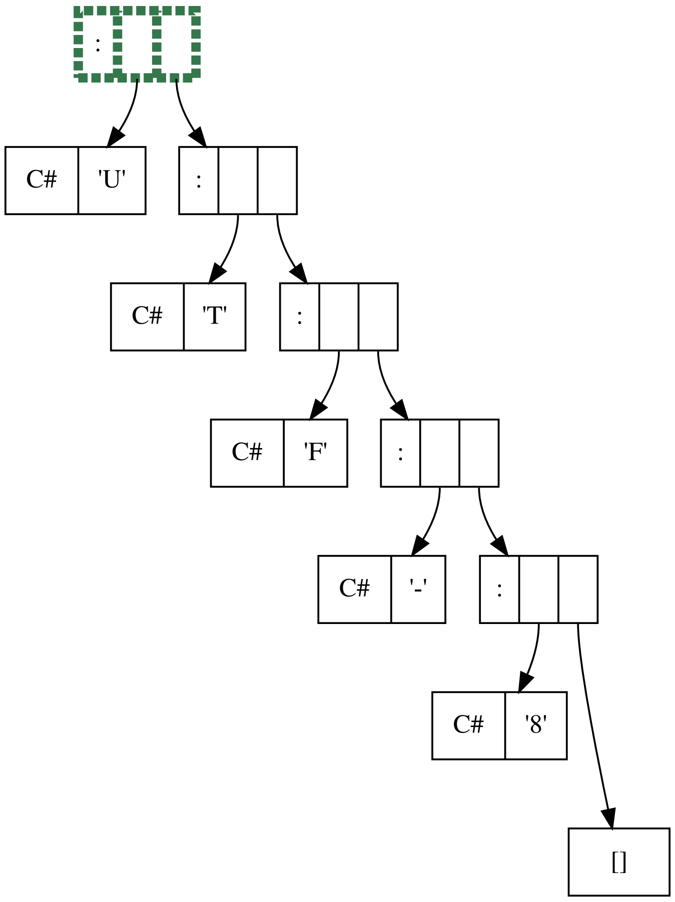


Figure 4.6: A list of characters.

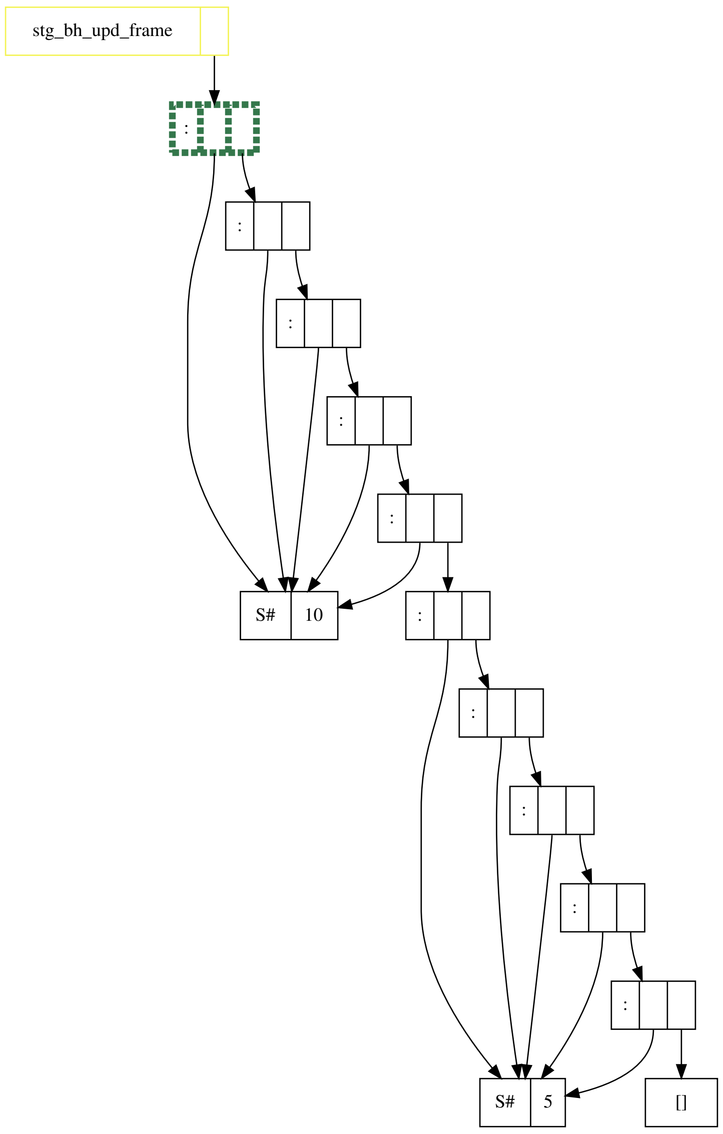


Figure 4.7: The sharing of values in the list [10,10,10,10,10,5,5,5,5,5].

The debugger is also able to handle cyclic data structures. The code that generates the example cyclic data structure can be seen in Listing 4.1.

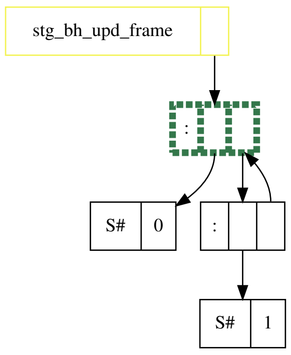


Figure 4.8: A cyclic list comprised of zeroes and ones.

A list is created where the first element, 0 is followed by 1 and then itself again. This creates an infinite sequence of zeroes and ones. This is visualised in the Figure 4.8. The first element of the cons operator in the graph is 0. The second element is another cons operator that points to the number 1 and its parent.

Listing 4.1: A function that creates a cyclic list [33].

```haskell
cyclic = do
    let x = 0:y
        y = 1:x
    x
```

To illustrate how the debugger can be used for profiling purposes the graph found in Figure 4.9 will act as an example. The graph is generated by the function `sum’`, which recursively adds all the elements of a list together, see Listing 4.2.

Listing 4.2: A function that causes a space leak and a fixed version of it.

```haskell
sum' n [] = n
sum' n (x:xs) = sum (n+x) xs

sumEager !n [] = n
sumEager !n (x:xs) = sum (n+x) xs
```

The first parameter `n` is the sum which will not be evaluated until the entirety of the list has been traversed. In Figure 4.9, the sum corresponds to the left part of the graph starting with `Thunk:4065c8` and can be seen as the unevaluated expression (. . . (((0+1)+2)+3). . .). The value 100 is the last value of the list as the graph is created by the call `sum’ 0 [1..100]`. This is a space leak since the unevaluated expression takes up more memory than needed. This can be avoided by circumventing Haskell’s lazy evaluation by using the `!` declaration, as seen with `sumEager` in Listing 4.2.

