Department of Computer Science
OST – University of Applied Sciences
Campus Rapperswil-Jona

Autumn Term 2021

Haskell Substitution Stepper

Bachelor thesis: project documentation

Autors:	Dominik Dietler Robin Elvedi Jan Huber
Version:	January 14, 2022
Advisor:	Prof. Dr. Farhad D. Mehta
External Co-Examiner:	Jasper Van der Jeugt
Internal Co-Examiner:	Prof. Dr. Daniel Patrick Politze

Notice

This is the public version of the report. It does not contain the project plan, legal documents, personal statements and meeting minutes.

Abstract

Functional programming languages such as Haskell are fundamentally different from imperative languages. While Haskell code is very expressive and elegant, the language involves a steep learning curve. Concepts like higher-order functions, lazy evaluation and recursion are difficult to teach. Moreover, debugging a functional language is more difficult due to a less intuitive execution model compared to imperative languages.

This thesis presents the implementation of the „Haskell Substitution Stepper“, a tool aiming to address those issues. The goal is to create an environment in which developers can analyze Haskell expressions. To achieve this, the evaluation of those expressions is done step by step, and an explanation for each reduction is given. Such step by step reductions are often printed in textbooks to help understand a small program.

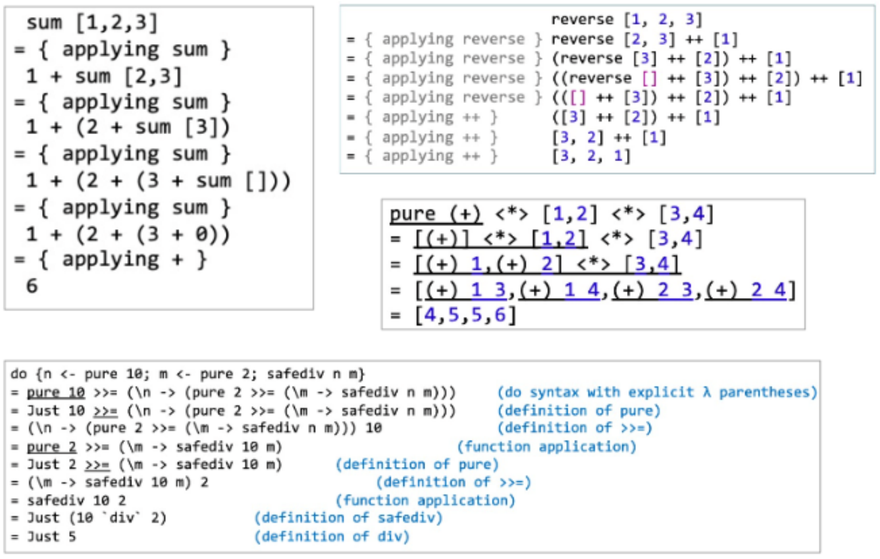
The first step was to research how the Glasgow Haskell Compiler (GHC) transforms and evaluates expressions in the backend. Existing similar solutions were assessed and ideas for improvement were collected. Finally, the „Haskell Substitution Stepper“ was implemented using the intermediate representation „Core“ from GHC.

The result is a command line application. Developers can load a source file with Haskell expressions into the stepper and see the individual reduction steps. The tool can be used for debugging or to better understand specific Haskell expressions. While the stepper is using the Core intermediate representation for the reduction, the result can be pretty printed in a format that looks like regular Haskell code.

Management Summary

Goal

In the imperative programming paradigm, a debugging tool with an appropriate visualization of the program counter and internal state is often used as an aid to visualize and learn about program execution. The functional programming paradigm does not have the concept of a program counter or internal state. Executing a program in a functional programming language is typically viewed as evaluating an expression using repeated substitution, as seen in the image below.



```
sum [1,2,3]
= { applying sum }
 1 + sum [2,3]
= { applying sum }
 1 + (2 + sum [3])
= { applying sum }
 1 + (2 + (3 + sum []))
= { applying sum }
 1 + (2 + (3 + 0))
= { applying + }
 6
```

```
                          reverse [1, 2, 3]
= { applying reverse } reverse [2, 3] ++ [1]
= { applying reverse } (reverse [3] ++ [2]) ++ [1]
= { applying reverse } ((reverse [] ++ [3]) ++ [2]) ++ [1]
= { applying reverse } (([] ++ [3]) ++ [2]) ++ [1]
= { applying ++ }      ([3] ++ [2]) ++ [1]
= { applying ++ }      [3, 2] ++ [1]
= { applying ++ }      [3, 2, 1]
```

```
pure (+) <*> [1,2] <*> [3,4]
= [(+)] <*> [1,2] <*> [3,4]
= [(+) 1,(+) 2] <*> [3,4]
= [(+) 1 3,(+) 1 4,(+) 2 3,(+) 2 4]
= [4,5,5,6]
```

```
do {n <- pure 10; m <- pure 2; safediv n m}
= pure 10 >>= (\n -> (pure 2 >>= (\m -> safediv n m)))     (do syntax with explicit λ parentheses)
= Just 10 >>= (\n -> (pure 2 >>= (\m -> safediv n m)))     (definition of pure)
= (\n -> (pure 2 >>= (\m -> safediv n m))) 10              (definition of >>=)
= pure 2 >>= (\m -> safediv 10 m)                      (function application)
= Just 2 >>= (\m -> safediv 10 m)       (definition of pure)
= (\m -> safediv 10 m) 2                     (definition of >>=)
= safediv 10 2                          (function application)
= Just (10 `div` 2)           (definition of safediv)
= Just 5                      (definition of div)
```

Figure 1: Haskell substitutions

Although derivations such as those listed are used when teaching functional programming and reasoning about functional programs, there is no tool support for automatically generating such derivations. Having such tool support could greatly help learning programming in and debugging programs written in the functional style. Hence, the main aim of this project is to implement a substitution stepper for the functional programming language Haskell that can be used to visualize the execution of a functional program.

Implementation

The first Proof of Concept was built on a simplified subset of the Haskell AST. Further research into the Glasgow Haskell Compiler and discussions with Haskell experts showed that it is more feasible to work with GHCs intermediate language Core than with the original Haskell. This switch also enabled a closer coupling to GHC, and the output of the resulting prototype proved to be more readable and user-friendly.

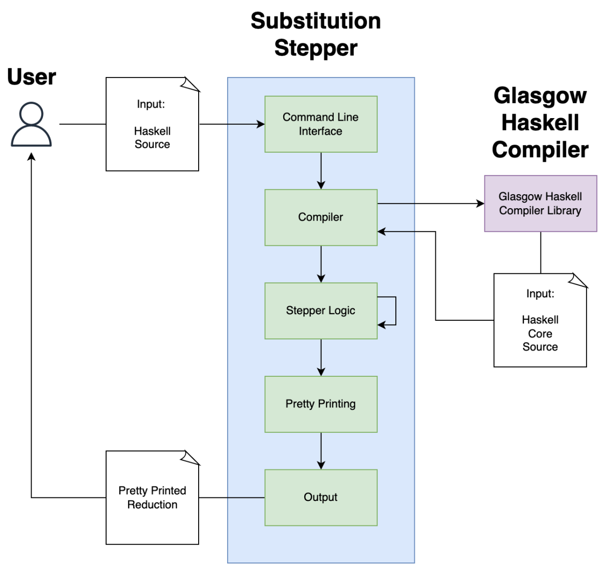


Figure 2: Substitution Stepper architecture

Conclusion

The result of this project is a command line tool, that can successfully step through most Haskell programs and produces outputs that closely resemble the examples given in the task description. In comparison to similar, previously existing tools, the Haskell Substitution Stepper supports a larger part of Haskell and is more closely coupled with the Glasgow Haskell Compiler.

Example Input Expression: `reverse [1, 2, 3]`

Manual Reduction (Textbook)

```
= reverse [2, 3] ++ [1]
= (reverse [3] ++ [2]) ++ [1]
= ((reverse [] ++ [3]) ++ [2]) ++ [1]
= (([] ++ [3]) ++ [2]) ++ [1]
= ([3] + [2]) ++ [1]
= [3, 2] ++ [1]
= [3, 2, 1]
```

Automatic Reduction (Stepper)

```
= (reverse [2, 3]) ++ [1]
= 3 : (([] ++ [2]) ++ [1])
= [3, 2, 1]
```

Example Input Expression: `do {n <- pure 10; m <- pure 2; safediv n m}`

Manual Reduction (Textbook)

```
= pure 10 >>= (\n -> (pure 2 >>= (\m -> safediv n m)))
= Just 10 >>= (\n -> (pure 2 >>= (\m -> safediv n m)))
= (\n -> (pure 2 >>= (\m -> safediv n m))) 10
= pure 2 >>= (\m -> safediv 10 m)
= Just 2 >>= (\m -> safediv 10 m)
= (\m -> safediv 10 m) 2
= safediv 10 2
= Just (10 `div` 2)
= Just 5
```

Automatic Reduction (Stepper)

```
= (pure 10) >>= (\n -> (pure 2) >>= (\m -> safeDiv n m))
= (\n -> (pure 2) >>= (\m -> safeDiv n m)) 10
= (\m -> safeDiv 10 m) 2
= Just (div 10 2)
= Just 5
```

Figure 3: Substitution Stepper output compared to textbook

Contents

List of Figures

List of Tables

List of Listings

Abbreviations

API	Application Programming Interface
ASM	Assembly
AST	Abstract Syntax Tree
CI	Continuous Integration
CMM	Haskell C– implementation
GHC	Glasgow Haskell Compiler
GHCi	Glasgow Haskell Compiler's interactive environment
IO	Input/Output
MVP	Minimal Valuable Product
NF	Normal Form
RUP	Rational Unified Process
STG	Spineless Tagless G-machine
UI	User Interface
UML	Unified Modeling Language
WHNF	Weak Head Normal Form

1 Task Description

The following pages contain the original task description.

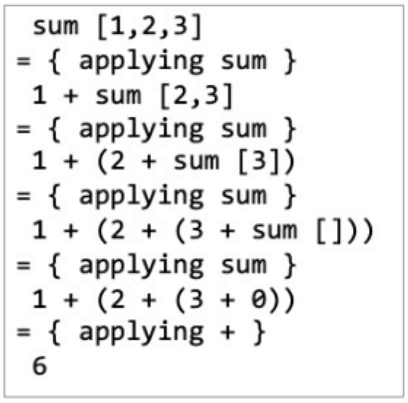
Haskell Substitution Stepper

Task Description

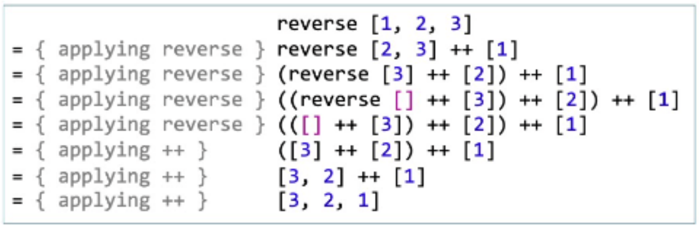
1. Setting

In the imperative programming paradigm, a debugging tool with an appropriate visualisation of the program counter and internal state is often used as an aid to visualise and learn about program execution. Such tools enjoy widespread use by beginners wanting to learn how the paradigm works, as well as professional programmers trying to find causes for unexpected program behaviour.

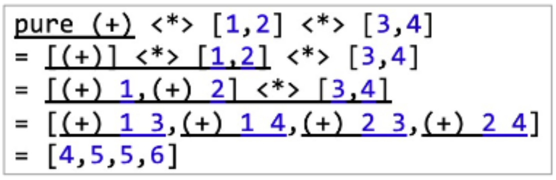
The functional programming paradigm does not have the concept of a program counter or internal state. Other techniques are therefore required to visualise the execution of functional programs. Executing a program in a functional programming language is typically viewed as evaluating an expression using repeated substitution:



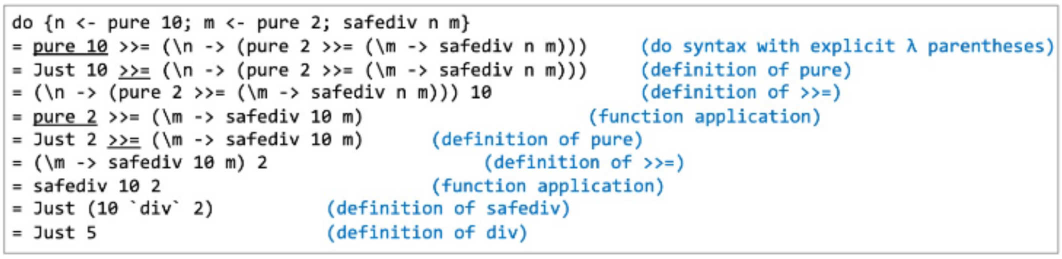
```
 sum [1,2,3]
= { applying sum }
 1 + sum [2,3]
= { applying sum }
 1 + (2 + sum [3])
= { applying sum }
 1 + (2 + (3 + sum []))
= { applying sum }
 1 + (2 + (3 + 0))
= { applying + }
 6
```



```
                       reverse [1, 2, 3]
= { applying reverse } reverse [2, 3] ++ [1]
= { applying reverse } (reverse [3] ++ [2]) ++ [1]
= { applying reverse } ((reverse [] ++ [3]) ++ [2]) ++ [1]
= { applying reverse } (([] ++ [3]) ++ [2]) ++ [1]
= { applying ++ }      ([3] ++ [2]) ++ [1]
= { applying ++ }      [3, 2] ++ [1]
= { applying ++ }      [3, 2, 1]
```



```
pure (+) <*> [1,2] <*> [3,4]
= [(+)] <*> [1,2] <*> [3,4]
= [(+) 1,(+) 2] <*> [3,4]
= [(+) 1 3,(+) 1 4,(+) 2 3,(+) 2 4]
= [4,5,5,6]
```



```
do {n <- pure 10; m <- pure 2; safediv n m}
= pure 10 >>= (\n -> (pure 2 >>= (\m -> safediv n m)))     (do syntax with explicit λ parentheses)
= Just 10 >>= (\n -> (pure 2 >>= (\m -> safediv n m)))     (definition of pure)
= (\n -> (pure 2 >>= (\m -> safediv n m))) 10              (definition of >>=)
= pure 2 >>= (\m -> safediv 10 m)                   (function application)
= Just 2 >>= (\m -> safediv 10 m)       (definition of pure)
= (\m -> safediv 10 m) 2                      (definition of >>=)
= safediv 10 2                          (function application)
= Just (10 `div` 2)            (definition of safediv)
= Just 5                       (definition of div)
```

Although derivations such as these are used when teaching functional programming and reasoning about functional programs, there is no tool support for automatically generating such derivations. Having tool support for generating such derivations could greatly help learning programming in and debugging programs written in the functional style.

2. Goals

The main aim of this project is to implement a substitution stepper (HaskSubStep) for the functional programming language Haskell that can be used to visualise the execution of a functional program.

The following is a brief and unstructured list of initial requirements:

1. Input:
   a. A program context containing existing definitions, types, etc.
   b. The target expression/program to start the derivation.
2. Output: A stepwise derivation of the target expression to its normal form.
   a. Each step of the derivation must be a valid Haskell expression.
   b. If possible, each step must be justified in an appropriate manner (e.g., an indication of the substitution rule used)
3. The user may choose to advance the derivation or fold parts of the derivation in a controlled manner (e.g., compress or step over derivation steps related to certain trivial functions)
4. The standard application programming interface (API) of the Glasgow Haskell Compiler (GHC) should be used as far as possible in order to reuse existing functionality and allow the tool to remain usable with future versions of GHC with minimal porting effort.
5. The tool must be usable for a beginner with little experience in functional programming. Its user interface must be carefully designed to this end. For instance, the tool should allow hiding parts of large expressions that remain the same, or allow highlighting subexpressions that change between steps.
6. In the likely case where a graphical user interface is required, a plugin to the Haskell Language Server VS Code Plugin should be considered.
7. Regardless of the choice of user interface, an open API and/or command line interface (CLI) should be provided to allow the tool to be open to additional user interfaces.
8. The tool will be released under the GPLv3 license.

Since there is currently no existing tool that offers such functionality, further refinements and modifications to this list that serve the main aim of this project are possible during its course.

3. Deliverables

- Product documentation in English that is relevant to the use and further development of the tool (e.g., requirements, domain model, architecture description, code documentation, user manuals, etc.) in a form that can be developed further with the product and is amenable to version control (e.g., LaTeX or Markdown).
- Project documentation that is separate from the product documentation that documents information that is only relevant to the current project (e.g., project plan, time reports, meeting minutes, personal statements, etc.).
- Additional documents as required by the department (e.g., poster, abstract, presentation, etc.)
- Any other artefacts created during the execution of this project.

All deliverables may be submitted in digital form.

2 Introduction

2.1 About this Document

This document is the project documentation for the bachelor thesis. It targets readers who are not only interested in the final product but also in the process of building the Haskell Substitution Stepper. This document contains the project plan, design decisions, problems, and solutions.

The product itself is documented on GitLab Pages and targets the users (product documentation). It contains a guide on how to install and use the substitution stepper:
`http://haskell-substitution-stepper.pages.gitlab.ost.ch/product-docu/index.html`

2.2 Links

GitLab Repositories (private access)

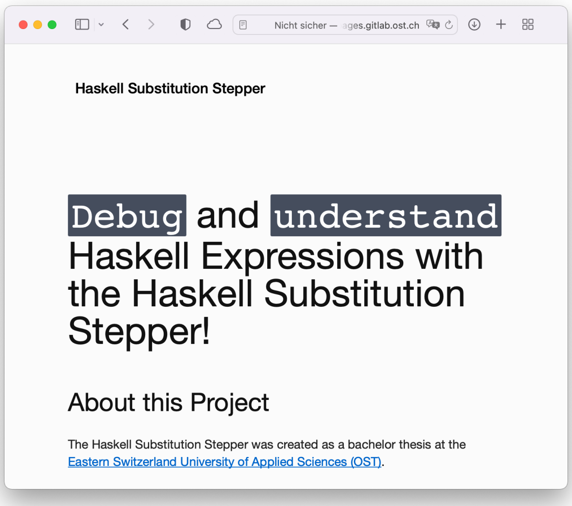
`https://gitlab.ost.ch/haskell-substitution-stepper`

Jira (private access)

`https://hasksubstep.atlassian.net/jira/software/projects/HAS/boards/1`

Product Documentation (public, OST-internal access)

`http://haskell-substitution-stepper.pages.gitlab.ost.ch/product-docu/index.html`



Haskell Substitution Stepper

Debug and understand Haskell Expressions with the Haskell Substitution Stepper!

About this Project

The Haskell Substitution Stepper was created as a bachelor thesis at the Eastern Switzerland University of Applied Sciences (OST).

Figure 4: Product documentation as GitLab Page

2.3 About us

We are a group of three students working on this project:

Dominik Dietler	I find Haskell very interesting, because it is so different to other programming languages. Furthermore I enjoy working with and on Compilers.
Robin Elvedi	Haskell sparked my interest in programming language design and domain-specific languages. I've always wanted to use Haskell within a more practical project and was delighted to be able to use it for our thesis.
Jan Huber	My first experience with Haskell was in a module at OST University of Applied Sciences about programming languages and formal methods. I was fascinated about the idea of functional programming and impressed by the compact and expressive syntax of Haskell.

2.4 Goal and Motivation

Our main goal is to help Haskell programmers understand what their code does. When we ourselves were learning the basics of the language, we realized that Haskell differs in fundamental ways from other well-known programming languages such as Java or C#. As a student, it is often a challenge to tell what a function does by just reading the function body.

In the literature about Haskell, we frequently encountered listings of expressions and their evaluation steps [1]. See the task description in Chapter 1 for an example. Those listings helped us a lot to understand the functions from the Prelude. Seeing the evaluation of an expression step by step is helpful, especially when recursion comes into play. Functions like `foldl` are difficult to explain using only words. They are more intuitively understood when given some practical examples.

We wondered whether it is possible to automatically generate such a stepwise reduction from a given Haskell expression. From a student's perspective, it would certainly be helpful to be able to experiment with self-made examples in this way. Characteristics of Haskell such as higher-order functions, operator precedence and evaluation order could be taught in a more playful manner.

3 Project Plan

Notice

This public version of the document does not contain the full project plan

3.1 Software Requirement Specification

3.1.1 Introduction

The software requirements listed in this chapter are not explicitly stated in the task description. They were listed as additional or implicit goals to create a common ground in the team and to avoid misconceptions.

3.1.2 Functional Requirements

The evaluation of the functional requirements are listed in Chapter 8.

- A user can enter arbitrary Haskell Code into the stepper as text input (not a simplified Version of Haskell).
- A user can provide multiple files at once, input files can reference other files.
- A user can configure the verbosity of the stepper output.
- The stepper explains each reduction step that was taken.
- The stepper follows the rules of laziness and non-strictness.
- A user can work with custom functions, custom types and custom type classes.
- A user can work with functions, types, and type classes from the Standard Haskell Prelude where it is feasible.

3.1.3 Non-Functional Requirements

The evaluation of the non-functional requirements are listed in Chapter 8.

- **Performance**: Generating and Printing the reduction steps for a simple expression like `reverse [1, 2, 3, 4]` does not take any longer than 10 seconds on the development machines.
- **Security**: The software does not use dynamic loading with `eval` to implement the reduction rules.
- **Constraints**: Reductions are implemented in the the Haskell programming language.
- **Installability**: There is a one-command installer for the software.

- **Learnability**: The tool can be used and understood by someone without having read the full user manual.
- **Fault tolerance**: Expressions that cannot be stepped do not result in a crash of the application.
- **Fault tolerance**: Infinite loops during reduction do not result in a crash of the application, instead the reduction is aborted.
- **Maintainability**: The code is structured in a modular way. The user interface can potentially be exchanged without having to change the backend.
- **Modifiability/Changeability**: The backend can be individually configured. Not every configuration option has to be shown to the user.
- **Testability**: The CI pipeline fails if there are failing automated tests.

4 Analysis and Research

4.1 Knowledge Gaps

In the following chapters, we research open questions to provide enough information to outline how a substitution stepper could work. We also assess how realistic and effective the various approaches are.

Our biggest questions are:

- does GHC offer an API to evaluate an expression step by step, or is there a way to pause the execution of the GHC and analyze the intermediate results?
- is there a way that allows us to directly work with the Haskell language instead of a desugared language (Core, STG) or a reduced language such as Duet?

4.1.1 Consulted Experts

To get some feedback about feasibility, we contacted some well-known names in the Haskell community: David Luposchainsky (Creator of stgi[1]), Vladislav Zavialov (GHC contributor), Twan van Laarhoven (Creator of the simple-reflect package[2]), Bastiaan Heeren (Co-Author of the paper ‘Evaluating Haskell expressions in a tutoring environment‘[3]), Steve Krouse (Author of Lamda Bubble Pop[4]) and Stephen Diehl (Software Engineer and Haskell Expert).

[1] `https://github.com/quchen/stgi`
[2] `https://hackage.haskell.org/package/simple-reflect`
[3] `https://arxiv.org/pdf/1412.4879.pdf`
[4] `https://chrisuehlinger.com/LambdaBubblePop/`

4.2 The Haskell Language

4.2.1 Language Characteristics

Creating a stepper for Haskell requires being aware of the language characteristics. Some of the following characteristics (for example non-strict semantics) are a special challenge when building a stepper. Other language features (for example pureness) are an advantage and make the work easier. In this chapter, there are some characteristics listed that have an impact on our work.

4.2.2 Sharing

GHC uses sharing, which means that temporary data is sometimes calculated and stored in memory when it is needed more than once. This happens in different situations. For example, every time a `let` binding is used. In `let k = add 1 2`, the result `k = 3` is stored in memory and used every time `k` is used again. (This would also be the case in many other languages, such as Java). Another example is `where`:

```haskell
x * x
where x = 1 + 2
```

Listing 1: Example of sharing

Here, the value of `x` will also only be calculated once. As a rule of thumb, it can be said that named values will be shared by GHC. Sharing can be tricky as it can lead to space leaks if a programmer is not aware of the behavior. This is the case when a computation is easy, but the result is large [2]. The GHC is rather conservative in using sharing even though sharing is described as a benefit when using pure functions. GHC does not remember every function call: calling the same function with the same arguments usually results in the same computation twice.

To support sharing for the substitution stepper, we would have to keep a representation of the heap during the evaluation. However, in a first version, we ignore that sharing exists and just evaluate some expressions multiple times.

4.2.3 Memoization

Memoization is similar to sharing but described as a programming technique. Memoization can be used by the programmer to store intermediate results instead of calculating them multiple times. This will improve the performance of an application. Knowing when GHC uses sharing allows programmers to optimize their source code for speed.

4.2.4 Non-Strictness

Non-strictness is a feature that distinguishes Haskell from many well-known programming languages.

Technically speaking, non-strict semantics means that a Haskell expression can be rewritten/-substituted even if there are some subexpressions that do not resolve. This is achieved with the outermost-first evaluation strategy: When a function-application is reduced, the arguments are not fully resolved into normal form before using them inside the function. This means that arguments are evaluated only if needed. This is also known as call-by-name strategy [3].

For example, in a language such as Java, an expression like `1.0/0.0` would immediately lead to an exception. However, in Haskell, the compiler would not immediately try to convert the expression into a value. Therefore, there might not be any error if the result of the illegal operation is not used later in the program flow. The following program would terminate without error:

```
Prelude> fst (1.0/2.0, 1.0/0.0)
0.5
```

Listing 2: Example of non-strictness in Haskell

Let's take a look at the definition of the `fst`-function from the prelude[5]:

```
fst :: (a,b) -> a
fst (x,_) =  x
```

Listing 3: Example of an ignored expression

As we see, the second expression in the tuple is simply ignored.

Haskell is not a completely non-strict language. Where required, evaluation of thunks can be enforced using the built-in function `seq`. This function's intent is to offer a way for programmers to force evaluate their arguments because the non-strict evaluation can sometimes lead to performance issues. Pattern matching is also strict, so expressions get evaluated in order to check if there is a matching pattern.

The term laziness is often confused with non-strictness, but it is not exactly the same. While non-strictness is given by the specification of the language, laziness is one (popular) way to evaluate a non-strict language [4].

[5]`https://hackage.haskell.org/package/base-4.15.0.0/docs/Prelude.html`

"Basically laziness == non-strictness + sharing"

(From the Haskell Wiki [5])

As a lazy evaluation can reduce the performance, GHC uses a strictness analysis to evaluate which arguments will always be evaluated by a function and can therefore be evaluated by the caller instead [5].

The concept of non-strictness allows Haskell to have infinite data structures, like `[1..]` which can also be passed to functions like `take 10 [0..]`.

For the evaluation stepper, non-strictness can be achieved by following the rules of the outermost-first evaluation. Expressions should be kept as thunks instead of values as long as possible.

4.2.5 Normal Form

Haskell differs between the **Weak Head Normal Form** (WHNF), and the **Normal Form** (NF)

In the lamda calculus as well as in the Haskell language, there is the term **redex**. Redex means 'reducible expression' and is used to denote an expression that is not yet fully evaluated.

- To get the **Weak Head Normal Form**, the evaluation is no longer continued when the outermost data constructor or lamda abstraction of an expression is reached [6] [7].
- In the plain **Normal Form**, an expression is completely evaluated and there are no redexes left.

Here are some expressions that are not in normal form and not in weak head normal form:

```haskell
"Hello" ++ " World" -- application of (++) operator
(\x -> [x, x+1, x+2]) 1 -- application of lamda
1 == 1 -- application of (==) operator
```

Listing 4: Some unevaluated expressions (not in normal form)

Here are the evaluated expressions from before in normal form:

```haskell
"Hello World" --string
[1, 2, 3] --list
True --boolean
```

Listing 5: Some expressions in normal form

Every expression in normal form is also in weak head normal form, while the opposite must not be true.

Recognizing the weak head normal form is easy. Whenever the outermost part of an expression is a data constructor, a lamda or partial function application, the expression is in weak head normal form. Here are some examples:

```haskell
'H' : ("ello World") --data constructor
\x -> [x, x+1, x+2] --lamda
True --data constructor
```

Listing 6: Some expressions in weak head normal form but not in normal form

The weak head normal form is used for lazy evaluation because it offers enough information without having to evaluate everything. For example, a pattern matching on a list can be done without having to evaluate every single element inside the list.

The goal of the evaluation stepper is to bring expressions into the weak head normal form. This is also what GHCi does when an expression is entered.

4.2.6 Pureness

Haskell is a pure language, which means that functions do not have a side effect by changing some internal state which could potentially change the result of a future computation. Still, Haskell can do Input/Output by wrapping up I/O actions with an I/O monad.

For the goal of stepping through an evaluation, pureness is a huge advantage because it allows expressions to be rewritten and evaluated without having to worry that the evaluation or rewriting itself could have any potentially unwanted side effects. It also means that at any point, an expression inside the code could be substituted with a reduced form of the expression without changing the behavior of the program.

4.3 Glasgow Haskell Compiler

4.3.1 About the Glasgow Haskell Compiler

The Glasgow Haskell Compiler (GHC)[6] is the best known compiler for Haskell. In the next subchapters, we analyze the process of compiling. We refer to the first three compilation stages as the 'frontend': parsing, renaming and type checking.

We also analyze the intermediate languages 'Haskell Core' and 'STG' (Spineless Tagless Graph reduction machine).

The main goal of this section is to assess the feasibility of using GHC's frontend to parse Haskell code and use the resulting AST for stepping through a program.

4.3.2 Compilation

Overview of the GHC pipeline Using a substitution stepper might give the impression that Haskell code is processed in a linear way. However, this is not the case. Haskell does not have an imperative program flow, but instead is non-strict as discussed earlier. The Haskell code is therefore put into a graph structure by the compiler to be processed further from there. Let's take a look at how the Haskell code is actually processed by the GHC pipeline:

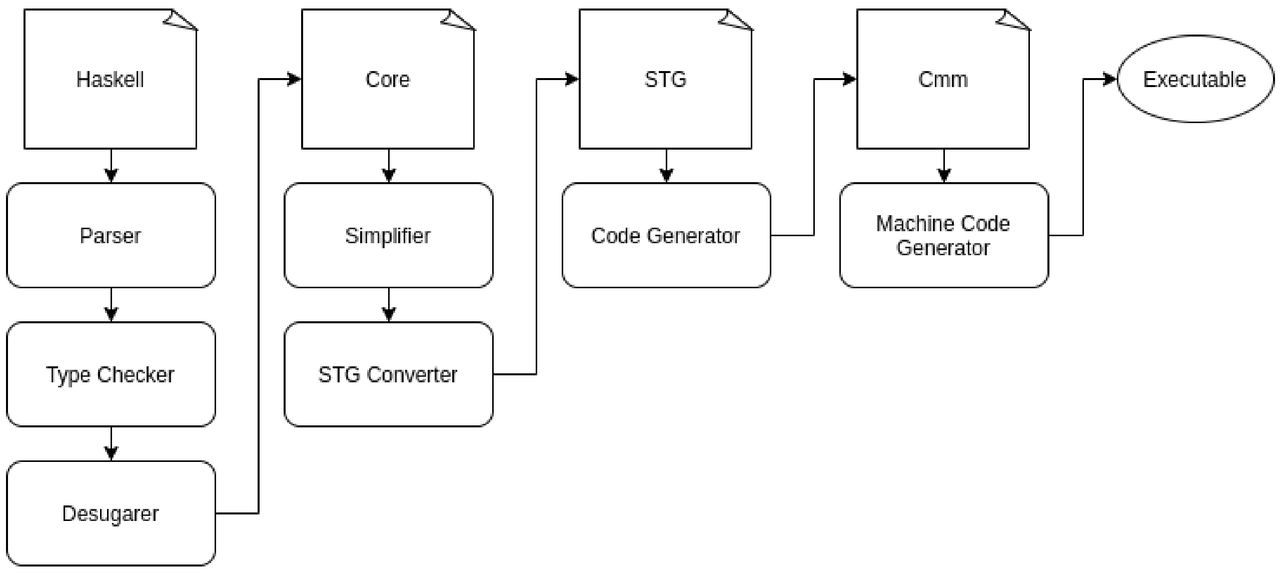


Figure 5: GHC stages

[6]https://www.haskell.org/ghc/

Parser Haskell source code is first fed to the parser. This results in an abstract syntax tree `HsSyn RdrName` with identifiers of type `String`. After the renaming step, the syntax tree ends up being of type `HsSyn Name`

Typechecker The syntax tree of type `HsSyn Name` is then passed to the type checker. The type checker validates the code and generates a syntax tree with complete type information of type `HsSyn Id`, where `HsSyn Id` is a `Name` with complete type information.

Haskell Core The Haskell program is now converted into the Haskell Core language. Haskell Core is also called desugared Haskell, because many language features of Haskell get stripped away in Haskell Core. However, Haskell Core is not a subset of Haskell but a language on its own. But the concepts that are used also exists in the original Haskell. Here is an example of a Haskell expression with the rewritten variant in Haskell Core [8]:

```haskell
f x y = not x && y
```

Listing 7: Definition of a function with two arguments in Haskell

```haskell
f =
  \(x :: Bool) ->
  \(y :: Bool) ->
  (&&) (not x) y
```

Listing 8: Definition of a function with two arguments in Haskell Core

As we see in the listing, Haskell Core still looks a lot like Haskell. It is more laborious to read as it uses nested lambdas to define the function.

This representation of Haskell serves a couple of purposes. The most obvious one is simplicity. Translating Haskell into a much smaller language early in the pipeline simplifies all subsequent steps. Transforming and optimizing a tiny language such as Core can be done more elegantly and concisely than doing the same for a large language.

Another, perhaps less obvious, advantage of Core is that it acts as a barrier for any language features that may be added in the future. If a Haskell language extension compiles to Core, it is guaranteed to also pass through all the other stages of the compiler as well. This results in those new language features also benefiting from the optimizations that happen further down the line.

Simplifier The simplifier does numerous transformations of type `Core -> Core`. It eliminates dead code, inlines functions and applies so-called ‘rewrite rules‘, which are used to optimize the abstract language features Haskell offers us into more efficient representations.

STG Converter In the next step, the code is transformed into an ‘STG‘ representation (spineless, tagless graph reduction machine).

STG An STG defines how expressions can be evaluated for a higher-order, functional and lazy language and is optimized for performance and therefore has no types anymore [9][10]. The STG language mainly consists of applications, evaluations, branching, definitions and lamdas. A lot of the Haskell syntax is gone: `if`-statements, to give one concrete example, are replaced by pattern matching on a boolean [11]. There are also no more `where`-clauses, guards or nested expressions anymore. Surprisingly, there are not many rules to evaluate STG-Code (Source: own illustration, based on [12]):

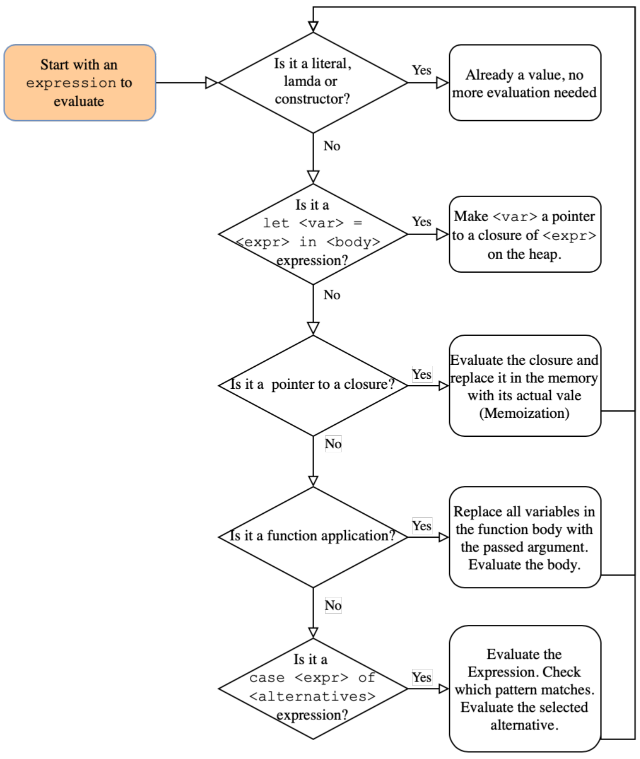

Start with an expression to evaluate
Is it a literal, lamda or constructor?
Yes
Already a value, no more evaluation needed
No
Is it a let <var> = <expr> in <body> expression?
Yes
Make <var> a pointer to a closure of <expr> on the heap.
No
Is it a pointer to a closure?
Yes
Evaluate the closure and replace it in the memory with its actual vale (Memoization)
No
Is it a function application?
Yes
Replace all variables in the function body with the passed argument. Evaluate the body.
No
Is it a case <expr> of <alternatives> expression?
Yes
Evaluate the Expression. Check which pattern matches. Evaluate the selected alternative.

Figure 6: Evaluation rules for STG (simplified)

Code Generator From this point, `.cmm` files are generated as the next intermediate step.

Cmm This file type basically contains C– code with some minor differences [13].

Machine Code Generator The final step turns the Cmm code into an executable file.

4.3.3 A Deeper Look into the Abstract Syntax Trees

Running Example In the next few sections, we work with the following code example in Haskell to show how this code gets transformed in the different stages of GHC:

```haskell
addOne :: Integer -> Integer
addOne x = x + 1
```

Listing 9: Function used to generate the AST

This code example defines a function `addOne` that takes an `Integer` as a parameter and returns the same value increased by one.

Complexity of the ASTs The Haskell language is specified by the Haskell2010 language report [14]. In theory, this compact and expressive language could be represented in an easy-to-understand abstract syntax tree (AST). However, looking at how most modern Haskell projects are written, one cannot help but notice the extensive use of multiple language extensions by which syntax and language features are expanded vastly. This approach to language evolution by selective language extensions is a rather unique trait for a programming language, and as such requires some additional considerations when designing a parser and accompanying AST. Extensibility and plugability are key features of such a design. As a consequence of this design, the AST that GHC generates ends up being large and unwieldy, as noted by several experts we consulted during the evaluation.