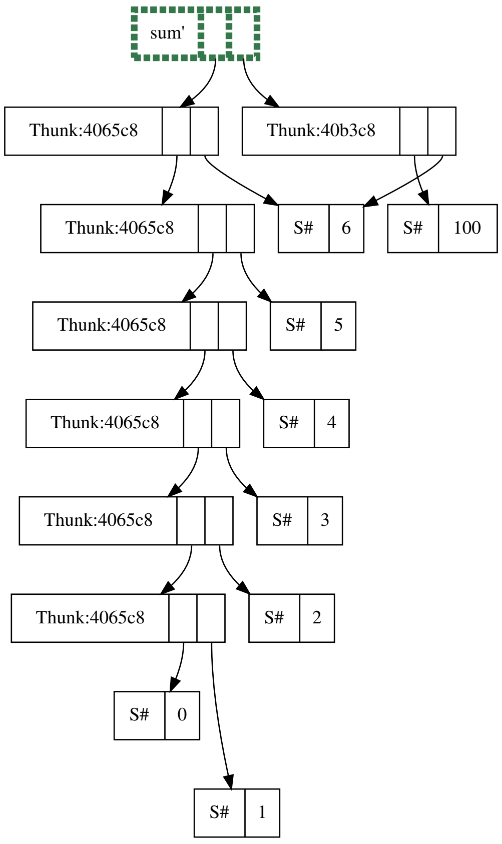


Figure 4.9: A graph that illustrates a space leak.

4.2.3 Algorithm

To generate the graph our algorithm uses debug data parsed from an executable by HDB. Whenever a step action is taken or a breakpoint is hit, HDB returns a closure. That closure can then be used to both determine the node that is currently being evaluated and request new closures from HDB.

The algorithm is as follows: when a closure is found by HDB, the name, address, pointers and arguments, are gathered and converted to a new data type. The info table and the entry code are not displayed to the user. The conversion is done in a recursive manner where every pointer of a closure is sent to HDB, by using `peekClosure`, so a new closure is returned. This was implemented by defining every node using a recursive data structure. See Listing 4.3 for the internal representation of a node and the different node types in our program. Thus every instance of a node has a list of other nodes in it.

Listing 4.3: The representation of a node in the graph.

```haskell
data NodeType = Root | Stack | Ordinary
                deriving (Show,Eq)

data Node = NodeClo String String [Node] [String] NodeType
          | EmptyClo
          | DuplicateClo String
            deriving (Show,Eq)
```

The type of node is also added, it is either a `Root` node, meaning that it is the currently evaluated node, a `Stack` node, meaning that it only has parents derived from the stack trace and it is not the currently evaluated node. An `Ordinary` node is derived from a `Root` node but is added to the graph regardless if the stack trace is present or not. A node can also be empty, if HDB returns no closure, or be marked as a duplicate if the node already has been added. As a catch-all, the last case is used to avoid infinite loops in the recursion due to cyclic data structures.

The reason for modelling the graph in this way was that we thought it would be easy to implement features related to interactivity. Specifically there were plans to enable removal of sub-graphs from the representation. It is possible to delete nodes by clearing the list of the node that holds references to the them. This would remove all nodes that are not referenced elsewhere in the graph. However, this specific approach is no longer feasible after cyclic data structures were handled. This is because a `DuplicateClo` contains insufficient information to be drawn correctly if a `NodeClo` with the same address is removed. Functionality similar to this can be implemented by storing the addresses of the nodes that should not be drawn, the difference would be that nodes that are referenced elsewhere in the graph would be hidden.

The stack is handled in the same way with the exception that the edges between the different objects of the call stack are added in order to show the ordering of them. The bottom-most element of the stack points to the next object on the stack until the top is reached. The top-most element on the stack then directly points to the node that is currently being evaluated. The exception to this is when the topmost element is a return frame. In that case, the frame itself is the node that is currently being evaluated.

Since it is possible that the topmost element of the stack is the element being currently evaluated the decision was made to always show the closure returned by HDB even though the stack trace was hidden. This is to ensure that the currently evaluated node is always shown. This also means that some stack objects in the graph will have a different number of arguments depending on if the stack trace is shown or not, as shown and displayed in Figure 4.10 for an example. This is because it is only possible to get detailed information about a stack object from the `getStack` function. Without using this function the graph will only display the arguments returned directly from HDB.

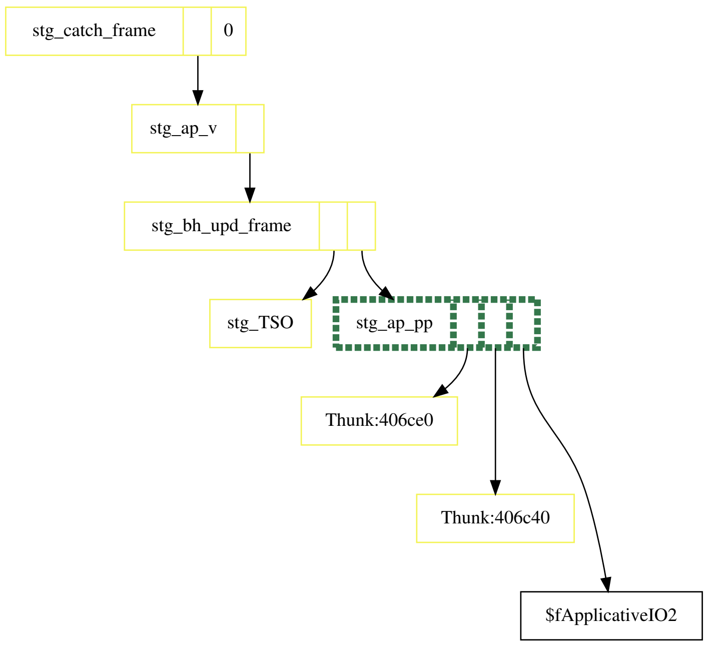


Figure 4.10: `stg_ap_pp` is an object that resides on the stack. If the stack is hidden, the two thunks in the graph would not be drawn, decreasing the number of arguments of the currently evaluated node.

Arguments of a closure are handled the same way as if it was the closure itself that had a pointer to it. In practice, it was deemed this had no effect on the debugging potential of the application. The resulting graph is then sent to the Haskell binding of Graphviz and an SVG is generated.

There are special cases that need to be handled by the algorithm. When a closure is received it is not always apparent how the data should be handled. For example, there is no separation between an integer and a character in regards to its representation as a value. This means that the only differentiating factor between the two types is the names of the constructors. Therefore the algorithm was modified to rectify this. If the closure is a constructor of a character the value is then converted from an integer to a character, for example in Figure 4.6.