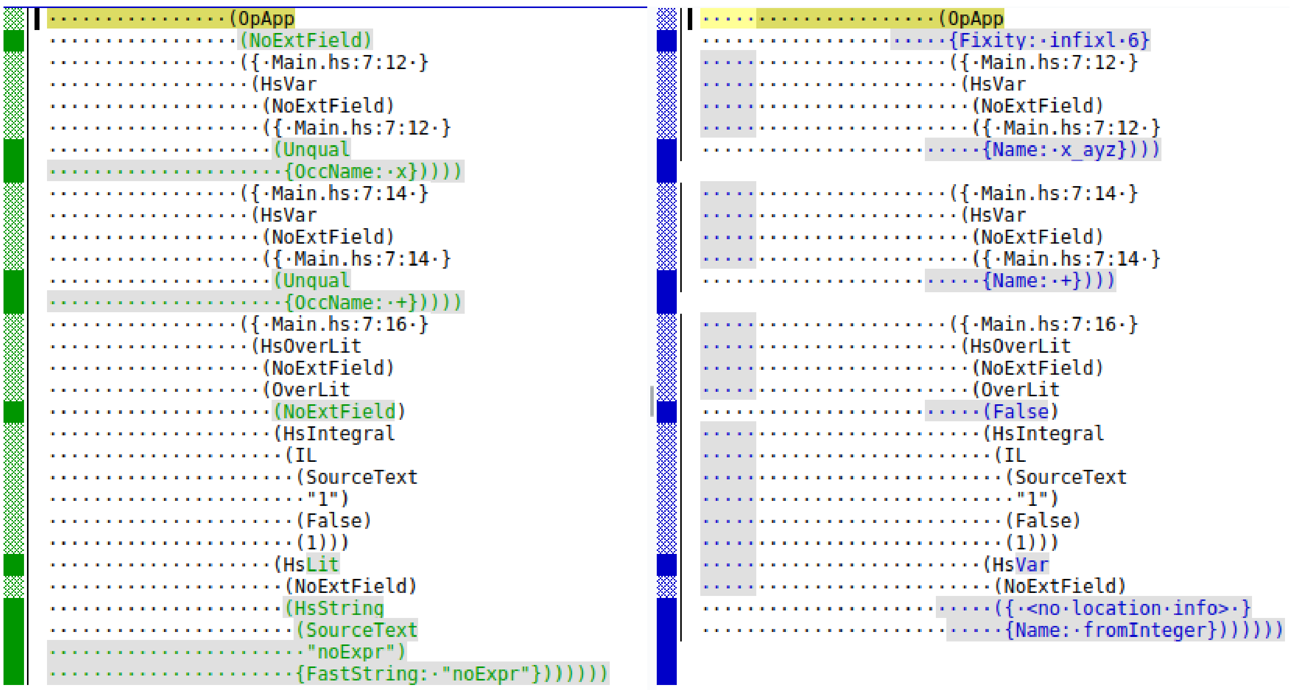
Parser AST Omitting the function declaration and some noise, the resulting AST after parsing takes the following form:

```
(OpApp -- applying an operator to a value
(NoExtField)
({ Main.hs:7:12 }
  (HsVar
  (NoExtField)
  ({ Main.hs:7:12 }
    (Unqual
    {OccName: x}))))
({ Main.hs:7:14 }
  (HsVar
  (NoExtField)
  ({ Main.hs:7:14 }
    (Unqual
    {OccName: +}))))
({ Main.hs:7:16 }
  (HsOverLit
  (NoExtField)
  (OverLit
    (NoExtField)
    (HsIntegral
    (IL
      (SourceText
      "1")
      (False)
      (1)))
    (HsLit
    (NoExtField)
    (HsString
      (SourceText
      "noExpr")
      {FastString: "noExpr"}))))))
```

Listing 10: AST after parsing

We begin at the `OpApp` node in the tree, which marks the beginning of the subtree for applying an operator to a value, in this case `+` is applied to `x` and `1`. A recurring pattern in this tree and all subsequent examples is the `NoExtField` leaf. It represents the absence of a TTG (Trees that grow) extension point as described in 'Trees that Grow' [15]. Although not relevant for our examination of this example, it hints at the extensibility of the tree.

Renamer AST Moving on in the compiler pipeline, this tree is decorated with additional information by the renamer. It can be useful to visualize the next stage as a diff to the previous (parser AST):



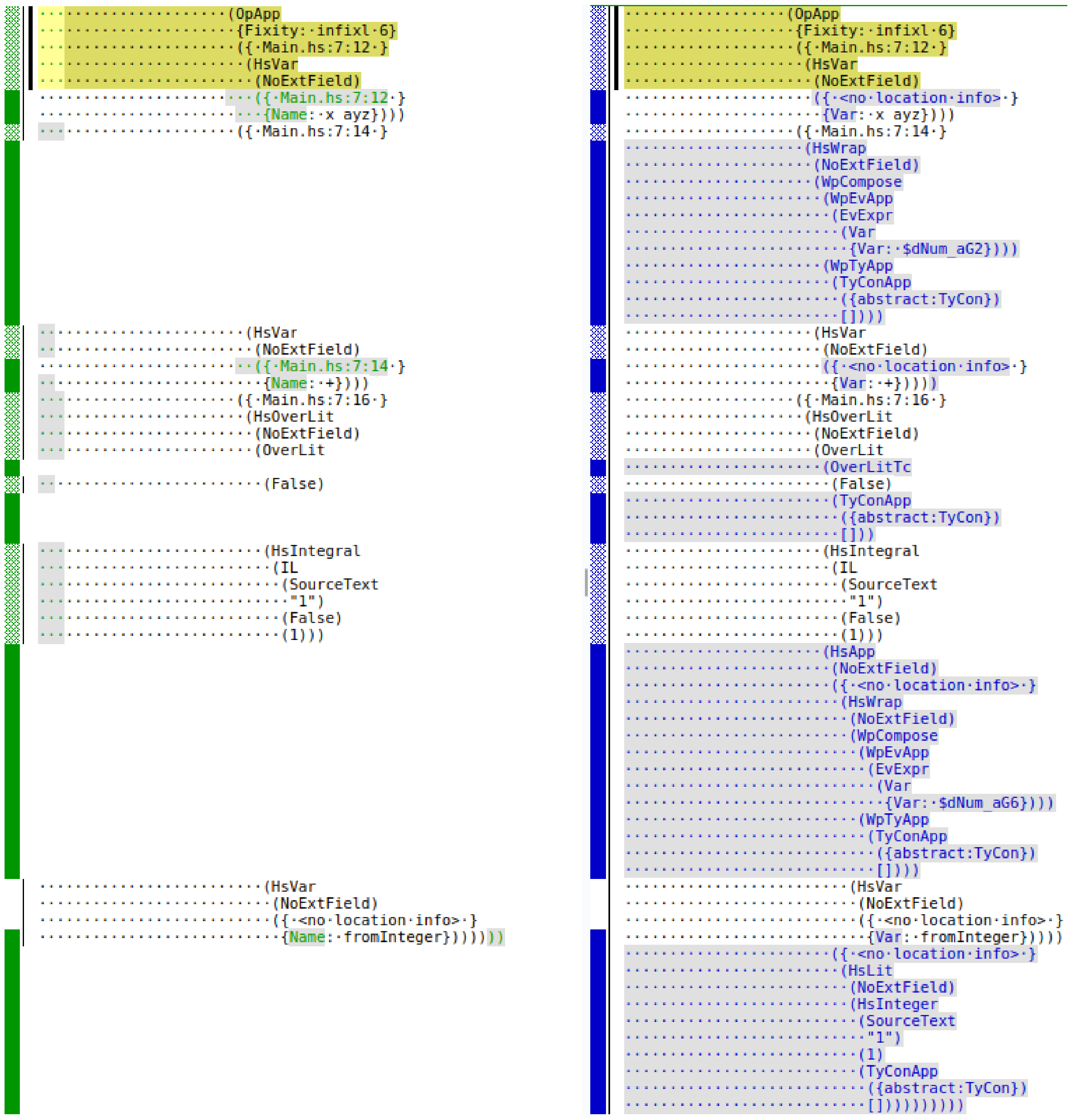
```
                (OpApp
                 (NoExtField)
                 ({ Main.hs:7:12 }
                  (HsVar
                   (NoExtField)
                   ({ Main.hs:7:12 }
                    (Unqual
                     {OccName: x}))))
                 ({ Main.hs:7:14 }
                  (HsVar
                   (NoExtField)
                   ({ Main.hs:7:14 }
                    (Unqual
                     {OccName: +}))))
                 ({ Main.hs:7:16 }
                  (HsOverLit
                   (NoExtField)
                   (OverLit
                    (NoExtField)
                    (HsIntegral
                     (IL
                      (SourceText
                       "1")
                      (False)
                      (1)))
                    (HsLit
                     (NoExtField)
                     (HsString
                      (SourceText
                       "noExpr")
                      {FastString: "noExpr"}))))))
```

```
                     (OpApp
                      {Fixity: infixl 6}
                      ({ Main.hs:7:12 }
                       (HsVar
                        (NoExtField)
                        ({ Main.hs:7:12 }
                         {Name: x_ayz})))

                      ({ Main.hs:7:14 }
                       (HsVar
                        (NoExtField)
                        ({ Main.hs:7:14 }
                         {Name: +})))

                      ({ Main.hs:7:16 }
                       (HsOverLit
                        (NoExtField)
                        (OverLit
                         (False)
                         (HsIntegral
                          (IL
                           (SourceText
                            "1")
                           (False)
                           (1)))
                         (HsVar
                          (NoExtField)
                          ({ <no location info> }
                           {Name: fromInteger}))))))
```

Figure 7: From the parser AST to the renamer AST

We can see how the renamer turns unqualified names from the parser into qualified and unique names. We also see that the mysterious `noExpr` from the parser stage is revealed to be a `fromInteger` call.

Typechecker AST Now we move on to the final tree representation before the transformation into the Core language. This is done by the typechecker.



```
(OpApp
 {Fixity: infixl 6}
 ({ Main.hs:7:12 }
  (HsVar
   (NoExtField)
   ({ Main.hs:7:12 }
    {Name: x ayz})))
 ({ Main.hs:7:14 }
  (HsVar
   (NoExtField)
   ({ Main.hs:7:14 }
    {Name: +})))
 ({ Main.hs:7:16 }
  (HsOverLit
   (NoExtField)
   (OverLit
    (False)
    (HsIntegral
     (IL
      (SourceText
       "1")
      (False)
      (1)))
    (HsVar
     (NoExtField)
     ({ <no location info> }
      {Name: fromInteger}))))))
```

```
(OpApp
 {Fixity: infixl 6}
 ({ Main.hs:7:12 }
  (HsVar
   (NoExtField)
   ({ <no location info> }
    {Var: x ayz})))
 ({ Main.hs:7:14 }
  (HsWrap
   (NoExtField)
   (WpCompose
    (WpEvApp
     (EvExpr
      (Var
       {Var: $dNum_aG2})))
    (WpTyApp
     (TyConApp
      ({abstract:TyCon})
      [])))
   (HsVar
    (NoExtField)
    ({ <no location info> }
     {Var: +}))))
 ({ Main.hs:7:16 }
  (HsOverLit
   (NoExtField)
   (OverLit
    (OverLitTc
     (False)
     (TyConApp
      ({abstract:TyCon})
      []))
    (HsIntegral
     (IL
      (SourceText
       "1")
      (False)
      (1)))
    (HsApp
     (NoExtField)
     ({ <no location info> }
      (HsWrap
       (NoExtField)
       (WpCompose
        (WpEvApp
         (EvExpr
          (Var
           {Var: $dNum_aG6})))
        (WpTyApp
         (TyConApp
          ({abstract:TyCon})
          [])))
       (HsVar
        (NoExtField)
        ({ <no location info> }
         {Var: fromInteger}))))
     ({ <no location info> }
      (HsLit
       (NoExtField)
       (HsInteger
        (SourceText
         "1")
        (1)
        (TyConApp
         ({abstract:TyCon})
         [])))))))))
```

Figure 8: From the renamer AST to the typechecker AST

We see some new segments which end with a `TyConApp` node. These segments result in a `TyCon` (Type Constructor) being applied by `TyConApp` to an abstract type constructor.

STG This is how the same code looks after the Core to STG conversion:

```
addOne =
    \r [x_sI7]
        let { sat_sId = CCCS S#! [1#]; } in  + $fNumInteger x_sI7 sat_sId;
```

Listing 11: STG

STG code is difficult to read. Before the conversion to STG a simplifier runs multiple times, making the code more performant.

Conclusion As seen in the previous example, even a simple function produces a complex AST. Although interpreting this AST on a basic level, as demonstrated, can be done, diving into the details is a time-intensive task. The amount of different nodes in the tree as well as all the operations performed are somewhat overwhelming. We conclude that it is not feasible to base our substitution stepper on any of the ASTs produced by the three stages in GHCs frontend.

We also acknowledge our lack of understanding of various implementation details used in the design of the parser, renamer and especially the typechecker. The difficulty of applying the insights of the design talked about in the ‘Trees that Grow‘ document [15] and the real-world implementation in GHC is also noteworthy.

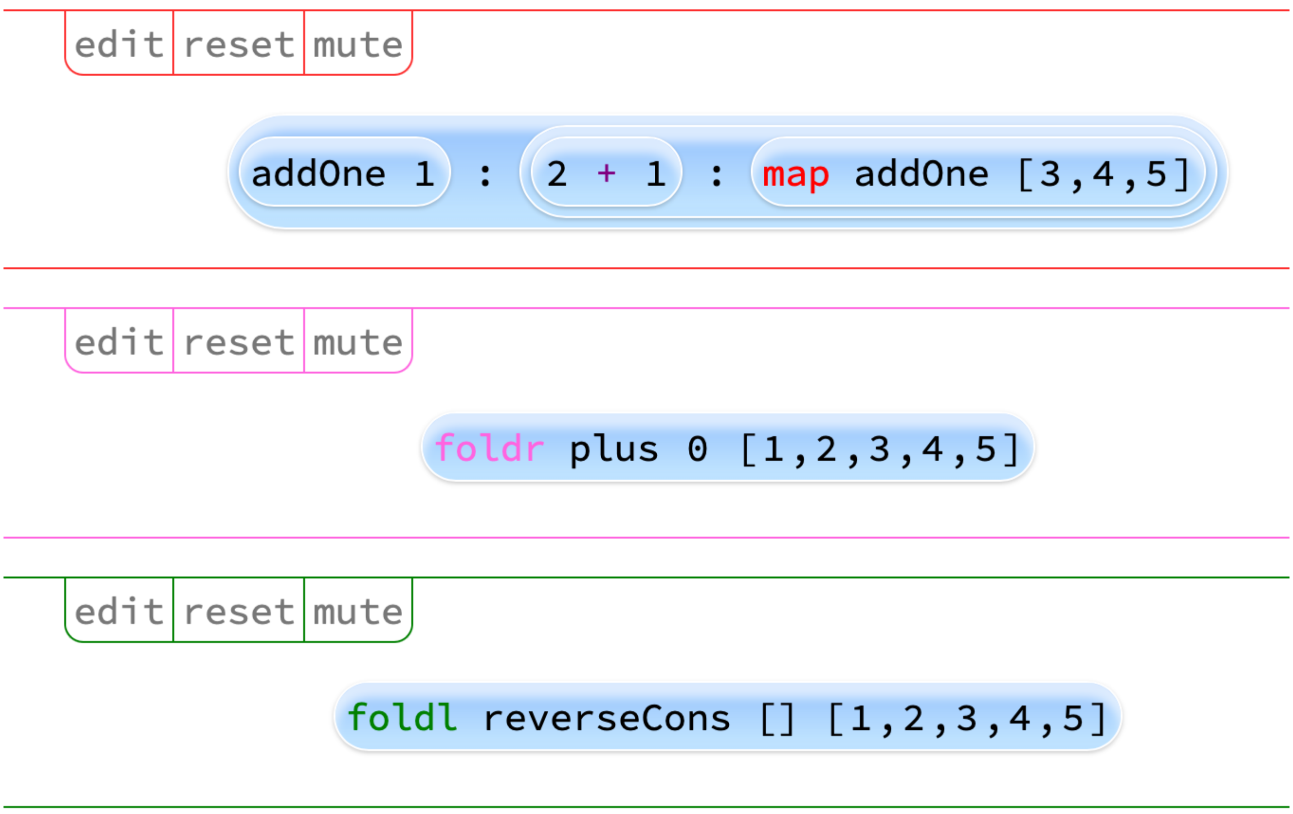
4.4 Existing Solutions

4.4.1 Overview

There are multiple existing solution that provide stepwise evaluation of Haskell expressions. They mainly differ in the language subset they support. We analyze the popular solutions and present our findings in the following chapters. None of the tools fulfilled the all the requirements and goals that we have for this project.

4.4.2 Lambda Bubble Pop / hs.js

Lambda Bubble Pop[7] is an intuitive web-based graphical tool for stepwise evaluation of very simple functional-style programs. It is based on the hs.js tool provided by the Lambda-Lessons project[8]. It implements a haskell-like / pseudo-haskell language that only supports integer- and list- types. It is written exclusively in javascript and runs locally in the browser. The tool does not use the GHC-toolchain, but implements its own parser and evaluator.



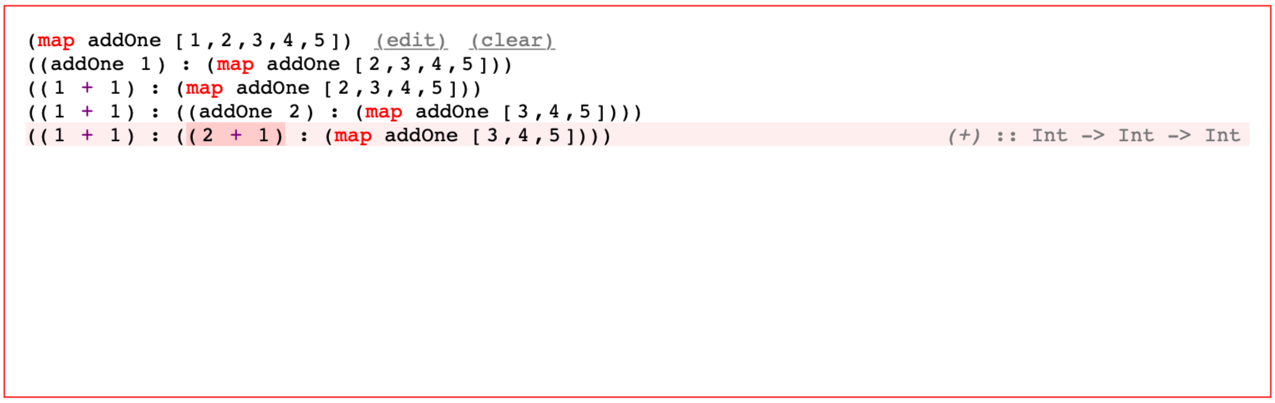
```
addOne 1 : 2 + 1 : map addOne [3,4,5]
```

```
foldr plus 0 [1,2,3,4,5]
```

```
foldl reverseCons [] [1,2,3,4,5]
```

Figure 9: Screenshot of Lamda Bubble Pop

[7] https://chrisuehlinger.com/LambdaBubblePop/
[8] https://stevekrouse.com/hs.js/



```
(map addOne [1,2,3,4,5])  (edit)  (clear)
((addOne 1) : (map addOne [2,3,4,5]))
((1 + 1) : (map addOne [2,3,4,5]))
((1 + 1) : ((addOne 2) : (map addOne [3,4,5])))
((1 + 1) : ((2 + 1) : (map addOne [3,4,5])))                (+) :: Int -> Int -> Int
```

Figure 10: Screenshot of Lamda Lessons

The limitations of those tools are quite obvious. Trying to evaluate the following expression results in a parser error:

```haskell
getLast :: String -> String
getLast [] = []
getLast [x] = [x]
getLast (_:xs) = getLast xs
```

Listing 12: Lamda Lessons breaking example

Because `String` is not a supported datatype, this function will not parse. It fails with the not so helpful (at least to novices) message: `Expected " " or "]" but "x" found.`

4.4.3 Duet

The approach that Duet[9] takes is very similar to Lambda Bubble Pop, but with a much larger subset of the Haskell language. It supports quite a few data types as well as type classes. The custom parser and evaluator are both written in Haskell itself, the same also goes for the rest of the code-base. There are no obvious shortcomings with this approach, as most programs will evaluate stepwise without any issue. The examples show some simple factorial functions, as well as more complicated programs, such as a complete monad instance for the `Maybe` datatype:

```
class Monad (m :: Type -> Type) where
bind :: m a -> (a -> m b) -> m b

class Applicative (f :: Type -> Type) where
  pure :: a -> f a

class Functor (f :: Type -> Type) where
  map :: (a -> b) -> f a -> f b

data Maybe a = Nothing | Just a

instance Functor Maybe where
  map =
    \f m ->
      case m of
        Nothing -> Nothing
        Just a -> Just (f a)

instance Monad Maybe where
  bind =
    \m f ->
      case m of
        Nothing -> Nothing
        Just v -> f v

instance Applicative Maybe where
  pure = \v -> Just v

  main = bind (pure 1) (\i -> Just (i * 2))
```

Listing 13: Code example in Duet

The above example can be evaluated in two ways. One yields a concise evaluation, the other a more detailed one.

[9]https://github.com/chrisdone/duet

```
bind (pure 1) (\i -> Just (i * 2))
Just (1 * 2)
Just 2
```

Listing 14: Concise stepping

```
bind (pure 1) (\i -> Just (i * 2))
(\m f ->
   case m of
     Nothing -> Nothing
     Just v -> f v)
  (pure 1)
  (\i -> Just (i * 2))
(\f ->
   case pure 1 of
     Nothing -> Nothing
     Just v -> f v)
  (\i -> Just (i * 2))
case pure 1 of
  Nothing -> Nothing
  Just v -> (\i -> Just (i * 2)) v
case (\v -> Just v) 1 of
  Nothing -> Nothing
  Just v -> (\i -> Just (i * 2)) v
case Just 1 of
  Nothing -> Nothing
  Just v -> (\i -> Just (i * 2)) v
(\i -> Just (i * 2)) 1
Just (1 * 2)
Just 2
```

Listing 15: Detailed stepping

There is also basic support for I/O. The functions `GetLine` and `PutStrLn` enable simple writing and reading from and to the terminal. Duet is intended to be used as a teaching tool for Haskell. We believe it meets this goal completely. It can be used in the browser with a simple user interface that supports direct text input or as a CLI utility that reads in `.hs` files.

Duet does not support `let` and `where` bindings. This limitation can be overcome by the use of inline lambda functions. For small examples, this shortcoming is a minor inconvenience. However, for larger examples it can become a bit of a hassle.

Named functions and the `import`-statement are also not supported. This results in the following examples are all not working in Duet:

```haskell
-- named function
f x = x ++ "!"
main = putStrLn (f "hi")

-- where binding
main =
putStrLn x
where x = "hi"

-- let binding
main =
let x = "hi"
in putStrLn x

-- using imports
import Data.List
main =
  let x = intercalate "," ["Hi", "Bye"]
  in putStrLn x
```

Listing 16: Not supported in Duet

Duet operates independently of the GHC ecosystem and is implemented using a custom tokenizer, parser, typechecker and stepper. Duet also does not try to implement a prelude, although there are some example-implementations of well-known prelude-functions, such as `foldl` or `map`. We think that Duet is the most advanced solution of all the analyzed ones. In the next chapter, we take a more profound look inside the code of Duet.

4.4.4 How Duet works

Documentation The stepper for Duet itself is completely written in Haskell. Unfortunately, the code is documented only sporadically and partly non-descriptive names are used. Therefore, it was difficult to analyze how the stepping process works. The process is shown here in a simplified way for illustration purposes.

Function runStepper In the file `Main.hs` there are the functions `runStepper` and `runStepperIO`, which we have looked deeper into. These functions accept an expression in the form of a string, as well as the context and binding groups. The maximum number of substitution-steps can also be defined.

Conversion of the String into an Expression In a first step, the expression-string is converted into an expression type. For this purpose, there is a helper function `lookupNameByString`. The binding groups and the string are passed as parameters to this function. The function looks up the matching bindings for the passed names.

Expression Type The `Expression`-type looks like this:

```haskell
data Expression (t :: * -> *) i l
  = VariableExpression l i
  | ConstructorExpression l i
  | ConstantExpression l Identifier
  | LiteralExpression l Literal
  | ApplicationExpression l (Expression t i l) (Expression t i l)
  | InfixExpression l (Expression t i l) (String, Expression t i l)
  ↪  (Expression t i l)
  | LetExpression l (BindGroup t i l) (Expression t i l)
  | LambdaExpression l (Alternative t i l)
  | IfExpression l (Expression t i l) (Expression t i l) (Expression t i
  ↪  l)
  | CaseExpression l (Expression t i l) [CaseAlt t i l]
  | ParensExpression l (Expression t i l)
  deriving (Show, Generic, Data, Typeable, Functor, Traversable,
  ↪  Foldable, Eq)
```

Listing 17: Expressions in Duet

(Note: `Let`-Expressions are listed, although they are not supported)

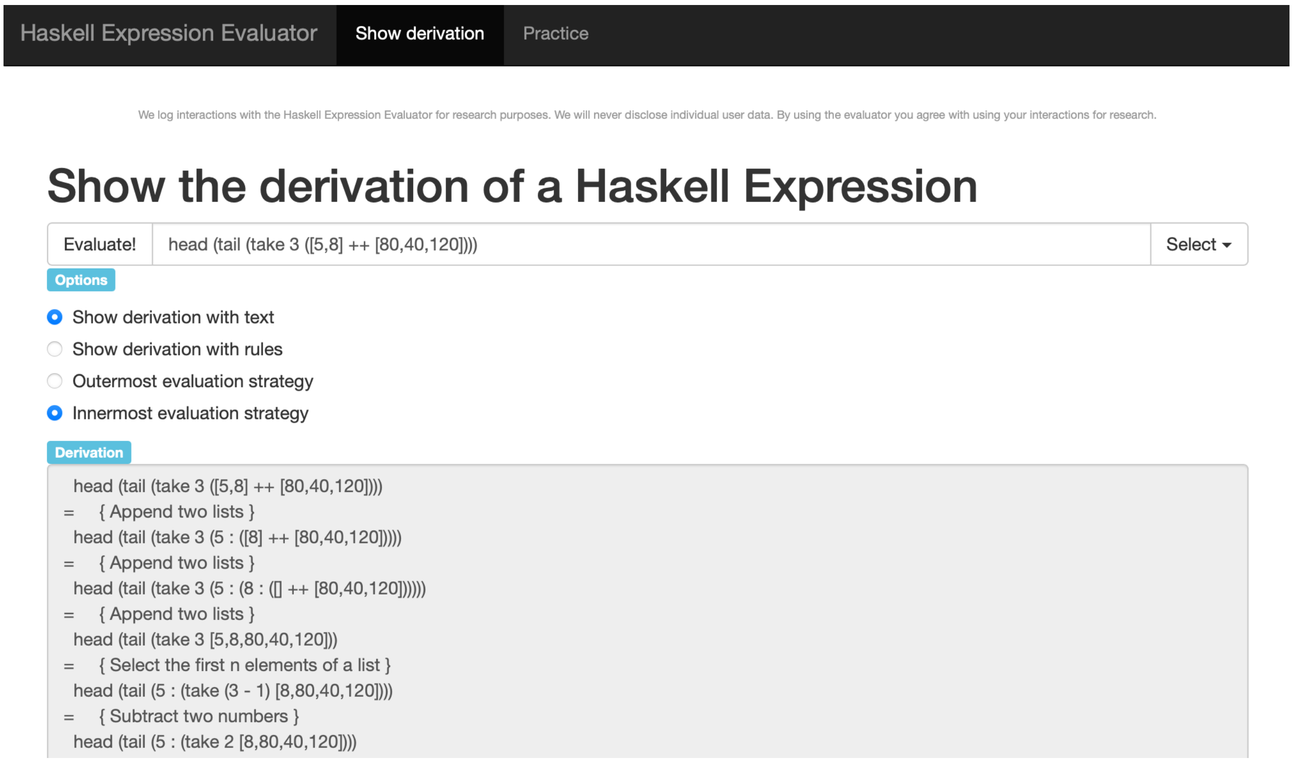
The Stepper Loop To fully reduce an expression, the function `expandSeq1` is called repeatedly in a loop. The expression obtained from this function is then printed to the console. This loop runs until the expression has been reduced completely or until the desired number of `maxSteps` has been reached.

`expandSeq1` is the actual heart of the stepper. The function is called with an expression as well as a Context and the BindingGroups as parameters. Since the type of the expression is known, it is possible to react differently to the different expression types using a `case-of`-statement.

The final goal of the substitution is to obtain the normal form. As Duet is non-strict but not-lazy, all sub-expressions are fully reduced.

4.4.5 Haskell Expression Evaluator

Haskell Expression Evaluator[10] is a prototype to step through Haskell-Expressions. Compared to Duet, the prelude is already implemented, so expressions using functions from the prelude can be directly evaluated. However, there is no possibility to enter the surrounding context of an expression. Custom functions can be added if the project is run locally. (The functions are defined in a file on the web server that cannot be changed from the user's perspective). The syntax is limited and only supports integers, lists, recursion, and functions with pattern matching. To reduce the expressions, there is a set of rewrite rules implemented. For each substitution-step, the name of the rule that was chosen is shown. The user can also choose between the 'outermost evaluation strategy' and the 'innermost evaluation strategy'. Despite its limited possible applications, the tool is interesting because it is very well documented in the corresponding research paper [16].



We log interactions with the Haskell Expression Evaluator for research purposes. We will never disclose individual user data. By using the evaluator you agree with using your interactions for research.

Show the derivation of a Haskell Expression

head (tail (take 3 ([5,8] ++ [80,40,120])))

[✓ Show derivation with text]
Show derivation with rules
Outermost evaluation strategy
[✓ Innermost evaluation strategy]

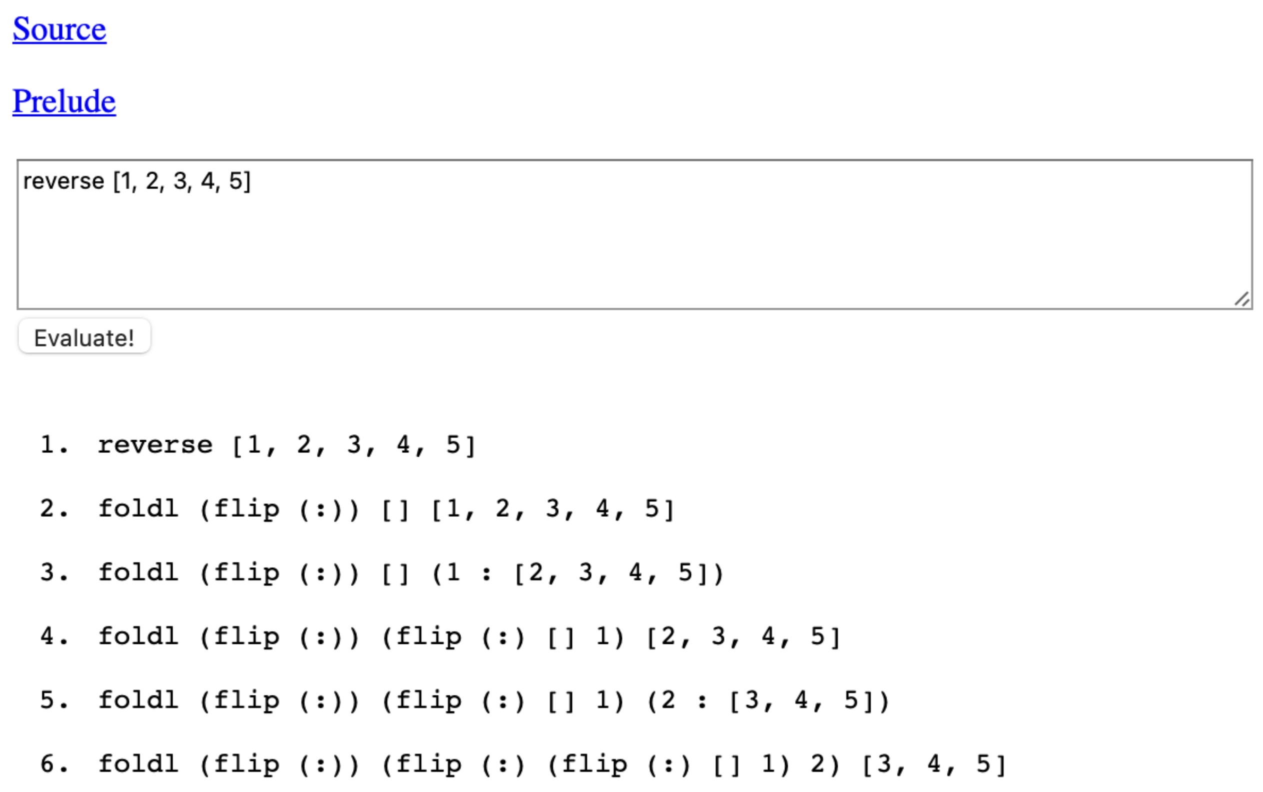
```
 head (tail (take 3 ([5,8] ++ [80,40,120])))
=    { Append two lists }
 head (tail (take 3 (5 : ([8] ++ [80,40,120]))))
=    { Append two lists }
 head (tail (take 3 (5 : (8 : ([] ++ [80,40,120])))))
=    { Append two lists }
 head (tail (take 3 [5,8,80,40,120]))
=    { Select the first n elements of a list }
 head (tail (5 : (take (3 - 1) [8,80,40,120])))
=    { Subtract two numbers }
 head (tail (5 : (take 2 [8,80,40,120])))
```

Figure 11: Screenshot of Haskell Expression Evaluator

[10]https://ideas.science.uu.nl/HEE/

4.4.6 Stepeval

The user interface of Stepeval[11] is quite simple. Users can enter an expression with a function from the prelude which is then evaluated step by step. The possibilities are restricted. As a user there is no way, to enter own function-definitions. The implementation does not use a typed AST, so the type information gets lost during evaluation [17]. Unlike the Haskell Expression Evaluator, the evaluation strategy cannot be chosen. The input is parsed using the haskell-src-exts package [12].



```
reverse [1, 2, 3, 4, 5]
```

```
1. reverse [1, 2, 3, 4, 5]
2. foldl (flip (:)) [] [1, 2, 3, 4, 5]
3. foldl (flip (:)) [] (1 : [2, 3, 4, 5])
4. foldl (flip (:)) (flip (:) [] 1) [2, 3, 4, 5]
5. foldl (flip (:)) (flip (:) [] 1) (2 : [3, 4, 5])
6. foldl (flip (:)) (flip (:) (flip (:) [] 1) 2) [3, 4, 5]
```

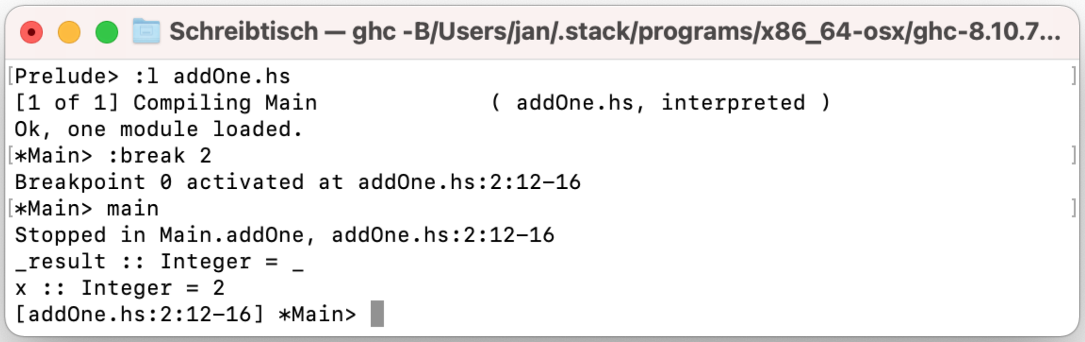
Figure 12: Screenshot of Stepeval

[11]https://github.com/bmillwood/stepeval, interactive demo available here: http://bm380.user.srcf.net/cgi-bin/Prelude.txt
[12]https://hackage.haskell.org/package/haskell-src-exts

4.4.7 GHCi Debugger

The GHCi debugger offers some interesting possibilities to analyze source code. Breakpoints can be set on functions as well as on expressions. For lazy evaluated expressions, evaluation can be forced using `:force`. However, there are no sub-steps shown [18].

Here, we use the `addOne` function that we defined before. We set a breakpoint on the second line and then call `addOne 2`. As we see, the execution is paused:


Schreibtisch — ghc -B/Users/jan/.stack/programs/x86_64-osx/ghc-8.10.7...

```
Prelude> :l addOne.hs
[1 of 1] Compiling Main             ( addOne.hs, interpreted )
Ok, one module loaded.
*Main> :break 2
Breakpoint 0 activated at addOne.hs:2:12-16
*Main> main
Stopped in Main.addOne, addOne.hs:2:12-16
_result :: Integer = _
x :: Integer = 2
[addOne.hs:2:12-16] *Main>
```

Figure 13: GHCi debugger on running example

4.4.8 GHC-vis

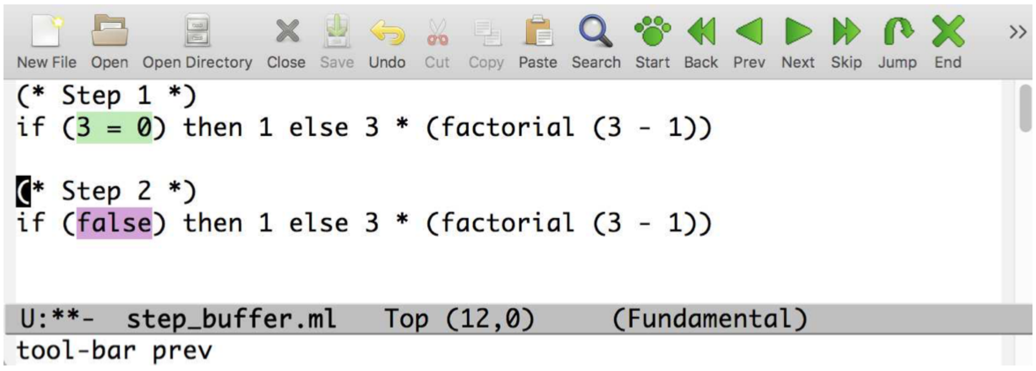
With GHC-vis[13], data structures on the heap can be displayed graphically. This primarily helps to understand the concepts of lazy evaluation and sharing. The tool is mainly used inside the interactive GHCi environment. Data structures can be defined with `let`, and then be displayed visually with the command `:view`. The references between the individual data structures are also shown. The expressions are not evaluated, but this can be enforced. GHC-vis can also be used as a library.

[13]http://felsin9.de/nnis/ghc-vis/

4.4.9 OCaml Stepper

The OCaml stepper was created as a lecturing-support for a programming class in the Ochanomizu University in Tokyo, Japan. The idea was to provide students with a basic use tool to ‘debug‘ their code. OCaml is in essential parts different from the Haskell language, although the syntax looks kind of familiar. A big difference is that OCaml, in contrast to Haskell, is that impure functions can have side effects. Similar to Haskell, the language is often used in an academic context.

The OCaml stepper does not support the whole language but still a big part of it though (user-defined data types, user-defined modules, exception handling...). The stepper was created by changing the interpreter. The built-in parser is used to load the source into an AST. For each reduction step, the entire program gets reconstructed [19].



```
(* Step 1 *)
if (3 = 0) then 1 else 3 * (factorial (3 - 1))

(* Step 2 *)
if (false) then 1 else 3 * (factorial (3 - 1))


U:**-  step_buffer.ml   Top (12,0)     (Fundamental)
tool-bar prev
```

Figure 14: Screenshot of the OCaml Stepper

4.5 Risk Analysis

4.5.1 Introduction

Before we started to think about the general architecture of the application or concrete implementation details, we wanted to make sure that the project is actually feasible to do in the way we wanted it to work. We also asked some Haskell experts about their thoughts and inputs. In the following chapters, we list some of the potential problems.

4.5.2 Missing API Interface

In the run-up to the project, we already expected that there will not yet be a ready-to-use solution for displaying the individual evaluation steps of the GHC. In our search for such a functionality, we did not find anything suitable. There is no API interface for the GHC to resolve expressions step by step. Also, an expert of the GHC-API, Vladislav Zavialov (GHC contributor since 2018), confirmed that he is not aware of such an interface:

> “As far as I'm aware, GHC doesn't include a step-by-step evaluator for Core expressions at the moment“
>
> (Vladislav Zavialov)

This is presumably the reason all analyzed projects that resolve Haskell expressions step by step do not cooperate with the GHC or other Haskell compilers, but instead implement their own parser for a limited Haskell-like language.

4.5.3 Conversion to Haskell Core and STG before evaluation

In a first compilation-step, Haskell code is converted into Haskell Core [20] as seen in Chapter 4.3.2. Doing a step by step reduction would be much easier for Haskell Core, however the resulting sub-expressions would be more complicated to read. However, some of the experts we asked encouraged us to work with Haskell Core:

Twan van Laarhoven, (Creator of the simple-reflect package for Haskell[14]), expressed concerns if it is applicable to work with the full Haskell language:

> “I think getting a step by step evaluation of the full Haskell language, in the way that a human would understand it out of Ghc is not easily possible. The way Ghc works [...] is nothing like the term rewriting that you normally think of.“
>
> (Twan van Laarhoven)

David Luposchainsky (Creator of stgi[15]) recommended us to work with Core:

[14]`https://hackage.haskell.org/package/simple-reflect`
[15]`https://github.com/quchen/stgi`

> “[Haskell Core is] probably what you want to implement, and then convince people they want to watch“
>
> (David Luposchainsky)

Unfortunately, there is no way to convert Haskell Core code back into Haskell form because such a conversion is ambiguous and certain syntactic information is lost in the conversion process. At least, the conversion to Core happens before running the simplifier, which would drastically change the whole program structure. Haskell Core does not support the same set of syntax features. One of the main differences between Haskell and Haskell Core is that `case` is ‘strict‘. The evaluation order is not specified in the Haskell report, but the operational semantic is surprisingly compact.