Closures that are black holes, indirections and aspects of MVars are also hidden by default as they were deemed to affect the readability of the graph negatively without simplifying the debugging experience. A problem encountered during the process of extracting closure data was that for one type of array closure called `ArrWordsClosure`, the payload could be very large. The payloads were results of memory allocation processes and consisted of long strings containing arrays. The decision was therefore to limit the size of this type of array to 20 elements, and to illustrate this to the user, the string "..." was added as a suffix.

4.2.4 The graph as an SVG

There were two reasons why an SVG file was chosen as the preferred output format. Firstly it would be possible to extract data from it. This was used to centre the canvas of the graph on the currently evaluated node by extracting the coordinates of the node. Secondly, SVG files can be scaled without any loss of image quality. This fact was used to implement the functionality to zoom in to the graph.

4.3 The Graphical User Interface

A substantial amount of work went into crafting a well designed and functional GUI. The goal was to create a usable product that is intuitive to use and the overall look of the program was not the most important aspect of it. However, care was spent on those parts as well when time allowed it since sometimes resources had to be prioritised to implement new functionality instead of refining the GUI.

At the beginning of the project, research was done and inspiration was gathered from popular IDEs and text editors with integrated debuggers like VSCode, IntelliJ, and Eclipse. When creating mockups of the planned GUI it was clear that they resembled typical debuggers in an IDE. The reasoning for this was that since these debugging environments are widely used, the users likely would have experience with them, therefore it would be unwise to deviate completely from these types of graphical layouts.

The GUI is created in an attempt to satisfy both the primary persona and the secondary. As the secondary persona would have limited knowledge about the GHC heap and stack, the application was adapted with this in mind. Most notably this was the reason why a help section was implemented. The application also has advanced features that might be utilised by a more experienced user, such as the primary persona. For example, it is possible to set the maximum node depth, hide the stack trace, view a history of previously current evaluations in terms of their names, and by using compile options: showing long arrays completely and displaying the graph in its entirety since some nodes are hidden by default.

The primary window of our GUI contains necessary functionality for starting a debug session, performing operations, and analysing the results. In addition to the primary window, the user is able to view *About* and *Help* pages by selecting the menu items in the menu bar. This menu also includes items for opening a file dialog for selecting which program to debug, setting of maximum node depth, as well as opening a source code file for viewing.

When starting the application, the dialog for selecting an executable is displayed. The purpose of starting with a dialog is to have a clear entry point for the user, as seen in Figure 4.11.

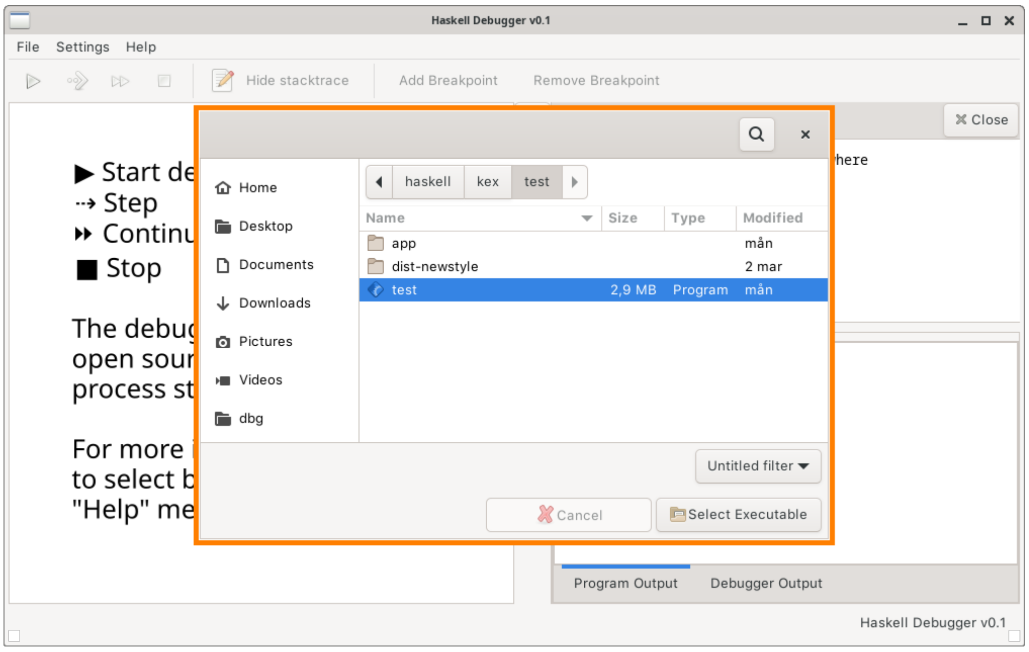


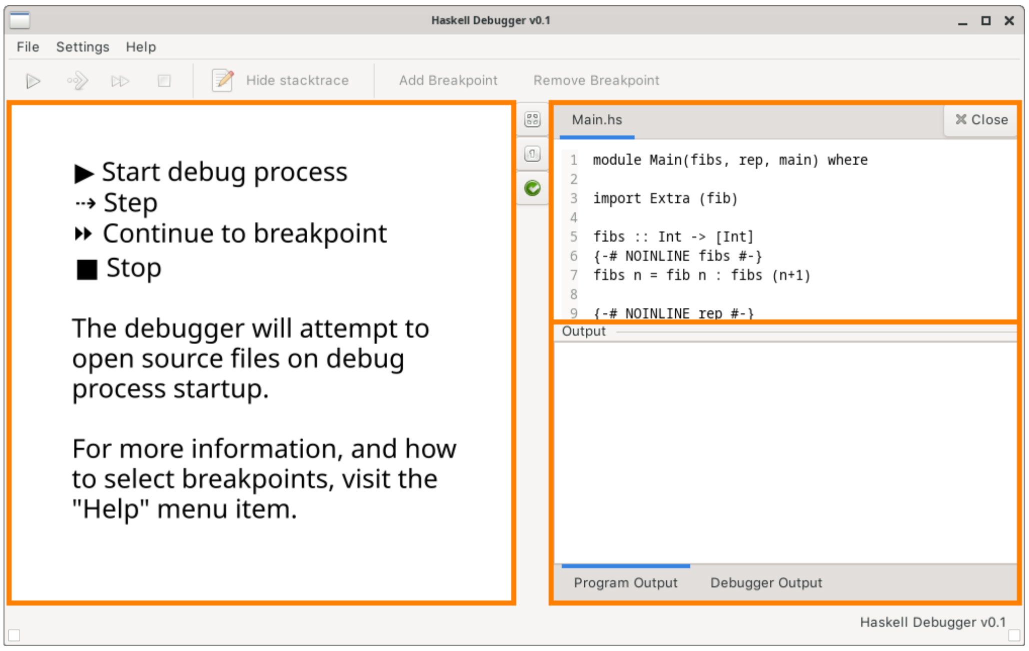
Figure 4.11: A clear Entry-point where the user selects an executable file.