In contrast to Haskell Core, STG code is one level deeper with a deterministic evaluation order, an explicit stack and no types, which makes it easy to be resolved step by step. However, STG is complicated to read.

There is an existing tool to step through an STG program called stgi[16]. This tool is not based on GHC but on the original STG definition. (The STG language has changed overtime and is not compatible with that tool anymore, but STGi is still used for teaching purposes.)

4.5.4 Let Statements

Implementing the support for named bindings such as in `let`, `where` or top-level functions like `f x = x + 1` poses a challenge in the case a custom Haskell-like language would be implemented. Assuming we choose to go down that path, this is mainly a design challenge that needs to be overcome. This has also been stated by Chris Done in his Duet project.

[16] `https://github.com/quchen/stgi`

4.5.5 Conclusion and Further Procedure

To reach our main goal of stepping through Haskell source code in a user-friendly way, all approaches analyzed in this section have pros and cons:

	Advantages	**Disadvantages**
Stepping a Haskell-like language	- What the user expects - Most useful for beginners	- Requires a custom implementation of a Haskell like language (similar to Duet) - Deemed not possible using GHC provided APIs due to the execution model of Haskell in GHC
Stepping Core or STG	- Can be implemented in a custom way (STGi) or using GHC APIs - Follows actual execution steps of a Haskell program	- Not intended to be human readable - Code cannot be converted back to original Haskell
Tracing	- Can be implemented using GHC APIs - Follows actual execution steps of a Haskell program	- Rather experimental approach - Deducing substitution steps from a trace might not always be possible

Table 1: Pros and cons for different approaches

Together with our advisor, we have agreed that we - if possible - would try to build a substitution stepper that works with the GHC-API in order to support the full Haskell language. However, it turned out that this approach is not feasible. For the proof of concept, we only work with the GHC-frontend to get the typechecker AST and then continue to do the reduction based on the AST we get from GHC.

5 Proof of Concept

5.1 Idea

As we have seen in the previous chapters, the GHC generates an AST that is almost impossible to keep track of in its complexity. A large part of the information contained in the AST is not relevant for us (currently). We have therefore decided to work with a simplified AST. The reduction and output of the individual reduction-steps is based on this simplified AST. The typechecker AST we get from the GHC can be converted into this simplified AST.

This approach brings some advantages:

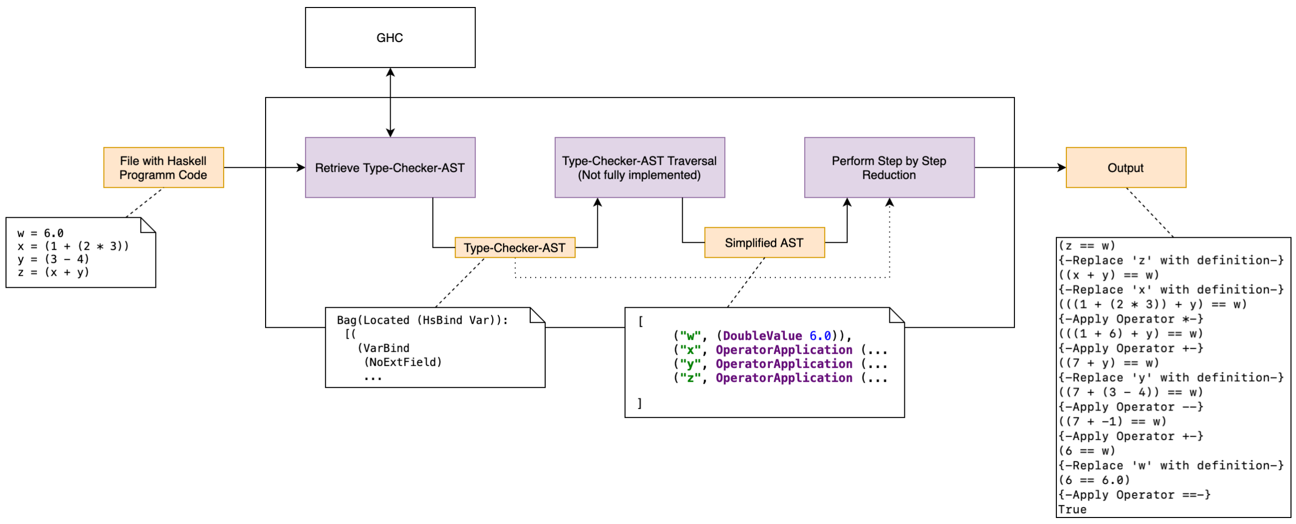
- The simplified AST transparently shows exactly which syntax constructs are currently supported.
- The code becomes easier to read. We only have to deal with the complex typechecker AST in one place (this is where the GHC AST is converted to our simplified AST). All other functions work with the simplified AST, which will significantly increase the readability of the code.
- Extracting the relevant information from the GHC AST is done in a single location and is not scattered throughout the code.
- There is an additional layer between the GHC AST and the stepper functions. Theoretically, it would even be possible to use a different AST than the GHC AST without having to rewrite the entire program. This approach gives us some flexibility if something changes in the GHC AST or if we decide to use a different version of the GHC AST (for example, the parser AST instead of the typechecker AST).

This approach also brings some disadvantages:

- We have to convert the data from one structure into another, this adds some overhead. (This was not implemented for the prototype.)
- The more syntax features the simplified AST supports, the more it will look like the GHC AST and therefore add some level of redundancy.
- The more syntactic constructs we support, the more time we have to spend thinking about the design of the simplified AST. Mistakes can be costly to correct later.
- The collaboration in the team must be well planned because we work with a data structure that changes again and again during the project.

5.2 Architecture

Here is the architecture of the proof of concept:


GHC
File with Haskell Programm Code
Retrieve Type-Checker-AST
Type-Checker-AST Traversal (Not fully implemented)
Perform Step by Step Reduction
Output
Type-Checker-AST
Simplified AST

```
w = 6.0
x = (1 + (2 * 3))
y = (3 - 4)
z = (x + y)
```

```
Bag(Located (HsBind Var)):
 [(
   (VarBind
    (NoExtField)
    ...
```

```
[
     ("w", (DoubleValue 6.0)),
     ("x", OperatorApplication (...
     ("y", OperatorApplication (...
     ("z", OperatorApplication (...

]
```

```
(z == w)
{-Replace 'z' with definition-}
((x + y) == w)
{-Replace 'x' with definition-}
(((1 + (2 * 3)) + y) == w)
{-Apply Operator *-}
(((1 + 6) + y) == w)
{-Apply Operator +-}
((7 + y) == w)
{-Replace 'y' with definition-}
((7 + (3 - 4)) == w)
{-Apply Operator --}
((7 + -1) == w)
{-Apply Operator +-}
(6 == w)
{-Replace 'w' with definition-}
(6 == 6.0)
{-Apply Operator ==-}
True
```

Figure 15: Architecture of the proof of concept

5.3 Example Code

In the next few sections, we use the following Haskell code example (running example) as the input for the proof of concept:

```
> x = 1 + 2 * 3
> y = 3 - 4
> z = x + y
> z
6
```

Listing 18: A basic Haskell example code

5.4 Implementation and Design Thoughts

5.4.1 Simplified Abstract Syntax Tree

The goal of the first simplified AST design was to support only a few syntax constructs. These are:

- Integer Values
- Operator Application
- Named Bindings

(Later in this chapter, we extend our simplified AST with more base types)

With those features, basic Haskell programs can be represented.

This is how the example code looks like in the simplified AST structure:

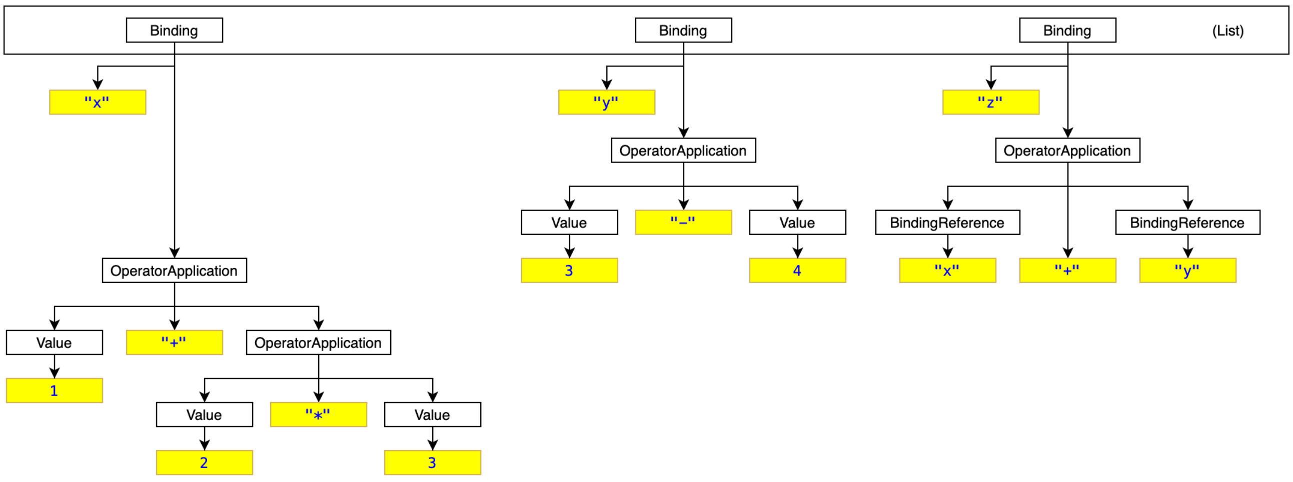


Figure 16: Simplified AST

In comparison – the same program is represented much more verbosely with the typechecker AST from the GHC. In this figure only the relevant nodes and their parents are shown. In addition, many nodes have been omitted for simplicity (like `Wraps` and `Bags`).

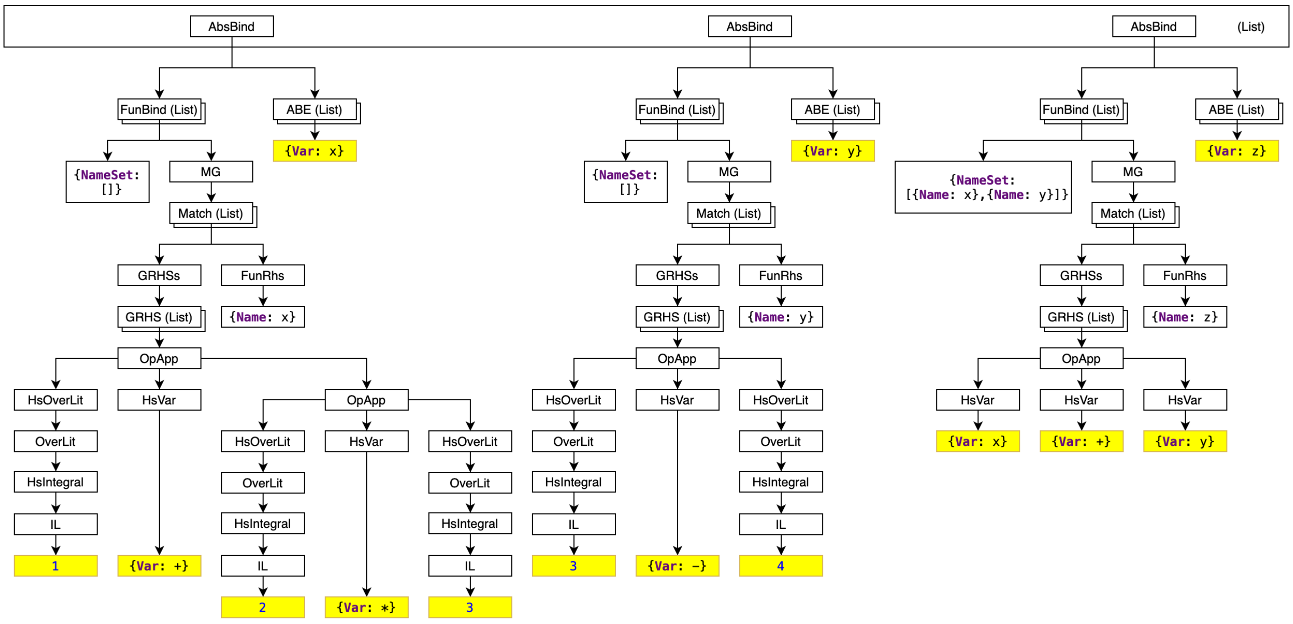


Figure 17: Type checker AST

For the definition of the simplified AST, we use a recursive data type `Expression`, which is structured as follows:

```haskell
data Expression
    = BindingReference String {-for example "x"-}
    | Value Int {-for example 1-}
    | OperatorApplication {leftExpression :: Expression, operator ::
    ↪  Operator, rightExpression :: Expression}} {-for example "1 + 1"-}
```

Listing 19: Expression data type

Here, `Operator` is just an alias for `String`.

5.4.2 Step by Step Reduction

For demonstration purposes, we built a simple stepper function to resolve expressions step by step using the simplified AST. Since only a tiny part of the Haskell syntax is implemented, this stepper function is simple. The stepping function takes two parameters:

- a list of bindings `[Binding]`. A `binding` is a Tuple that consists of a `String` (for example `"x"`) and an expression, for example `1+2*3`.
- an `expression` to reduce, for example, `x+1`

The stepper function creates a new expression that is reduced at one point. Thus, the function can be called several times until the expression is in the weak head normal form. The following reduction rules have been implemented:

- `BindingReferences` like `x` are replaced by their definition.
- If an `OperatorApplication` has the left and the right sub-expression in the form of a `Value`, the operation is calculated and the `OperatorApplication` gets replaced by a `Value`.
- Otherwise, the left and right sub-expressions of an `OperatorApplication` get recursively reduced.

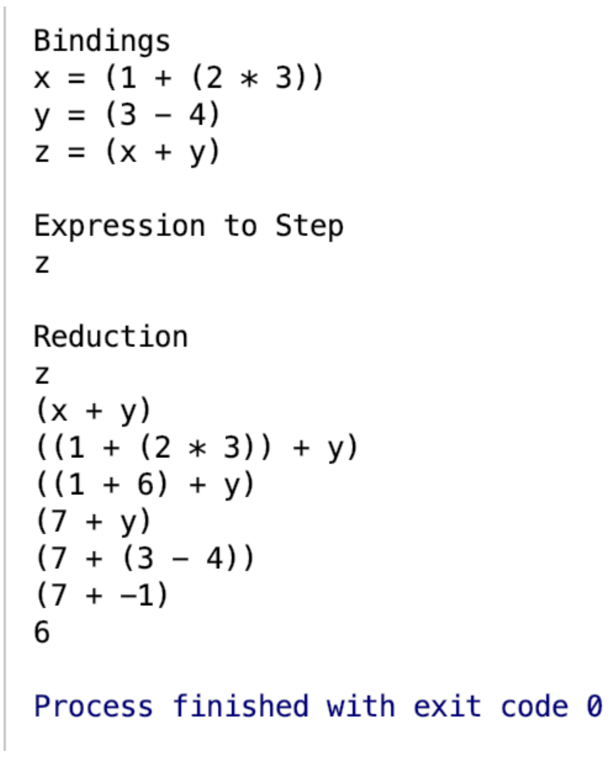
This approach works for the algebraic operators `+`, `-` and `*` that we have implemented for now. However, when we want to implement operators such as `||` or `&&` it will not be the best approach to reduce the operands on both sides of the operator as this would lead to non-strict behavior.

This is how the rules are implemented in code:

```haskell
applyStep :: [Binding] -> Expression -> Expression

{-cannot reduce more-}
applyStep _ (Value value) = Value value

{-replace binding reference with actual expression (Delta Reduction)-}
applyStep bindings (BindingReference reference) = findBinding reference bindings

 {-Apply Operator-}
 applyStep bindings (OperatorApplication (Value leftValue) operator (Value rightValue)) = (Value (applyOperator operator leftValue rightValue))

 {-Reduce Right-Side-Expression -}
 applyStep bindings (OperatorApplication (Value leftValue) operator rightExpression) = OperatorApplication (Value leftValue) operator (applyStep bindings rightExpression)

{-Reduce Left-Side-Expression -}
applyStep bindings (OperatorApplication leftExpression operator rightExpression) = OperatorApplication (applyStep bindings leftExpression) operator rightExpression
```

Listing 20: Function applyStep (slightly simplified)

Since the instance declaration `show` is implemented for the type `Expression`, the individual reduction steps can easily be printed to the console. The reduction output of the code example shown above looks like this:



```
Bindings
x = (1 + (2 * 3))
y = (3 - 4)
z = (x + y)

Expression to Step
z

Reduction
z
(x + y)
((1 + (2 * 3)) + y)
((1 + 6) + y)
(7 + y)
(7 + (3 - 4))
(7 + -1)
6

Process finished with exit code 0
```

Figure 18: Console output from the proof of concept (v1)

Here is a visualization of the `Expression`-type during reduction:

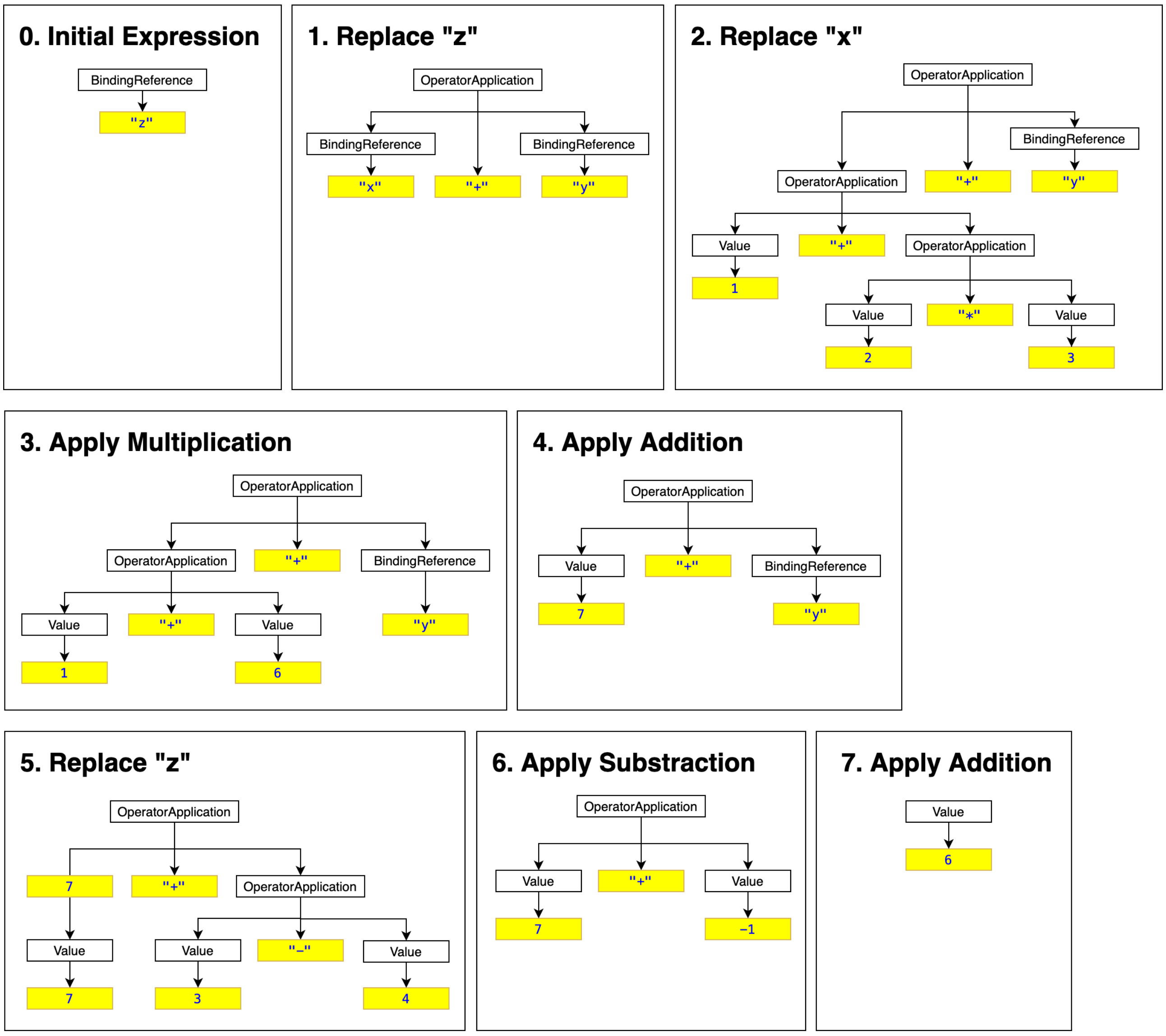


Figure 19: Reduction steps performed on the simplified AST

In the first version of the proof of concept, the input-bindings and the input-expression are created directly in the simplified AST structure.

5.4.3 Syntax Tree Conversion

We did not finish the tree-traversal function that would convert the typechecker AST into our simplified AST. For now, we hardcoded our expressions to reduce for the demo purpose.

5.4.4 Handling Invalid Expressions

Our Solution With our expression data type, it is possible to represent invalid expressions. For example, if we use unknown operators:

```
OperatorApplication (Value 1) "?" (Value 2)
```

Listing 21: Invalid expression in the simplified AST

Some errors would be recognized by the GHC before converting the GHC typechecker AST into our simplified AST. Still, we wanted to be able to represent invalid expressions. We added the `InvalidExpression` constructor to the type `Expression`. Every operation on an `InvalidExpression` will result in an `InvalidExpression` again.

The constructor `InvalidExpression` takes a `String` where a human-readable error message can be added. This message is printed if an `InvalidExpression` is printed to the console. We can also use `InvalidExpression` if a feature is not yet implemented.

Discarded Alternative Approach Instead of the `InvalidExpresion`-constructor, we could just have used `Maybe Expression` in all the helper functions, so a function could result in `Nothing` if a calculation or reduction on an expression leads to an invalid result.

We have first implemented this approach but soon realized that this really bloats up the whole code everywhere we have to wrap and unwrap the `Maybe Expression`. Additionally, we think that having an `InvalidExpression` is a more consistent approach and easier to understand for someone who reads the code.

5.4.5 Adding Support for more Base Types to our Simplified AST

Our Solution The first version of the proof of concept only supported `Integer` values. Next, we wanted to support more base types of Haskell. We choose to represent the different types in the following way:

```haskell
data Expression
    = StringValue String
    | IntegerValue Integer
    | DoubleValue Double
    | BoolValue Bool
    | CharValue Char
    | ListValue [Expression]
    | TupleValue (Expression, Expression)
    | OperatorApplication { leftExpression :: Expression
                          , operator :: Operator
                          , rightExpression :: Expression
                          }
    | BindingReference String
    | InvalidExpression String
```

Listing 22: Support for more base types in the simplified AST

Disadvantage of this approach This approach leads to some kind of redundancy. This can be seen in the following example:

```haskell
instance Show Expression where
  show (StringValue x) = x
  show (IntegerValue x) = show x
  show (DoubleValue x) = show x
  show (BoolValue x) = show x
  show (CharValue x) = show x
  show (ListValue x) = show x
  show (TupleValue x) = show x
  show (BindingReference x) = x
  show (InvalidExpression reason) = "(Invalid Expression: " ++ reason ++
   ↪ ")"
  show (OperatorApplication (leftExpression) operator (rightExpression)) =
   ↪ "(" ++ show leftExpression ++ " " ++ operator ++ " " ++ show
   ↪ rightExpression ++ ")"
```

Listing 23: Redundant implementation of show

As we see, the implementation for most value types look exactly the same.

Discarded Alternate Approach We first thought about making a polymorphic expression type like the following, where `a` could be an arbitrary type:

```haskell
data Expression a
        = Value a
    | ...
```

Listing 24: Polymorphic expression type

This would have allowed us to easily represent any type possible. However, this approach is not possible: Expressions can be nested and not every sub-expression will have the same type. For example, we can add an `Integer`-Value with a `Double`-Value:

```haskell
OperatorApplication (IntegerValue 1) "+" (DoubleValue 2.0)
```

Listing 25: Addition of IntegerValue and DoubleValue

Furthermore, in some cases, we will not be able to know the type before evaluating the entire expression.

How this is solved in Duet Duet has a similar `Expression` structure, as seen in Chapter 4.4.4. But the difference is that it uses multiple polymorphic types. Another difference is that there is only one constructor for literals. Literals are represented within a separate `Literal` type:

```haskell
data Literal
  = IntegerLiteral Integer
  | CharacterLiteral Char
  | RationalLiteral Rational
  | StringLiteral String
  deriving (Show, Generic, Data, Typeable, Eq)
```

Listing 26: Literal Type in Duet

The duet solution has advantages compared to our solution: the derivation of type classes can automatically be done because the underlying types already implemented the required functions. This eliminates a lot of the boilerplate code we have written for our simplified AST.

Expression also makes use of multiple polymorphic types (instead of just one like we have tried). This makes sense as it allows writing the code in a much more generic way.

We think that the Duet approach is the most elegant way to solve the problem. We did not rewrite the prototype to make use of this approach but see it as a valuable learning.

5.4.6 Support for more operators

Our Solution To apply operators like +, -, *, /, == and so on, we made Expression a derived instance of Num, Fractional and Eq.

Here, for example, is the implementation of the (+)-Operator:

```haskell
instance Num Expression where
        (+) (IntegerValue x) (IntegerValue y) = IntegerValue ((Prelude.+) x y)
        (+) (DoubleValue x) (DoubleValue y) = DoubleValue ((Prelude.+) x y)
        (+) (IntegerValue x) (DoubleValue y) = DoubleValue ((Prelude.+) (fromInteger x) y)
        (+) (DoubleValue x) (IntegerValue y) = DoubleValue ((Prelude.+) x (fromInteger y))
        (+) _ _ = InvalidExpression "+ not supported for this type"
```

Listing 27: Implementation of the plus operator for the simplified AST

As seen here, the result of an addition with incompatible types is an InvalidExpression.

More Generic Operator Application A problem of our code is that the implementations of +, - and * almost look the same. The only thing that changes is the operator itself.

We tried to make a more generic function for arithmetic operations. The signature would look something like this:

```haskell
applyNumOperator :: Num a => (a -> a -> a) -> Expression -> Expression -> Expression
applyNumOperator operator leftExpression rightExpression = {-...-}
```

Listing 28: Wanted approach to make a generic function to apply numeric operators

However, we could not find a way to implement such a function. The problem is, that we cannot pass a concrete Type like Double or Integer to a function that expects a Num a. This shows that a type class in Haskell cannot really be compared to what other languages know as an interface.

How this problem is solved in Duet It turned out that the approach we have chosen is very similar to Duet. Duet also implemented each operator twice: once for `IntegerLiteral` and once for `RationalLiteral`. These two are the only numeric types that Duet supports.

5.4.7 Show Substeps during Operator Application while Preserving Non-Strictness

Problem In the first version of the proof of concept, we always reduced the operands of an operator to normal form before applying the operator. This worked well when we only supported algebraic operators and the integer type. But now that we also support types such as list, tuple and so on, we cannot just reduce the operands to normal form before operator application – this would be strict behavior.

Example of the Problem An implementation like the following would (in this case) still lead to the correct result, but it would not reflect the non-strict behavior of GHC:

```
((1,(1 + 2)) == (2,(2 + 1)))
{-Apply Operator +-}
((1,3) == (2,(2 + 1)))
{-Apply Operator +-}
((1,3) == (2,3))
{-Apply Operator ==-}
False
```

Listing 29: Strict operator application

In the next example expression, such an implementation would result in an error despite the expression being valid in original Haskell:

```
(1.0,(1.0 + 2.0)) == (2.0,(1.0/0.0))
{- Apply Operator + -}
(1.0,3.0) == (2.0,(1.0/0.0))
{- Apply Operator / -}
InvalidExpression "cannot divide by 0"
```

Listing 30: Unwanted strict behavior in the proof of concept

Our Solution We omit this problem by not showing any sub-steps during operator application. We reduce each expression argument to normal form before passing it to the operator. But the behavior is still lazy because if the operator does not need to evaluate some arguments, `reduce` will not actually be called as long as we do not print the sub-steps:

```
(1.0,(1.0 + 2.0)) == (2.0,(1.0/0.0))
{-Apply Operator ==-}
False
```

Listing 31: Operator sub-steps are not printed

Possible Solution with Sub-steps and Non-Strictness To support the printing of sub-steps for an operator application, we are not allowed to reduce the operands to normal form before operator application. However, this implicates that we would have to implement our own operators like `==` and others because the operators must be able to do the reduction by themselves – if they need to do it. This means that it is not enough for an operator function to know the lefthand-side expression and the righthand-side expression because it also has to know the context and bindings to do a reduction if necessary. Additionally, operators like `==` either have to use `IO()` to print the sub-steps or add the reduction steps to a specific data structure.

If implemented, the sub-steps of an operator application could look like this:

```
(1.0,(1.0 + 2.0)) == (2.0,(1.0/0.0))
{-Apply Operator ==-}
(1.0 == 2.0) && ((1.0 + 2.0) == (1.0/0.0))
{-Apply Operator ==-}
False && ((1.0 + 2.0) == (2.0 + 1.0))
{-Apply Operator &&-}
False
```

Listing 32: How operator-substeps could be printed (concept)

5.4.8 Function Application without Eval

We want to be able to pass expression arguments to any function. Having that feature allows us to reduce function applications in one single step. This is useful if the substitution stepper can not understand the function-body itself because it uses features that we do not support (yet).

Applying functions in one single step could look like this:

```haskell
reverse [1, 2, 3, 4]
{-apply reverse -}
reverse [4, 3, 2, 1]
```

Listing 33: Single step reduction of function application

Of course, it would be preferred in this example to parse the function-body of `reverse` to show the sub-steps. But if this is not possible in some cases, we think a single step would be a adequate alternative.

So to pass an `Expression` to any function, we have to provide a converter function that converts the `Expression` into the type that is expected by the function that should be called. We came up with the following functions that take a function as well as 1, 2 or 3 parameters in the form of an `Expression`:

```haskell
applyOneArgument :: (a -> b) -> FunctionArgument a -> b
applyOneArgument function (unwrap, expression) = (function (unwrap
↪  expression))

applyTwoArguments :: (a -> b -> c) -> FunctionArgument a ->
↪  FunctionArgument b -> c
applyTwoArguments  function firstArgument secondArgument =
↪  applyOneArgument (applyOneArgument function firstArgument)
↪  secondArgument

applyThreeArguments :: (a -> b -> c -> d) -> FunctionArgument a ->
↪  FunctionArgument b -> FunctionArgument c -> d
applyThreeArguments function firstArgument secondArgument thirdArgument =
↪  applyOneArgument (applyTwoArguments function firstArgument
↪  secondArgument) thirdArgument
```

Listing 34: Functions to apply arguments

A `FunctionArgument` looks like this:

```haskell
type FunctionArgument a = ((Expression -> a), Expression)
```

Listing 35: FunctionArgument type

Now we can call any function like this:

```haskell
applyOneArgument (not) (expressionToBool, (BoolValue True))

expressionToBool :: Expression -> Bool
expressionToBool (BoolValue x) = x
```

Listing 36: Example of generic function/operator application with expression

We can also use functions that expect their arguments in the form of a type class, as long as `Expression` derives from this type class. For example: for functions that expect a `Num` type, we can just use the `id` function as converter:

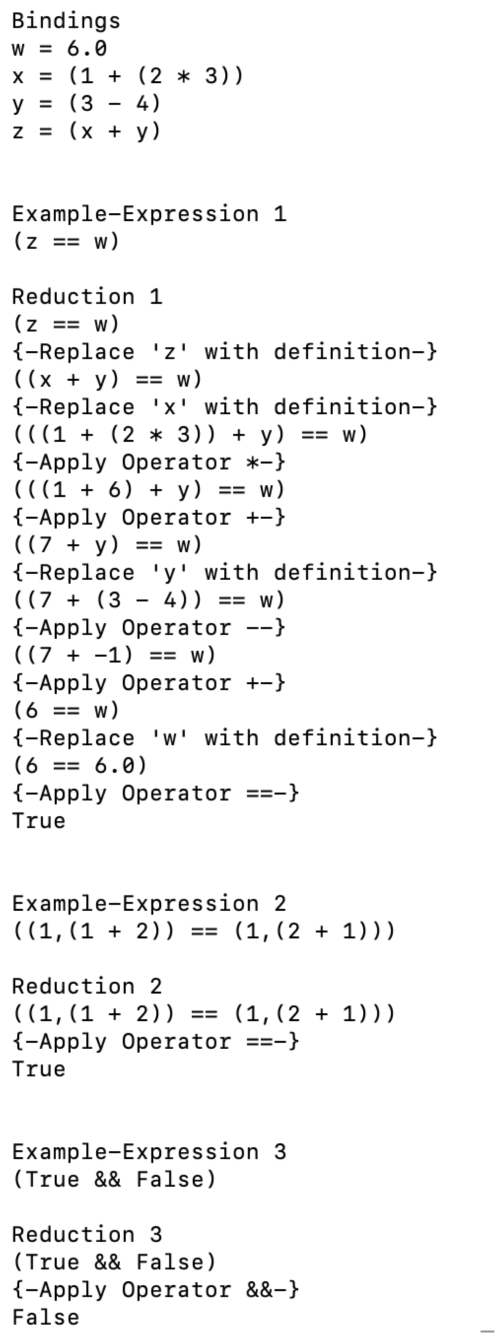
```haskell
applyTwoArguments (^) (id, (IntegerValue 2)) (expressionToInteger,
↪  (IntegerValue 3))
```

Listing 37: Usage of functions with type class parameter

We used this capability to add support for more operators like `&&` or `||`.

5.5 Result

Here are some example reductions done by the proof of concept that demonstrate the handling of different types and operators:



```
Bindings
w = 6.0
x = (1 + (2 * 3))
y = (3 - 4)
z = (x + y)


Example-Expression 1
(z == w)

Reduction 1
(z == w)
{-Replace 'z' with definition-}
((x + y) == w)
{-Replace 'x' with definition-}
(((1 + (2 * 3)) + y) == w)
{-Apply Operator *-}
(((1 + 6) + y) == w)
{-Apply Operator +-}
((7 + y) == w)
{-Replace 'y' with definition-}
((7 + (3 - 4)) == w)
{-Apply Operator --}
((7 + -1) == w)
{-Apply Operator +-}
(6 == w)
{-Replace 'w' with definition-}
(6 == 6.0)
{-Apply Operator ==-}
True


Example-Expression 2
((1,(1 + 2)) == (1,(2 + 1)))

Reduction 2
((1,(1 + 2)) == (1,(2 + 1)))
{-Apply Operator ==-}
True


Example-Expression 3
(True && False)

Reduction 3
(True && False)
{-Apply Operator &&-}
False
```

Figure 20: Console output from the proof of concept (v2)

5.6 Discussion and Further Procedure

5.6.1 Use of the GHC Frontend

When we started working on the proof of concept, we were thinking about three alternatives that we could take:

1. Work with the GHC frontend (typechecker AST) and provide our backend to reduce expressions
2. Work with the Duet language and add more features
3. Create a user interface where users can enter an expression and apply their own reduction steps. This solution would use GHC to check the validity of the user input.

With the proof of concept, we tried out the first option. We think that this approach is theoretically doable, although it would not be possible to support the whole Haskell-Syntax in a reasonable time. Also, we had troubles to work with the typechecker AST and to convert it into our own simplified AST because it is enormously large.

5.6.2 GHC Typechecker AST vs Core AST

After presenting the proof of concept, we re-discuessed the idea of stepping Haskell Core instead of Haskell Code with our advisor and our external examiner. At the beginning of the project, we rejected the idea of stepping Haskell Core because Haskell Code is easier to read for the user. However, we realized that working with Haskell Core would give us indeed many advantages:

1. There are substantially less syntactic constructs. That means that even if we do not support the full syntax of Haskell we could support a larger percentage of the syntax, making the tool usable in more situations.
2. For the user of the substitution stepper it could be interesting to see how Haskell input is converted into Haskell Core. The tool would also be interesting for debugging language extensions or similar projects.
3. For someone who can read Haskell Code, Haskell Core is a bit more difficult to read but still understandable.
4. The reduction will be less experimental.

Working with Core does not give us the same ability to do equational reasoning but as we have seen in the analysis chapter (4.4.5), there are already solid solutions to education like the Haskell Expression Evaluator [16].

For all those reasons, we decided that stepping through Haskell Core is better than stepping original Haskell. So for the real implementation we will work with Haskell Core. This means that we cannot reuse much of our code written for the proof of concept. However, we think that the proof of concept still gave us many learnings that we will be able to apply for the real stepper.

5.6.3 Use of a Simplified AST

For the proof of concept, we created our own `Expression` type (simplified AST). There are already some advantages and disadvantages of this approach listed on top of this chapter. For the prototype, handling our own `Expression`-Type was simple, but it would become more complicated the more features we support.

The alternative would be to use the AST that we get from GHC and do the reduction directly on this structure. For the typechecker AST, this would be very complicated. Since we already struggled to convert the AST into our custom structure, it would be even more complex to apply reduction rules on the typechecker AST itself as we would have to understand all of the fields and how they are used in order to construct a valid tree.

However, working with Haskell Core, the AST is much simpler and easier to understand.

Considering the pros and cons of using our own data structure, we decided to first work with a custom data structure that is very similar to Core and later work with the original `CoreExpr` type.

5.6.4 Representation of the Heap

We also discussed if we should somehow show a representation of the heap to the user while doing the reduction. The advantage would be that we would be able to show concepts like sharing. However, we decided that for now we will not show the heap and maybe even build a solution that does not support sharing.

5.6.5 New Architecture

After those decisions, the new architecture for the stepper will look like this:

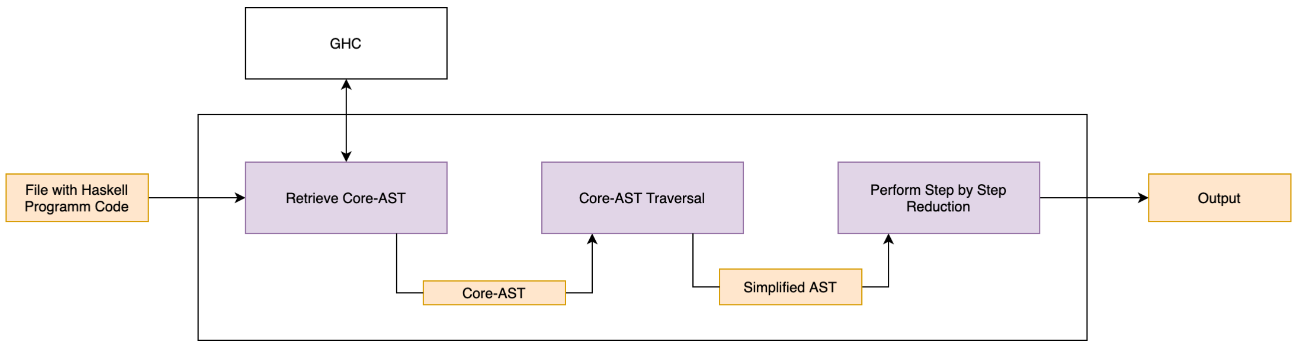


Figure 21: New architecture with Haskell Core

5.6.6 Conclusion

We will have to change a part of the underlying architecture for the final implementation. The proof of concept has forced us to think about our software design and the implementation challenges. This gave us valuable insights about how we can build the final solution.