The decision was made to design the GUI as a multi-paned application. The layout of the primary window has three panes, which can be viewed at the same time. These panes are is displayed in Figure 4.12. These panes are resizable within limits, and all panes are scrollable if necessary.

According to Cooper [29], the advantages of a multi-paned application are that independent but related information can be viewed at the same time. This reduces navigation between windows for the user, which suits our application. How to divide and place the functionality in our panes, was decided through research of existing debuggers.

In the studied programs like VSCode, IntelliJ, and Eclipse, the debug output is usually at the bottom and the code view to the right. In order to try to satisfy the users, the same layout pattern was used. The reason why the code was shown was so it would be possible to facilitate an easy way to set breakpoints and to notify the user when a breakpoint is hit. In the left pane, we chose to include the graphical visualisation of the heap and stack.

As already mentioned, the intention was to simplify the use of the debugger when navigating and managing windows. Therefore, the complete tool-set of frequently used operations during the debugging process was placed in our primary window as well. All these tool buttons are collected in the toolbar at the top, right under the menu bar. Both the toolbar and the menu bar are highlighted in Figure 4.13).



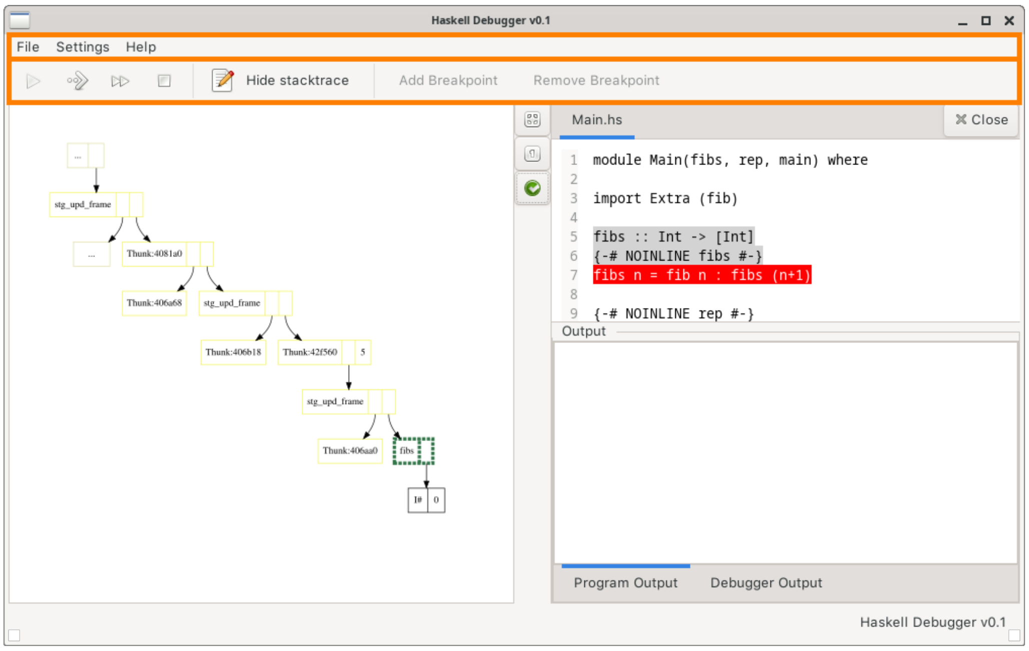
▶ Start debug process
⇢ Step
⏩ Continue to breakpoint
■ Stop

The debugger will attempt to open source files on debug process startup.

For more information, and how to select breakpoints, visit the "Help" menu item.

```
1  module Main(fibs, rep, main) where
2
3  import Extra (fib)
4
5  fibs :: Int -> [Int]
6  {-# NOINLINE fibs #-}
7  fibs n = fib n : fibs (n+1)
8
9  {-# NOINLINE rep #-}
```

Figure 4.12: Primary window, where the core functionality is split into 3 panes.



```
1  module Main(fibs, rep, main) where
2
3  import Extra (fib)
4
5  fibs :: Int -> [Int]
6  {-# NOINLINE fibs #-}
7  fibs n = fib n : fibs (n+1)
8
9  {-# NOINLINE rep #-}
```

Figure 4.13: The Menu-bar and the Tool-bar, additionally a breakpoint has been hit on the evaluation at line 7.

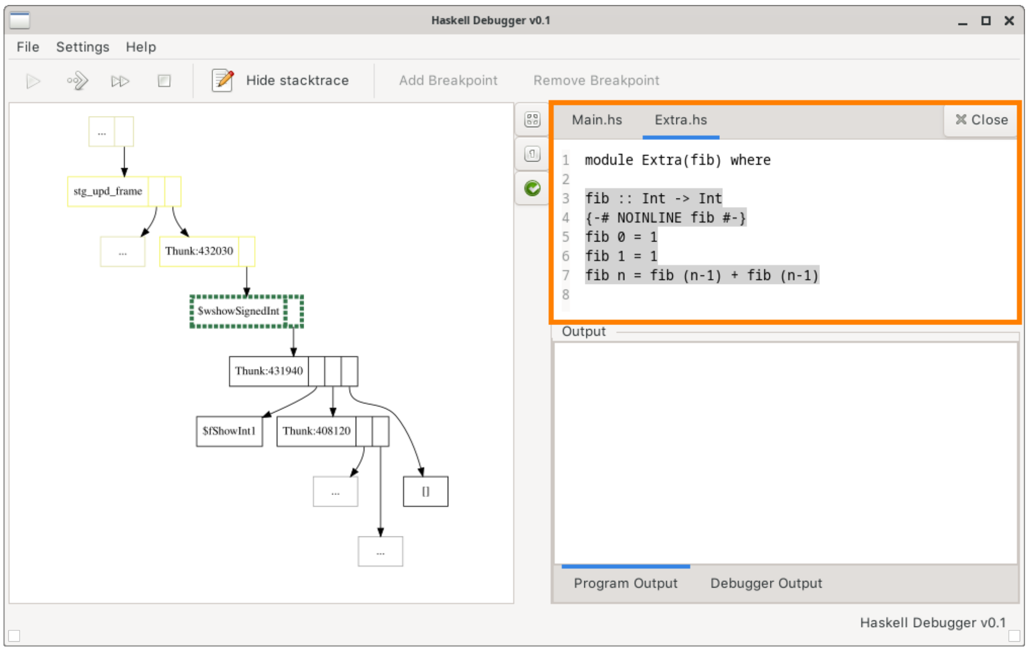
If possible the design of the tool buttons were made graphical rather than textual. This was a design choice made to facilitate for the users. Instead of understanding the buttons through reading, it is faster to recognise them with images [29].