6 Implementation

6.1 Planning of the Construction Phase

The construction phase started rather late because of the long time spent on the analysis. We experimented with different approaches and finally came to the conclusion that stepping Haskell Core is the most feasible approach. The detailed planning for this phase is written in a separate document, but here is a summary of what we expect the implementation to include:

- **Architecture**: Retrieve Haskell Core code from the GHC for a user-defined input file
- **Stepping**: Use of built-in helper functions from GHC whenever possible. Work with a data structure that is very close to the Core type, ideally work with the Core type itself
- **Supported Haskell Syntax**: Support the same language-features as the prototype and add more features like Lamda, Function Application, Pattern Matching, Recursion, Types, Let-Statements, Polymorphism, Cast, Coercion...
- **Prelude**: Add well-known functions from the Prelude, so they can be stepped through too
- **Configurability**: Let the user choose how verbose the output should be
- **Testing**: Cover more of the functionality with automatic tests
- **User Interface**: Create an easy-to-use user interface

6.2 Understanding the Core Type

6.2.1 Expression Type in Core

As a first step, it was important to understand how expressions are represented in the Core Type of GHC. It is impressive that Core only has 10 constructors to represent the whole syntax [21]:

```haskell
data Expr b -- "b" for the type of binders,
  = Var     Id --variables
  | Lit   Literal --literals
  | App    (Expr b) (Arg b) --function application
  | Lam    b (Expr b) --lamdas
  | Let    (Bind b) (Expr b) --let bindings
  | Case   (Expr b) b Type [Alt b]
  | Type   Type
  | Cast   (Expr b) Coercion
  | Coercion Coercion
  | Tick   (Tickish Id) (Expr b) --not important for us

data Bind b = NonRec b (Expr b)
            | Rec [(b, (Expr b))]

type Arg b = Expr b

type Alt b = (AltCon, [b], Expr b)

data AltCon = DataAlt DataCon | LitAlt  Literal | DEFAULT
```

Listing 38: Expr type for GHC Core

This means that GHC already does a lot of work for us: while in original Haskell there are many ways to write the same program, there is no more syntactic sugar in Core. This is good news for the stepper: if we support most of those constructors, we can support a large part of the Haskell language.

Looking at Core is even more impressive if it is compared with the HsExpr data structure, which is used to represent original Haskell and has significantly more constructors.

6.2.2 Polymorphic Parameter b

The polymorphic parameter `b` is the type of binders on an expression. Working with the Core language in GHC, it is of type `Var`:

```haskell
type CoreExpr = Expr CoreBndr
type CoreBndr = Var
```

Listing 39: CoreExpr type for GHC Core

The `Var` type contains a lot of information. What is most important for us is the name of a variable.

6.2.3 Literals

Looking at the `literal` type, it is evident that there are only very few base types [21]:

```haskell
data Literal
        = LitChar    Char
        | LitNumber !LitNumType !Integer Type
        | LitString !ByteString
        | LitNullAddr
        | LitRubbish
        | LitFloat   Rational
        | LitDouble  Rational
        | LitLabel   FastString (Maybe Int) FunctionOrData
        deriving Data
```

Listing 40: Literal type for GHC Core

Things like `Bool` or composite types like lists or tuples are missing. They are not needed because Core represents them differently, in fact, as application on a constructor.

6.2.4 Getting the Core Code

To get an idea of how Core looks, it was useful to convert some examples of Haskell source code to Core. There are multiple ways to achieve this:

- one can work with the ghc-core and then make use of the package programmatically:

```
cabal install ghc-core
```

Listing 41: Installation of the GHC Core package

- or one could use GHCi to print out Core representations directly on the console:

```
stack ghci --ghci-options "-ddump-simpl -dsuppress-idinfo
 ↪  -dsuppress-coercions -dsuppress-type-applications -dsuppress-uniques
 ↪  -dsuppress-module-prefixes"
```

Listing 42: Usage of GHCi to get Core

As seen in the command, there are many flags that are used to suppress some less interesting information.

In the next chapter there are some original Haskell code examples with their representation in Core. We do not show every aspect of Core here, but it should give an idea of how Core looks.

6.2.5 Core Example - id: Handling of Types, Parametric Polymorphism

```haskell
id :: a -> a
id x = x
```

Listing 43: id function (Haskell)

```haskell
id :: forall a. a -> a
id = \ (@ a) (x :: a) -> x
```

Listing 44: id function (Haskell Core)

It is interesting to see how Core handles polymorphic types. The first parameter of the `id` function is a type `a` (for example, one could pass `@ GHC.Types.Char`) and only then the ‚real‘ parameter (for example `'c'`) can be passed. Types are treated just like normal variables. This behavior is inspired by the **second order lambda calculus** [22].

6.2.6 Core Example - sayMe: Pattern Matching with case

```haskell
sayMe :: Integer -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe x = "Not between 1 and 3"
```

Listing 45: sayMe function (Haskell)

```haskell
sayMe :: Integer -> String
sayMe
  = \ (ds_d1py :: Integer) ->
      case == @Integer $fEqInteger ds_d1py 1 of {
        False ->
          case == @Integer $fEqInteger ds_d1py 2 of {
            False ->
              case == @Integer $fEqInteger ds_d1py 3 of {
                False -> unpackCString# "Not between 1 and 3"#;
                True -> unpackCString# "Three!"#
              };
            True -> unpackCString# "Two!"#
          };
        True -> unpackCString# "One!"#
        }
```

Listing 46: sayMe function (Haskell Core)

`case` is a structure that is frequently used in Haskell Core as it replaces multiple language features (syntactic sugar) from Haskell like `if-then-else`, function definitions with pattern matching or guards.

In the example, it can be seen how the pattern matching is converted into a nested `case` statement. Each of the case statements checks the equality of the parameter (which was named `ds_d1py`) with an `Integer` literal.

6.2.7 Core Example - logicalOperator: Mutli-Argument Function Bindings

```haskell
logicalOperator :: Bool -> Bool -> Bool
logicalOperator x y = not x && y
```

Listing 47: Multi-argument function (Haskell)

```haskell
logicalOperator :: Bool -> Bool -> Bool
logicalOperator = \ (x :: Bool) (y :: Bool) -> && (not x) y
```

Listing 48: Multi-argument function (Haskell Core)

Because of the pretty-printing used in this example, it looks like Core would support lamdas that take multiple parameters. However, this expression is actually a nested lamda where each lamda only takes a single parameter. A more accurate representation would look like this [8]:

```haskell
logicalOperator =
  \(x :: Bool) ->
  \(y :: Bool) ->
  (&&) (not x) y
```

Listing 49: Nested multi-argument function (Haskell Core)

6.2.8 Core Example - reverseList: Pattern Matching, Handling of Lists and Recursion

```haskell
reverseList :: [Integer] -> [Integer]
reverseList [] = []
reverseList (a:bc) = reverseList bc ++ [a]
```

Listing 50: reverseList function (Haskell)

```
Rec {
reverseList [Occ=LoopBreaker] :: [Integer] -> [Integer]
reverseList
  = \ (ds_d1pq :: [Integer]) ->
      case ds_d1pq of {
        [] -> [] @Integer;
        : a bc -> ++ @Integer (reverseList bc) (: @Integer a ([]
        ↪  @Integer))
      }
          end Rec }
```

Listing 51: reverseList function (Haskell Core)

Again, there is a `case` statement instead of the pattern matching. It is also interesting to see how lists are represented in Core. A list like `[1, 2, 3]` is represented as:

```
: @Integer 1 (: @Integer 2 (: @Integer 3 ([] @Integer)))
```

Listing 52: Representation of a list in Haskell Core

As we see in the example, Core does not have a special type for lists. Instead, the list is represented as a repeated application of the Cons-Operator (`:`).

6.2.9 Core Example - double: Type Classes

```haskell
double :: Num a => a -> a
double x = x + x
```

Listing 53: double function (Haskell)

```haskell
double :: forall a. Num a => a -> a
double
= \ (@a) ($dNum_a1oj :: Num a) (x :: a) -> + @a $dNum_a1oj x x
```

Listing 54: double function (Haskell Core)

[8]

Core does not have the concept of type classes. Instead, as we see here, functions that expect a parameter to be in a specific type class in Haskell get a new dictionary parameter (`dNum_a1oj`) in Core. This dictionary parameter contains implementations of the functions which are expected to be supported by the typeclass. (For example, for the typeclass `Num`, this would be operators like `+`, `-`, `*` and so on).

6.2.10 Optimization of Core Code

To make the output of the stepper as readable as possible, we work with the Core-output that is not optimized. In the original GHC-pipeline, Core code is transformed over and over again, making the code more performant but also more difficult to read. This is called the simplifier-phase. It includes: Inlining, elimination of dead code, case merge and much more. Developers can also write simplification-rules for their code, which are then used by the GHC [22].

However, this is not of interest for the stepper. The stepper works with the Core code immediately after the desugaring phase, still before any optimization happens. Like this, the Core code should still be more similar to what the user has entered.

Even though the unoptimized Core code is still pretty understandable, we decided to create a pretty printer that prints out Core code in a way that looks like original Haskell code.

6.2.11 Summary - Some important Things to know about Core

Here are some of the most significant changes in Haskell Core compared to original Haskell [22] [23]:

- Application always only has one single argument. Application of a function that expects more than one argument is represented as nested application.
- `case` is used a lot.
- `case` is where actual evaluation/reduction happens (nodes are removed from the AST).
- The `let` constructor is used for allocation (for example, whenever `where` is used in original Core).
- As Haskell itself, Core is non-strict (but `case` is strict in Core).
- Types are passed around like normal parameters.
- Infix operators are represented as function application.
- Every binder is explicitly annotated with its type (implicitly in original Haskell).

6.3 Retrieving Core from GHC

The first implementation step was to take a Haskell file as input and then use GHC as a library to convert it to Core. Using the GHC-library, the different intermediate steps are retrieved. For debugging purpose, the parser AST, the type checker AST and the Core AST are written out on a generated file.

This is the type provided from the GHC-library:

```haskell
type CoreSyn.CoreProgram = [CoreBind]
```

Listing 55: CoreProgram type definition

`CoreBind` itself is defined like this:

```haskell
type CoreBind = Bind CoreBndr
type CoreBndr = Var
```

Listing 56: CoreBind and CoreBndr type definition

A binding can either be recursive or non-recursive:

```haskell
data Bind b = NonRec b (Expr b) | Rec [(b, (Expr b))] deriving Data
```

Listing 57: Bind type definition

In simple terms, a binding is an assignment of a name to an expression. For example, in the Expression `x = 1`, `x` would be the non-recursive binding at the top-level and `1` would be the expression. Bindings are used for local `let`-statements as well.

(Note: The `Expr` type is described at the beginning of this Chapter.)

Once the Core AST is retrieved, we could use GHC's built-in pretty printer to take a look at the AST itself:

```haskell
showOutputable (pprCoreBindings coreAst)
```

Listing 58: Printing a Core AST

Still, it was a challenge to understand which types and constructors the various syntax constructs had. We added a function for debugging which takes a Core Expression - this expression (which is actually a tree) is then traversed and for every node, the corresponding constructor and additional information is printed.

6.4 Usage of Built-in Functions from GHC

6.4.1 The GHC Api

It is our goal to use as much of the built-in tools from GHC as possible. This not only reduces our development time, but also helps us to stay more compatible with GHC even if a new GHC version will be released.

The Core part of GHC consists of +/- 60 separate modules. Many of them provide helpful functions that work on the `Expr`-type. We went through the GHC source code to find out which functions are helpful for the implementation of the stepper.

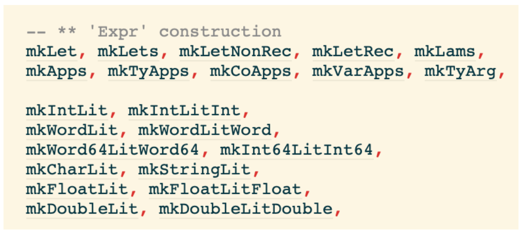
6.4.2 Maker Functions

Although the base `Expr`-Type is rather simple, working with Core can still be demanding.

To give an example, something simple like a Core Literal `1` is difficult to create from the ground up. In Core, `1` would be a Literal instance with the constructor `LitNumber`. `LitNumber` itself has multiple subtypes. Choosing the right subtype depends on the target machine where the code runs. Furthermore, the `LitNumber` constructor takes a `Type` as constructor argument, which itself is a nested data structure again.

This example illustrates that it is not feasible to create Core types from the ground up, especially if the resulting Core type is dependent on the type of computer being used.

Fortunately, GHC provides some maker-functions that can be used to create Expression Instances. This screenshot shows some of the functions that are exported by the GHC.Core-Module:



```
-- ** 'Expr' construction
mkLet, mkLets, mkLetNonRec, mkLetRec, mkLams,
mkApps, mkTyApps, mkCoApps, mkVarApps, mkTyArg,

mkIntLit, mkIntLitInt,
mkWordLit, mkWordLitWord,
mkWord64LitWord64, mkInt64LitInt64,
mkCharLit, mkStringLit,
mkFloatLit, mkFloatLitFloat,
mkDoubleLit, mkDoubleLitDouble,
```

Figure 22: Some maker helper functions from GHC Core

Thanks to those helper functions, some Core expressions can be easily created. For example, the `Integer` literal:

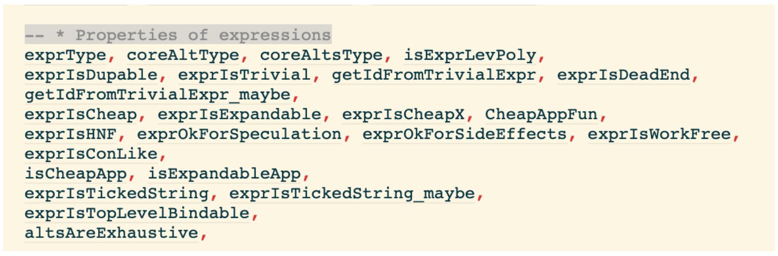
```
mkLitInt64 1
```

Listing 59: Creation of a Core Integer Literal

6.4.3 Information Retrieval

Similarly, it is useful to have functions to read some properties of expressions. This helped us a lot to implement the stepper-functionality.

Here, for example, are some functions from the GHC.Core.Utils module:



```
-- * Properties of expressions
exprType, coreAltType, coreAltsType, isExprLevPoly,
exprIsDupable, exprIsTrivial, getIdFromTrivialExpr, exprIsDeadEnd,
getIdFromTrivialExpr_maybe,
exprIsCheap, exprIsExpandable, exprIsCheapX, CheapAppFun,
exprIsHNF, exprOkForSpeculation, exprOkForSideEffects, exprIsWorkFree,
exprIsConLike,
isCheapApp, isExpandableApp,
exprIsTickedString, exprIsTickedString_maybe,
exprIsTopLevelBindable,
altsAreExhaustive,
```

Figure 23: Some property functions from GHC Core

6.4.4 Functions that have been especially helpful

Writing the stepper, there were some functions that have been especially useful for us and helped us to keep our codebase smaller.

- There is a function `exprIsHNF` in the Module GHC.Core that takes an expression and returns if this expression is in head normal form. For the stepper, this allows to check if an expression should be further stepped or if the stepping is complete. For the first prototype, this functionality had to be implemented from the ground up. Working with Core, we already got it for free.
- The modules GHC.Utils.Outputable and GHC.Core-PPr provide functions to pretty print Core Expressions.
- The function `collectArgs` in the module GHC.Core has the type signature `Expr b -> (Expr b, [Expr b])`. If there is a nested `App` expression, the function collects all the arguments passed to the function and returns them in a one-dimensional list. For the stepper, this was used to decide which reduction step should be used next and to evaluate functions that cannot be stepped (see Chapter 6.14.6).

6.4.5 No Functions directly related to Stepping found

It was disappointing that we did not find ready-to-use reduction-rules in the GHC. We were looking for functions that would — for example — apply an argument expression to a lamda expression or similar functionality.

However, it makes sense that such functions must not inevitably exist for Core as the real evaluation only happens at the STG-stage. For the stepper, it meant that we had to implement the reduction rules ourselves. (See Chapter 6.8)

6.5 First Draft: Stepping on a Simplified Core AST

6.5.1 Why did we first stick to a Simplified Core AST instead of working on the Core AST directly?

Although our advisor told us that it would be ideal to work on the original `Expr` type itself, we started the implementation with our own simplified Core-like AST. At first, we were skeptical as to whether stepping on Haskell Core was possible or not, and we have already made some experience with a simplified AST to create the prototype. Our biggest concern was, as described above, that while it is rather easy to read out some properties from existing Core expressions, it can become much more difficult to create a valid expression instance from the ground up. This is why we decided to first convert the user input file into Haskell Core (using the GHC-library) but then transform this Haskell Core AST into our simplified AST. This approach allowed us to first work on the „hearth" of the application, which the stepper-logic. During the implementation of the simplified AST, we tried to make the structure very close to the Core structure itself. We always had the idea in mind that this intermediate-structure could later be removed and replaced with the `Expr` type from Core itself.

Outlook to Chapter 6.6: Indeed, it was later possible to completely remove the simplified AST and implement the stepper on the Core type directly. This change was actually simpler than expected, and much of the code written was adopted unchanged.

6.5.2 First Draft: The Simplified Core AST

Our first draft of a simplified expression type looked like this:

```haskell
data ExpressionS
  = VarS String --for example "x" or "+"
  | LitS LiteralS --for example "4"
  | AppS {expressionS :: ExpressionS, argumentS :: ExpressionS} --for
  ↪  example "+ 1"
  | LamS {parameterS :: String, expressionS :: ExpressionS} --for example
  ↪  "\x -> ...""
  | CaseS {expressionS :: ExpressionS, alternativesS :: [AltS]} --for
  ↪  example "case True of {True -> "True"; False -> "False"}
  | TypeS --not implemented
  | MultiArgumentAppS {name :: String, argumentsS :: [ExpressionS]} --does
  ↪  not exist in original Core
  | InvalidExpression String --for example, InvalidExpression
  ↪  "unsupported expression"

data LiteralS
  = LitCharS Char
  | LitNumberS Integer
  | LitStringS String
  | LitFloatS Rational
  | LitDoubleS Rational
  | InvalidLiteral String

type AltS = (AltConS, [String], ExpressionS)

data AltConS
  = DataAltS String --pattern is a constructor, for example ":"
  | LitAltS LiteralS -- pattern is a literal, for example 5
  | DefaultS -- pattern is "_"
```

Listing 60: Simplified Core AST type

At the first sight, this structure looks very similar to the original `Expr` type. However, there were some key differences:

- Instead of nested data types like the `Expr` type, we often used primitive types. For example, while a `Var` in Core contains a lot of information (such as `varName, uniqueIdentifier, varType, scope...`) we simply used a `String` which only contained the name of a binding. The same goes for Literals as well.
- Type information was completely discarded
- A new constructor `MultiArgumentAppS` was introduced. This constructor was used to

represent applications with more than one argument. (This is not Core-like but made the internal handling of nested applications easier.)

- A new constructor `InvalidExpression` was introduced. This constructor was already used in the prototype to represent invalid states — for example, if the stepper does not support a specific language feature.

In general, the simplified AST contains only the information we needed for printing and stepping. All the other information from the original `Expr` type was discarded.

6.5.3 First Draft: Conversion from Core AST to the Simplified AST

The implementation of the conversion method was simple. Using the helper functions from GHC, it was simple to extract all the information required from the `Expr` type and fill it into the `ExpressionS` type.

6.5.4 First Draft: Pretty Printing

The printing function implemented for the `ExpressionS` type prints expressions in a form that look much like original Haskell. Here is an example: The first listing is a function in its original Haskell form. The second listing shows the same function transformed to and printed in Haskell Core. The third listing is the same function transformed and printed as our simplified AST:

```haskell
x = 1 + (2 * 3)
```

Listing 61: Algebraic expression (pretty-printed Haskell)

```
x = + @Integer $fNumInteger 1 (* @Integer $fNumInteger 2 3)
```

Listing 62: Algebraic expression (pretty-printed Haskell Core)

```
x = (+ 1) ((* 2) 3)
```

Listing 63: Algebraic expression (pretty-printed simplified Core AST)

6.5.5 First Draft: Rules for the stepper

Next, we implemented the main functionality: A function that takes the context and an `ExpressionS` and applies one single reduction-step to the expression, which is then returned as a result. Here is the signature of this function:

```haskell
applySingleStep :: [BindS] -> ExpressionS -> ExpressionS
applySingleStep bindings expression = {-...-}
```

Listing 64: Type signature of applySingleStep function

(Note: A `BindS` is just a simple type alias for `(String, ExpressionS)`. The list `[BindS]` passed to the function contains the whole surrounding context of the expression (which means all functions and variables defined by the user).