Not all buttons are graphical since we tried to keep the images to default GTK options. In some cases the functionality is too specific for GTK to be able to provide images with such a purpose. In those cases there are text together with the image, or just text. All three types of tool buttons can be seen in Figures 4.12 and 4.13. Buttons that are related to each other also become easier to spot for the user. For complementing the visuals the toolbar includes a tool-tip for every button explaining the button which is helpful to first-time users in the case when it is not immediately obvious what a button does [29].

Above the toolbar in Figure 4.13, we find the menu bar which is displayed in a conventional way. Here the user is able to do normal commands such as switching to the *About* page or the *Help* page, opening files, or setting maximum node depth.

To create a breakpoint the user must select a span in the source code which includes the whole definition of a function, but not necessarily the type declaration, since HDB needs a source span to return the name of the function. The source spans required by HDB are fairly exact, it allows for a span to be larger than necessary, but it does not match with a smaller span. If a span without a function declaration is chosen a breakpoint will still be created, but the program will not stop at this breakpoint.

This is done by either selecting a span with the mouse while holding the control key or by first selecting the span and then pressing the add breakpoint button in the toolbar. When this is done the selection is highlighted with a grey colour. A breakpoint is visible in the source code pane with grey background in Figure 4.14.



```haskell
module Extra(fib) where

fib :: Int -> Int
{-# NOINLINE fib #-}
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-1)
```

Figure 4.14: A selected breakpoint is highlighted.

Similarly, removing a breakpoint is done by selecting the highlighted code in the source pane, and then pressing the *Remove Breakpoint* button in the toolbar. This causes the breakpoint, as well as the highlighting in the source pane, to be removed.

Whenever a function with a source location is evaluated by HDB the source spans of that function are highlighted red in the GUI. The choice of colour is inspired by other debuggers which might highlight the current line in red as well.

Since the user must be able to set breakpoints in several different files the program needed a way to display these files simultaneously. This was done by using a tabbed pane for displaying the source code, with a different tab for each file. The user can navigate between the different files by using the tab markers at the top of the pane. The user can close a specific file by opening the tab associated with it and clicking the close button.

4.4 User testing

User testing was performed for the GUI with a total of 5 tests performed. These tests are explained in detail in Section 3.4.1. The results have been used to evaluate the effectiveness of certain GUI elements and some feedback has been applied to the GUI itself. An example of this is the selection of breakpoints, the process of selecting a breakpoint was previously to hold the control key of the keyboard and make a selection in the source code view. Now it is also possible to make just a selection and press the *Add breakpoint* button. This is more intuitive for a new user.

The process of starting, stepping, continuing, and stopping was seen as natural. This is because the design adheres to the “safe exploration” pattern from [34] which states that the user should be able to treat an application as they expect from its purpose. In the case of a debugger, it is designed similarly to other graphical debuggers which provide an expected workflow to a user with previous debugging experience.

The user testing proved that our application seemed to be successful with its purpose of displaying the heap and an evaluation process. The testees were overall satisfied with the program with a mean of 3 out of 5 if they would recommend the program when asked to score from 1 to 5.

In detail the testees found the GUI to behave as expected with the exception of breakpoint selection. The help frame was particularly appreciated by the testees as it provided accurate information about the colouring of the graph, breakpoint selection, and what the different output panes represent.

The testees thought there was a lack of pliancy for operations. This was denoted for successful selection of a breakpoint and selecting a binary. The testees could not know immediately if a breakpoint was selected or if a binary was loaded successfully into the program.

The graph itself was often not understood by the testees but after an explanation by the help frame they found it to be intuitive. One testee suggested displaying text panes as an option to the graph since it could be unintuitive for a new Haskell programmer. However, the main persona is an experienced Haskell programmer and would already know the reasoning why it is displayed as a graph so the testee’s suggestion was just considered briefly.

As a direct result of user testing a placeholder image with instructions was created and displayed whenever no debug process is active. This image can be seen in Figure 4.12. It contains basic instructions such as how to start debugging, step, continue, and stopping the process. Additionally, it informs the user that the debug process will try to automatically attempt to open files and that there is a help dialog with more information.

5
Discussion

5.1 Method Discussion

In this section we try to evaluate how the development transpired, if our initial choices were good and if there are any improvements that could have been made.

5.1.1 Development Process

The development process generally worked well. Scrum worked as intended and we had a product to show our stakeholder at the end of every sprint. Some changes had to made during the process regardless, specifically the amount of meetings were decreased in order to increase efficiency. Another change is that the project started with a quite open approach to the agile process but it was made more strict towards the end to ensure focus was kept on development.

5.1.2 Haskell Library Documentation

As previously mentioned our debugger uses the GTK [35] framework, and GTK has two different major bindings to Haskell. Our application started out with the gtk3 library [25]. At one point in development, it was discovered that it was considerably more difficult to find documentation for this library as it was not part of the standard search on Hoogle [36]. The other alternative, haskell-gi [37], had documentation available on Hoogle. However, at the time of this discovery, development had already begun and a switch was deemed detrimental to the progress. As these libraries were based upon GTK then it was also possible to use the original GTK documentation so this was used instead despite the functions not always being the same.

5.1.3 Other possible GUI approaches

Other approaches were considered regarding the type of debugger that was developed.

The first alternative approach would be to display the graph through a Javascript web browser interface. In addition to the GUI, a server would be running, connecting HDB with the user interface. Our base idea was developing as an actual web page rather than, for example, an Electron [38] application and this could be deemed unusual from a user's point of view.

The second option would be to implement it like a textual user interface (TUI), structuring the graph through ncurses [39] with a minimalistic TUI. The downside of this approach is that it was deemed that the program would have been less accessible to novice users.

The last considered approach was to develop it as a plugin in a text editor like VSCode. This was not chosen as only a single group member used VSCode and it was thought we could gain more Haskell knowledge by developing an standalone application.

The final result, a desktop application using a GTK binding was deemed as a sufficient choice as it provided an easy way to connect with the HDB library. It also enabled us to learn more about Haskell which was a common goal of the group members. GTK is also widely used in the GNU/Linux environment and is extensively documented. The downside of this approach is that the user has to open a different application in order to debug their desired program. If the debugger was integrated into an editor this disruption of the work flow would not be necessary.

5.1.4 Testing

When the complexity of the software grew it became apparent that the lack of integration testing impeded the speed of development. When some new features were added, other seemingly unrelated parts of the program stopped working. As the program used a multitude of shared mutable data these bugs could be hard to track down. Later in the development, a more systematic approach to integration testing of the GUI was adapted to rectify this.

Unit and property-based testing was meant to be performed for the parts of the program that was suitable. However, almost all development was GUI development and thus not suitable for property based testing. At the moment only one of the tests of the graph implementation uses QuickCheck. This is a function that generates random words that is be converted to chars.

5.1.5 User testing

While the user testing of our debugger proved that the GUI turned out functional the testing itself had potential issues. The tests were performed quite late in the development process which meant that the feedback received could be used minimally during development. Instead, the user testing results were regarded as an evaluation of our product from an external audience, as seen in Section 4.4.

All tests were performed by a certain group, IT or computer science students that were beginner Haskell programmers. This group consists of members from our secondary persona, meaning that no user tests were performed with any from the main target audience. This can skew the results since it can be difficult to gain information about some tasks without adequate Haskell knowledge. For example, if the program can be used to profile and how accurate the heap visualisation appears to be to someone outside of the project team.

Another potential issue is that there have been relatively few user tests performed.

The results given by these tests were valuable for evaluating the debugger so we consider them useful regardless, but a larger sample size consisting of several different groups could give more accurate feedback.

5.2 Result Discussion

The application will be analysed according to the purpose of the project; if it is possible to debug compiled Haskell programs, if it can be used as a teaching tool, and additionally if it is possible to use for profiling.

5.2.1 Debugging

Whilst the debugging process is provided by the backend, HDB, our application provides a way of displaying it and handling it that is easy to use for an average user. As described in Section 4.4, we found that a majority of the respondents thought that the application could be a useful tool to debug Haskell programs. However, in order to be able to efficiently debug a program, the user must be able to know exactly what the graph displays and what is of relevance to the problem they have.

Since breakpoints are defined per evaluation the user needs to be aware of what will be evaluated in a compiled program. If the compiler inlines an evaluation with a breakpoint there is no way for the user to be able to stop at that breakpoint. The lack of pliancy when choosing a breakpoint span means that the user is not alerted whenever an invalid span is chosen. The only way for the user to notice that, is to run the program to completion and see that the execution does not stop at a selected breakpoint.

Overall we evaluate the debugging process to be natural and like other debuggers with the deficit of a need to have knowledge about Haskell evaluation to be able to fully utilise the debugger.

5.2.2 The Graphical User Interface

As perviously mentioned in Section 4.4, the GUI is constructed according to the "safe exploration" pattern. To be able to provide an environment where the user can feel safe and comfortable we have taken inspiration from other debuggers that the group members have previous experience using.

Parts that we found most debuggers utilised were a source code window on the right side, a thinner pane with output, stack trace, and other information towards the bottom, and a large pane with a file tree on the left. Due to the way we visualise the evaluation process the file tree was replaced by the visualisation.

Other similarities are that when the evaluation moves to another file, this file is automatically opened and focused in the tabbed source file pane. The source span for the relevant evaluation is then highlighted in the pane. This is in contrast to most debuggers for programs in imperative languages, where the line number would be marked instead of a source span.

Because the GUI uses standard GTK theming the program will look differently depending on what theme is specified. This can cause issues for users as some themes may have parts that are more difficult to see. For example the theme used for the pictures in Chapter 4, Section 4.3 are using a grey theme that, according to user tests, makes it hard to see whether a tool button is active or not.

As seen in Section 4.4 setting breakpoints is not intuitive for a new user. To select a span as a breakpoint is a requirement given that you mark a specific evaluation. However, selecting a span which actually corresponds to an evaluation is hard to know. To improve this it could be possible to highlight a span differently if there is a linkername corresponding to this span. Finding a linkername through a span can be done through HDB's interface.

There is also a menu item for settings. Currently there is only one option: setting a maximum node depth. We think having this as a setting in the menu is a good choice as it does not update the graph instantly, it is required to generate a new graph with the new node depth. If this option was located by the other tool buttons a user might think that it does nothing when changed as it is not possible to receive immediate feedback.

5.2.3 The Visual Representation of the Heap

In the earlier stages of development other more conventional approaches of relaying information about the program's state were discussed, for example reconstruction of data types. This would mean that an object could be constructed from several closures. An example of this would be a list. On the heap, a list would consists of several closures but it would be possible to recreate it from the type information from the closures. This was deemed as an interesting feature but the problem with this abstraction is that granularity is lost. In general it is difficult to determine individual users' needs when it comes to the teaching potential of the program but the decision was made to prioritise an accurate representation of the GHC-heap before simplifications. This was done to ensure that every user can utilise the program, even if it might be more inaccessible to novice users.

At the same time, it is important to mention that the way the heap is modelled in the program is an abstraction. The decision was made to hide certain aspects of it from the user. For example, an MVar holds references to other threads that are waiting to use it. Since the debugger only works for single threaded programs it was not deemed necessary to show the thread state object that is pointed to. If the user wants a more accurate representation of the heap, that option exists by setting the compilation flag `SIMPLIFIED` to false.

A consequence of lazy evaluation and that all variables are immutable means that it does not make sense to follow a specific variable during the execution like for an imperative language. As an alternative to this, our debugger displays the evaluation in terms of the current state of the heap and stack for the current evaluation.

By utilising a graph with different colours to define the heap, the stack, and the current evaluation it is possible to trace the evaluation in a way that does not require knowledge of which order certain lines of code will be executed. The graph accurately shows the evaluation but it may be difficult for a new Haskell developer to know what the different terms mean. Sometimes the number of pointers between the different nodes can also get in the way of another which results in a cluttered graph that might be hard to read.

When using the debugger and reading the graph we came to realise that it can at times be difficult to locate the current node when the graph is generated. The cause of this is that the graph changes width and height so the centre keeps changing. This was solved by always displaying the currently evaluated node at the centre of the graph pane. Similarly to reduce the graph's size there is a menu option to set the maximum node depth of the graph, thus decreasing or increasing the size depending on the user's needs.

The decision to mark all the nodes that are derived from the stack trace as yellow might also cause some confusion to the user since it might not be intuitive. This is especially true since the nodes coloured yellow can be shown when the stack is hidden. It is not apparent to the authors that this solution is worse than any other solution. Another idea was to colour only the stack objects yellow but this results in a graph where it might be even harder to discern the nodes that are associated with the currently evaluated node.

We find the representation to be easy to use and follow, and the way it is displayed and interacted with to be natural. But as the user tests were lacking in experienced respondents we cannot confirm whether someone more experienced would appreciate the graph in the same way.

5.2.4 Performance

While developing the application there were multiple iterations of the program that had significantly worse performance than others. Most notably there were occasions where the CPU usage would spike or the RAM usage would eternally increase.

These bugs in particular depended on a GTK callback which looped eternally and a list of SVG file data that was added to at every debug step. These were solved by having the callback activate only on a change and by limiting the amount of data able to be collected at a debug step.

The application itself is built with the GTK framework and there is some lag when the application window is resized. No source have been located for this issue but the program does not crash and it does not affect the usage of the application so it is not deemed as something that needs to be corrected. But it may defer some users as the application does not always feel "snappy".

5.2.5 Stability

The integration testers of the program consist of the development team. As the testing has been done at the same time as development, and no member has had the sole purpose of integration testing, we can not guarantee the stability of the program completely. It works for the intended purposes: debugging a compiled Haskell program, viewing the graph, and interacting with the debug process.

One limitation to the stability is that there is no control over what kind of executable is loaded into the program. The file dialog from GTK is using Media Types [40] to filter the selections and currently media types cannot differ between GHC generated executables compared to executables generated elsewhere. So it is possible to load a binary that is not accepted by HDB. Doing this may cause the debugger to not function properly as HDB is not capable of interacting with these in the same way as GHC generated executables. There is a warning provided if the debugger can not find any Haskell source files so the user is aware that there might be issues continuing the debug process.

There is also no confirmation whether a program is multi-threaded or not. As HDB does not support multi-threaded programs it may impact the debugging experience in unpredictable ways.

5.3 The Ethics of Debugging

A debugger can be used to reverse engineer compiled programs since the information that a debugger provides could enable a reconstruction of the original software. To do this with our debugger the user would need to compile the program without using tools other than GHC itself, for example with the command: `ghc -g Main.hs -o Main`. This would produce an executable `Main` from a source code file `Main.hs`.

The creator of a program can hide the heap representation by removing debug information. This can be done using external tools, or Cabal by enabling the `-enable-library-stripping` and `-enable-executable-stripping` flags [41].

In practice, most programs are built using tools such as Cabal so it would be easy for a developer to remove debug information if they so desire. Reconstructing a program without accurate heap information is considerably more difficult than reconstructing a program with it. Therefore the possibility of reverse engineering a compiled program with our debugger is not deemed a substantial ethical issue.

5.4 Future Work

If there was additional time available for development we would like to focus on providing a simplified view of the graph, like the one in ghc-vis [3], and creating an interactable graph.

Additional valuable information that the GHC would provide during compilation could be added to the debugger using the `-dump` flags for the STG and Cmm output phases. Although we have not yet used the GHC as a library, it is very likely that it will help us in future work in implementing new features that would meet the needs of lower-level debugging. While we were planning future development, we were thinking about publishing the debugging GUI. If it turns out that there is interest in using the program, and this could be the case due to the lack of a debugger for compiled programs in Haskell, it is likely that program development with new developers will be faster than now.

If we compare our development approach in a small development team's closed environment with the cathedral in Eric S. Raymond's famous "The cathedral and the Bazaar" [42] then an open bazaar and a permissive license might give new life to the future development of our debugger.

There are some ideas that we thought of during development but were never able to act upon. The two most valuable of those ideas would be to parse the DWARF data of an executable to locate row and column ranges that would correspond to a breakpoint in the HDB library. It is however possible to use HDB for this, by sending in the span of an entire file and using the function `findFunction` it is possible to get the functions in that file. Then `findSource` can be used to find files and spans to mark in the GUI. By using this the GUI would greatly simplify the breakpoint selection process for the user.

The other idea is to have an interactable graph. For example blacklisting a certain node type, removing a sub-tree, or displaying the source code which generated a certain node all within the graph image. There was an attempt at implementing this but this happened at a phase of the project where it was much more important to focus on report writing and removing bugs from the product so it was put aside. Still, the way the code is structured and since SVG-parsing functionally has already been written, these features would likely be relatively simple to implement.

6 Conclusion

There are three main purposes of the project:

1. To create a graphical debugger that uses HDB, a debugging library for compiled Haskell programs.
2. To be able to visualise the heap of compiled program execution in a manner that is easy to understand and simple enough to learn from.
3. And to be able to use the visualisation for profiling, i.e. analysing the program's execution in an attempt to optimise it.

A graphical interface has been created and it is structured in a way that there are no dependencies from the debug process of HDB and our graph displaying functionality to the GUI. This is deemed satisfactory since the processes are weakly coupled together. During user testing of the GUI we discovered that most users were satisfied with it, but there are some non-standard operations that were less intuitive. Therefore a "help" window is provided with accurate instructions on how to perform unintuitive tasks.

The heap is visualised as a graph and the internal names of nodes are simplified for ease of reading. However, if the graph itself was simplified further the representation would not be as accurate. This might be a trade-off to some users in terms of the program's usability as a teaching tool compared to profiling. A more realistic representation will provide more accurate information for profiling while it may be too large or cluttered to be used as a teaching tool for new Haskell programmers. Enabling the stack trace to be hidden or displayed as well as making it possible to limit maximum node depth is a middle ground that can enable both to some extent.

Overall the project's purposes have been fulfilled to a satisfactory level but there are improvements that can be made.

Bibliography

[1] W. E. Wong, V. Debroy, A. Surampudi, H. Kim, and M. F. Siok, *Recent Catastrophic Accidents: Investigating How Software was Responsible.* 2010, pp. 14–22. DOI: `10.1109/SSIRI.2010.38`.

[2] P. Kidwell, "Stalking the elusive computer bug," *IEEE Annals of the History of Computing*, vol. 20, no. 4, p. 1, 1998. DOI: `10.1109/85.728224`.

[3] D. Felsing, "Visualization of lazy evaluation and sharing," Bachelor's Thesis, Karlsruhe Institute of Technology, Germany, Sep. 2013.

[4] K. Angelov. "HDB." (2021), [Online]. Available: `https://github.com/krangelov/hdb` (visited on 2022-02-03).

[5] "The DWARF Debugging Standard," DWARF Standards Committee. (2021), [Online]. Available: `https://dwarfstd.org/` (visited on 2022-02-06).

[6] K. Angelov. "DATX02-22-03 A Debugger for Haskell." (2021), [Online]. Available: `https://chalmers.instructure.com/courses/17746/files/1710124` (visited on 2022-01-30).

[7] U. Drepper, R. McGrath, and P. Machata. "ELFUTILS." (2021), [Online]. Available: `https://sourceware.org/elfutils/` (visited on 2022-04-16).

[8] S. Marlow. "Haskell 2010 Language Report." (2022), [Online]. Available: `https://www.haskell.org/onlinereport/haskell2010/` (visited on 2022-02-07).

[9] "DWARFFAQ," DWARF Standards Committee. (2017), [Online]. Available: `http://wiki.dwarfstd.org/index.php?title=DWARF_FAQ` (visited on 2022-04-16).

[10] "ghc-heap: Functions for walking GHC's heap." (2022), [Online]. Available: `https://hackage.haskell.org/package/ghc-heap` (visited on 2022-05-12).

[11] B. Gamari. "DWARF support in GHC." (2020), [Online]. Available: `https://www.haskell.org/ghc/blog/20200403-dwarf-1.html` (visited on 2022-04-16).

[12] S. Marlow and S. L. P. Jones. "The glasgow haskell compiler, the architecture of open source applications, volume 2 draft chapter." (2012), [Online]. Available: `https://www.aosabook.org/en/ghc.html` (visited on 2022-04-25), licensed under CC BY 3.0.

[13] "Haskell pragmas," Haskell.org. (2022), [Online]. Available: `https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pragmas.html` (visited on 2022-02-11).

[14] S. Marlow and S. L. P. Jones, "The new GHC/Hugs runtime system," 1998.

[15] S. Peyton Jones, S. Marlow, and A. Reid, "The STG runtime system (revised)," Jan. 1999.

[16] P. Jones, S. L, and S. Peyton Jones, "Implementing lazy functional languages on stock hardware: The spineless tagless g-machine," *Journal of Functional Programming*, vol. 2, pp. 127–202, Jul. 1992. [Online]. Available: `https://www.microsoft.com/en-us/research/publication/implementing-lazy-functional-languages-on-stock-hardware-the-spineless-tagless-g-machine/`.

[17] S. Marlow, S. Peyton Jones, and S. Singh, "Runtime support for multicore haskell," in *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '09, Edinburgh, Scotland: Association for Computing Machinery, 2009, pp. 65–78, ISBN: 9781605583327. DOI: `10.1145/1596550.1596563`. [Online]. Available: `https://doi.org/10.1145/1596550.1596563`.

[18] S. Peyton Jones, "How to make a fast curry: Push/enter vs eval/apply," in *International Conference on Functional Programming*, Sep. 2004, pp. 4–15. [Online]. Available: `https://www.microsoft.com/en-us/research/publication/make-fast-curry-pushenter-vs-evalapply/`.

[19] S. Marlow, J. Iborra, B. Pope, and A. Gill, "A lightweight interactive debugger for haskell," in *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, ser. Haskell '07, Freiburg, Germany: Association for Computing Machinery, 2007, pp. 13–24, ISBN: 9781595936745. DOI: `10.1145/1291201.1291204`. [Online]. Available: `https://doi.org/10.1145/1291201.1291204`.

[20] A. Gill, "Debugging haskell by observing intermediate data structures," in *University of Nottingham*, Electronic, 2000.

[21] C. Reinke, "Ghood – graphical visualisation and animation of haskell object observations," in *ACM SIGPLAN Haskell Workshop, Firenze, Italy*, R. Hinze, Ed., ser. Electronic Notes in Theoretical Computer Science, Preliminary Proceedings have appeared as Technical Report UU-CS-2001-23, Institute of Information and Computing Sciences, Utrecht University. Final proceedings to appear in ENTCS., vol. 59, Elsevier Science, Sep. 2001. [Online]. Available: `https://kar.kent.ac.uk/13558/`.

[22] "GDB: The GNU project debugger," Free Software Foundation. (2022), [Online]. Available: `https://www.sourceware.org/gdb/` (visited on 2022-04-24).

[23] "Trello," Trello Inc. (2022), [Online]. Available: `https://trello.com/sv` (visited on 2022-05-08).

[24] "Git," The Git community. (2022), [Online]. Available: `https://git-scm.com/` (visited on 2022-02-07).

[25] "gtk2hs," gtk2hs. (2022), [Online]. Available: `https://github.com/gtk2hs/gtk2hs` (visited on 2022-04-15).

[26] "Graphviz." (2021), [Online]. Available: `https://graphviz.org/` (visited on 2022-05-12).

[27] "Dot." (2022), [Online]. Available: `https://graphviz.org/doc/info/lang.html` (visited on 2022-05-12).

[28] "graphviz: Bindings to Graphviz for graph visualisation." (2020), [Online]. Available: `https://hackage.haskell.org/package/graphviz` (visited on 2022-05-12).

[29] A. Cooper, *About Face : The Essentials of Interaction Design*, 4th ed. Somerset, NY, USA: John Wiley & Sons, Incorporated, 2014. DOI: `10.5555/2688796`.

[30] B. Martin and B. Hanington, *Universal Methods of Design : 100 Ways to Research Complex Problems, Develop Innovative Ideas, and Design Effective Solutions*. Rockport Publishers, an imprint of The Quarto Group, 2012.

[31] K. Claessen and J. Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '00, New York, NY, USA: Association for Computing Machinery, 2000, pp. 268–279, ISBN: 1581132026. DOI: `10.1145/351240.351266`. [Online]. Available: `https://doi.org/10.1145/351240.351266`.

[32] GHC Team and J. Dagit. "zenc: GHC style name Z-encoding and Z-decoding." (2021), [Online]. Available: `https://hackage.haskell.org/package/zenc-0.1.2` (visited on 2022-04-27).

[33] "Tyingtheknot," Haskell.org. (2021), [Online]. Available: `https://wiki.haskell.org/Tying_the_Knot` (visited on 2022-05-11).

[34] J. Tidwell, *Design Interfaces: Patters for Effective Interaction Design*, 2nd ed. O'Reilly Media, Inc, 2010.

[35] "GTK," The GTK Team. (2022), [Online]. Available: `https://www.gtk.org/` (visited on 2022-02-07).

[36] "Hoogle," Haskell.org. (2022), [Online]. Available: `https://hoogle.haskell.org` (visited on 2022-04-15).

[37] W. Thompson and I. G. Etxebarria. "haskell-gi." (2021), [Online]. Available: `https://github.com/haskell-gi/haskell-gi` (visited on 2022-04-15).

[38] "Electron," OpenJS Foundation. (2022), [Online]. Available: `https://www.electronjs.org` (visited on 2022-04-17).

[39] T. E. Dickey. "NCURSES – New Curses." (2022), [Online]. Available: `https://invisible-island.net/ncurses/` (visited on 2022-04-17).

[40] "Media Types," Internet Assigned Numbers Authority. (2022), [Online]. Available: `https://www.iana.org/assignments/media-types/media-types.xhtml` (visited on 2022-05-08).

[41] "dwarf," Haskell.org. (2020), [Online]. Available: `https://gitlab.haskell.org/ghc/ghc/-/wikis/dwarf` (visited on 2022-02-18).

[42] E. S. Raymond, "The cathedral and the bazaar," *Knowledge, Technology & Policy*, vol. 12, no. 3, pp. 23–49, 1999.