This is a simplified version of the implemented algorithm. For visualization purpose, the code is shown in a simplified version:

```haskell
--replace VarS with the corresponding expression
applyStep bindings (VarS name) = findBindingWithName bindings name

--lamda reduction
applyStep bindings (AppS (LamS parameter expression) argument) =
↪  deepReplaceVarWithinExpression parameter argument expression

--applyStep for left-hand side of application
applyStep bindings (AppS firstArgument secondArgument) = AppS (applyStep
↪  bindings firstArgument) secondArgument)

--reduce pattern pathing
applyStep bindings (CaseS expression alternatives) =
  if canBeReduced expression
    then CaseS (applyStep bindings expression) alternatives
    else findMatchingPattern expression alternatives

--reduction rule not yet implemented
applyStep bindings _ = InvalidExpression "No reduction implemented for
↪  this type of expression")
```

Listing 65: Implementation of applyStep for the simplified Core AST

Furthermore, we added support for the most well known algebraic and logical operators (not shown in the listing).

6.5.6 First Draft: Example

Even with those few simple reduction rules, it was surprising to how many expressions could be stepped. The following is a simple example. This is the input-source file:

```haskell
sayMe :: Integer -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe x = "Not between 1 or 2"

x = sayMe 4
```

Listing 66: Simplified Core AST stepper example input

Stepping the binding x prints out the following result to the console: (some sub-steps are skipped)

```
(sayMe 1)

{-Replace 'sayMe' with definition-}

(\ds_d1oa -> (case ((== ds_d1oa) 1) of {
        False -> case ((== ds_d1oa) 2) of {
                False -> "Not between 1 or 2";
                True -> "Two!";
        };
        True -> "One!";
}) 1)

{-Lamda Application-}

case (== 1 1) of {
        False -> case ((== ds_d1oa) 2) of {
                False -> "Not between 1 or 2";
                True -> "Two!";
        };
        True -> "One!";
}

{-apply == -}

case True of {
        False -> case ((== ds_d1oa) 2) of {
                False -> "Not between 1 or 2";
                True -> "Two!";
        };
        True -> "One!";
}

{-Replace with matching pattern-}

"One!"

{-reduction complete-}
```

Listing 67: Simplified Core AST stepper example output

6.6 Final Architecture: Removal of the Simplified AST

6.6.1 Introduction

Now that some easy expressions could be stepped, the next implementation step was to remove the simplified Core AST (`ExpressionS`) and to replace it with the original Core AST (`Expr Var`):

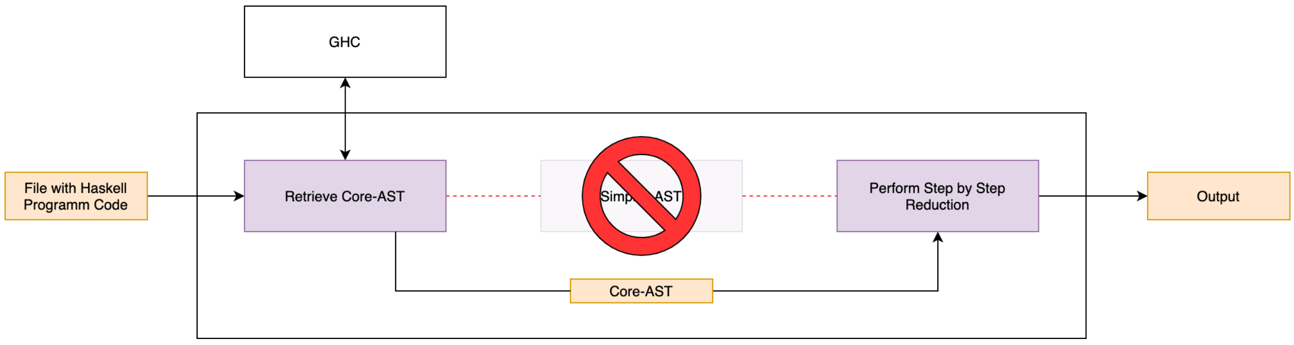

GHC
File with Haskell Programm Code
Retrieve Core-AST
Simp AST
Perform Step by Step Reduction
Output
Core-AST

Figure 24: Architecture without the simplified Core AST

Even though we were first very skeptical if a reduction on original Core code is feasible to do in a reasonable time, the transition from the simplified AST solution to the original Core AST solution went surprisingly well. This was mainly because during the reduction, Nodes are either replaced by other Nodes or completely removed from the AST. This is much easier compared to creating new Nodes in the Core AST. It is very rarely the case that new Nodes have to be created.

However, to get rid of the simplified AST, there were some challenges to overcome, which are described in the next chapter.

6.6.2 Challenges and Solutions

Problem / Challenge	**Solution**
The `Var` Type from `Expr` contains much more information than the String that is used instead for the simplified AST	There is no need to create new instances of `Var`. Only existing `Var`s inside expressions will be read and replaced by other Expressions.
After evaluating operators and functions (such as `+`) the result has to be converted back into a Core Literal.	There are helper functions such as `mkLitInt64` or `mkLitString` that allow us to create `Literal` instances. However, types that are not built-into Core (such Bool) will be more difficult to support.
Floating numbers are represented as `Rational` in Core `(type Rational = Ratio Integer)`. The results of algebraic calculations have to be converted into that type.	There is already a function that does that: `toRational Real a => a -> Rational`
There is not such thing as an `InvalidExpression` constructor for Core. What should we return from a function if something goes wrong while stepping? (for example, if a feature is not implemented?)	The workaround here is to use `Maybe Expr` in every function that might not result in a valid expression. This makes the code a bit harder to read but thanks to the monad syntax (`do` expressions) this should not be a big deal.
Type information has to be handled correctly and cannot just be thrown away.	Types in Core are used as if they were just normal variables / parameters. There is not much that has to be specially considered. One difference is that when we evaluate a function or operator like (`+`), there is not only the numeric arguments but also the type arguments.
Let, Cast and Coercion constructors were not implemented for the simplified AST.	The stepper can return `Nothing` if one of those constructors is encountered. Later, those constructors should be supported as well.

Table 2: Core AST: implementation challenges

6.7 Stepper Loop

The stepper loop is the driver of the reduction, responsible for the reduction and the printing of the reduction steps. The function takes an expression as argument and repeatedly calls `applyStep` until the expression is in head normal form. Afterwards, it checks if the reduction can be reduced even more to normal form and applies the necessary reductions.

6.8 Reduction Rules

6.8.1 List of Reduction Rules

Here are the implemented reduction rules:

Step for Pattern: (Var name)

- **Rule**: Search for a binding with the same Var. Then replace with the expression of the found binding.
- **Example Input (simplified)**:

```
addOne
```

- **Example Output (simplified)**:

```
(\x -> (+) x 1)
```

Step for Pattern: (App (Lam parameter expression) argument)

- **Rule**: Lamda Application
- **Example Input (simplified)**:

```
( \x -> (+) x 1 ) 1
```

- **Example Output (simplified)**:

```
(+) 1 1
```

Step for Pattern: (App expression argument)

- **Rule**: applyStep on expression
- **Example Input (simplified)**:

```
addOne 1
```

- **Example Output (simplified)**:

```
(\x -> (+) x 1 ) 1
```

Step for Pattern: (Case expression binding caseType alternatives)

- **Rule**: If expression is not in head normal form: applyStep on Expression, else: replace with matching pattern
- **Example Input 1 (simplified)**:

```
case ((==) 1 1) of {
   True -> ...;
   False -> ...
}
```

- **Example Output 1 (simplified)**:

```
case True of {
   True -> ...
   False -> ...
}
```

- **Example Input 2 (simplified)**:

```
case True  of {
   True -> 1;
   False -> 2
}
```

- **Example Output 2 (simplified)**:

```
1
```

- **Example Input 3 (simplified)**:

```
case (: 1 [2]) of
   : a b -> a
   : DEFAULT -> ...
```

- **Example Output 3 (simplified)**:

```
1
```

Step for Pattern: (Let binding expression)

- **Rule**: replace every occurrence from the local bound variable inside the expression with its definition. If the binding is recursive, it means that the body of the `let`-statement at some point references itself. In this case, the reference should not be replaced by its definition immediately as this would result in an infinite loop.
- **Example Input (simplified)**:

```
let {
  x :: Integer
  x = 1
} in (*) x x
```

- **Example Output (simplified)**:

```
(*) 1 1
```

Step for Pattern: (Cast expression cohersion)

- **Rule**: The `cohersion`-part of the expression is simply removed, and the reduction is continued on the `expression` only.
- **Example Input**:

```
($cchange @a)
`cast` (Sym (N:Navigatable[0] <Direction a>_N)
        :: Coercible (Direction a -> Direction a) (Navigatable (Direction
        ↪  a)))
```

- **Example Output (simplified)**:

```
($cchange @a)
```

6.8.2 Background

Those reduction rules follow the concept of non-strictness. That means that arguments are only evaluated when they are needed.

However, there is one exception which is not shown in the table. Whenever there is an `App expression argument` and the expression cannot be resolved, all arguments are collected and eagerly evaluated. (As described in Chapter 6.14.6, „Unsteppable / Unparsable Functions“). This is the case if the expression is a `Var` that points to a function that cannot be stepped. (This could be a function that uses syntactic constructs that the stepper cannot parse, or if it is a basic operator like `+`). It can also be the case for nested applications if the leftmost innermost application expression is a `Var` that points to a function that cannot be stepped.

Here is an example of `App` where the arguments have to be eagerly evaluated (because the `==` operator cannot be stepped):

```
(== 1) (neg 1)
```

Listing 68: Nested application example

This expression is a nested `App` in the form of `App (App Var Lit) (App Var Lit)`

If the stepper encounters such an expression, it uses the built-in GHC function `collectArgs` to convert the application into a tuple of type `(Expr, [Expr])`. The first element in the tuple is the function or operator, and the second element in the tuple is a list of arguments. If the

App-expression was nested, the argument-list would contain multiple arguments, if the App-expression was not nested, the argument-list contains only one single argument. The example above would now look like this (conceptually) after collecting the arguments:

```
((==), [1, neg 1])
```

Listing 69: Function and its collected arguments

Next, the stepper checks if some arguments are not yet in head normal form. If this is the case, applyStep is called on those arguments until every argument is in head normal form.

The example above would look like this after the reduction of the arguments (conceptually):

```
((==), [1, -1])
```

Listing 70: Function and its collected and fully reduced arguments

Finally, the function is evaluated. In the example, the result of the evaluation would be the Bool constructor False.

6.9 Basic Operations and Functions on Literal Types

Not every expression can be stepped — for example, a comparison of two Integers:

```
1 == 2
```

Listing 71: Equality on Integers in Haskell

```
==
  @Integer
  $fEqInteger
  (@Integer 1)
  (@Integer 2)
```

Listing 72: Equality on Integers in Core

Therefore, we implemented a function `evaluateUnsteppableFunctionWithArguments` that takes a function or operator name (represented as a `String`) and a list of expressions (arguments). The function then returns the evaluation of this expression. It is implemented without the use of `eval` which means it is secure and without side effects.

The function `evaluateUnsteppableFunctionWithArguments` is called whenever the stepper finds a function or operator application and cannot resolve the body of this function or operator. This is most likely the case when the function or operator cannot be represented in Haskell Code (this is the case for the equality operator for `Integer` types, as seen in the example above).

6.10 Support for Custom Types

A type instance in Core is represented as an application. For example, the expression `Maybe 5` is an `(App expr arg)` in Core, where `Maybe` ist the expression and `5` is the argument. This behavior made it easier to implement support for custom types and type classes.

The stepper supports custom datatypes, custom type classes, implementation of custom or existing type classes for custom types as well as automatic derivation of type classes.

6.11 Support for Type Classes

The biggest challenge was to extract the right type class function binding for the right type. For example, here we have a custom data type `Optional` that automatically derives from `Eq`. It looks like this:

```
data Optional a = None | Some a deriving (Eq)
```

Listing 73: Definition of a custom Optional type

Imagine that we make a comparison on two instances of `Optional`:

```
(Some 1) == (Some 2)
```

Listing 74: Equality on a custom Optional type in Haskell

This is the Core representation of the Haskell expression:

```
==
  @(Optional Integer)
  ($fEqOptional @Integer $fEqInteger)
  (Some @Integer 1)
  (Some @Integer 2)
```

Listing 75: Equality on a custom Optional type in Core

The actual implementation of the `==` operator for the `Optional` type is defined in the binding `$fEqOptional`. `$fEqOptional` is a function dictionary that contains every function required by the `Eq` type class. Here, it points to a lamda expression that contains the implementations for `==` and `/=`:

```
\ (@a) ($dEq_a1nN :: Eq a) ->
C:Eq @(Optional a) ($c== @a $dEq_a1nN) ($c/= @a $dEq_a1nN)
```

Listing 76: Core representation of $fEqOptional

This is how the `==` operator for the `Optional` type is actually generated:

```
\ (@a)
  ($dEq_a1nN :: Eq a)
  (ds_d1oL :: Optional a)
  (ds_d1oM :: Optional a) ->
  case ds_d1oL of {
    Some x ->
      case ds_d1oM of {
        Some y -> /= @a $dEq_a1nN x y;
        None -> /= False
      };
    None ->
      case ds_d1oM of {
        Some _ -> /= False
        None -> /= True
      }
  }
```

Listing 77: Core representation of $c== for Optional (simplified)

When using custom types with type classes, the stepper has to look for the right implementation of the function or operator that is being used. If multiple data types are used, multiple bindings might have the same name (there might be multiple `$c==`-bindings, for example). Therefore, it is always necessary to check the type signatures to choose the right implementation for the right type.

6.12 Error Handling

The `Maybe` type is used whenever a function might not have a result. In that case, the function returns `Nothing`. However, this is not the same as an error, it just means that a feature or specific syntax is not supported (yet) by the stepper. If this is the case, the stepping is canceled and the user is notified with a message („unsupported expression“).

Errors are thrown when a precondition is violated. Errors should never occur during runtime. The occurrence of an error at runtime means that there is a bug in the implementation.

6.13 Support for the Standard Haskell Prelude from the Haskell 2010 Report

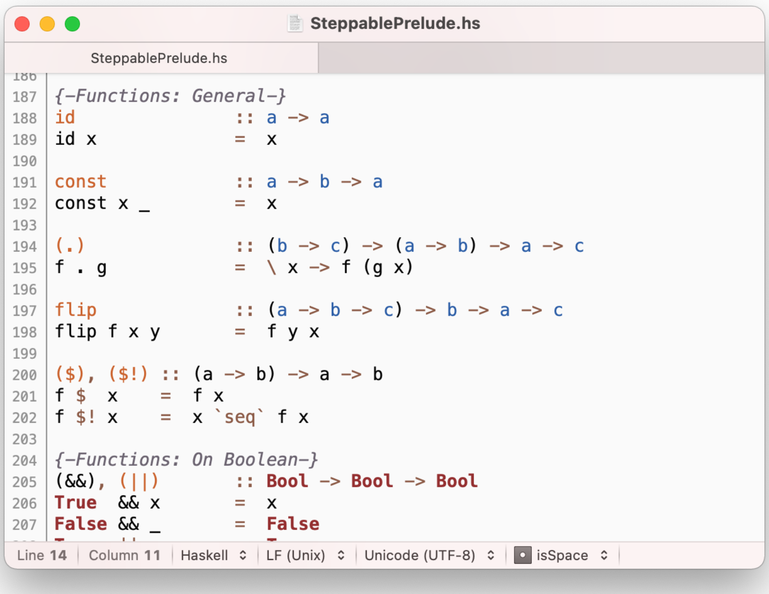
6.13.1 Introduction

The stepper should be easily accessible with a low entry barrier for beginners of the Haskell language. To allow users to quickly experiment with their own expressions, we added built-in support for the Standard Haskell Prelude from the Haskell 2010 Report [24].

Adding the prelude means that users do not have to re-define existing definitions from the prelude to step then. They can just work with the preexisting types, type classes and functions from the prelude.

Each Haskell implementation uses its own definition of the prelude. We took the original prelude definition because its main goal is code clarity instead of speed, which is perfect for our use case. At some points, we tweaked the prelude a little when we thought it would be helpful for the user.

While the prelude makes things easier for the users, they can still define custom types, type classes and functions.



```
{-Functions: General-}
id                  :: a -> a
id x                =  x

const               :: a -> b -> a
const x _           =  x

(.)                 :: (b -> c) -> (a -> b) -> a -> c
f . g               =  \ x -> f (g x)

flip                :: (a -> b -> c) -> b -> a -> c
flip f x y          =  f y x

($), ($!) :: (a -> b) -> a -> b
f $  x     =  f x
f $! x     =  x `seq` f x

{-Functions: On Boolean-}
(&&), (||)          :: Bool -> Bool -> Bool
True  && x          =  x
False && _          =  False
```

Line 14 | Column 11 | Haskell | LF (Unix) | Unicode (UTF-8) | isSpace

Figure 25: The steppable prelude is a separate file that is integrated into the stepper

6.13.2 Types

Besides the literal types and lists which were already supported by the stepper from the beginning, the prelude defines some more types: `Maybe`, `Either` and `Ordering`. The other types are not defined in the prelude itself but in the stepper backend. This is because not every type definition can be expressed in pure Haskell (for example tuples, lists and literal types).

> Declarations for special types such as Integer [...] cannot be given in Haskell.“
>
> (From the Haskell Report 2010 [24])

6.13.3 Type Classes

Type classes from the prelude are supported too, namely `Bounded`, `Enum`, `Eq`, `Floating`, `Fractional`, `Functor`, `Integral`, `Monad`, `Num`, `Ord`, `RealFloat` and `RealFrac`.

To integrate the type classes, we had to make an important architectural decision:

- **Option A**: Define and implement the type classes and type class instances inside the `SteppablePrelude.hs` file.
- **Option B**: Add support for the type class functions in the backend.

Option A would allow to step through functions defined in a type class. However, this option was not always possible to choose for two reasons:

- The Haskell prelude does not contain everything in plain Haskell. Many of the basic functions from type classes cannot be represented in Haskell code itself. For example, it is impossible to implement `Eq` for the `Int` type in pure Haskell. This is why expressions like `1 + 2` are evaluated in the backend.
- When we re-define well-known type classes in the `SteppablePrelude.hs` file (such as `Eq`), it would mean that those type classes would not be automatically derivable from anymore. From a user perspective, this would be bad. When users define custom types, they would have to manually implement the type classes. It would not be possible (without additional libraries and flags) tools to, for example, just write `deriving (Eq)` [25].

This is why we chose option B and added support for the type classes in the backend. This comes with the downside that functions defined in the backend are just evaluated instead of stepped. However, this is not a big deal as most of those functions are „unsteppable“ anyway: The prelude type classes mostly contain unsteppable functions such as `sin`, `+`, `toInteger` and so on.

The type classes `Read` and `Show` are not supported. This is because types such as `ReadS` or `ShowS` have no built-in equivalent representation in Haskell Core, and rebuilding such a type in

Core would be quite difficult. However, if users wish to work with type classes not defined in the prelude, they can simply provide custom type classes.

6.13.4 Functions

The prelude contains many helpful functions: Functions to work with types like `Maybe`/`Either`, functions to work with lists, operators, mathematical functions and much more. Those functions can now be used without any additional effort by the users. Users can also define custom functions.

6.13.5 Work with the Steppable Prelude from a User's Perspective

To work with the steppable prelude, the Module `SteppablePrelude` has to be imported. Furthermore, the automatic import of the standard prelude must be prevented so that there are no naming conflicts. A simple input file for the stepper which makes use of the `SteppablePrelude` could look like this [26]:

```haskell
{-# OPTIONS -fno-implicit-prelude #-}

module Source6 where
import SteppablePrelude

result = reverse [1, 2, 3]
```

Listing 78: Example stepper input file that uses the steppable prelude

A full guide on how to use the Steppable Prelude is included in the product documentation: `http://haskell-substitution-stepper.pages.gitlab.ost.ch/product-docu/guide/3_steppable_prelude.html`

6.13.6 Summary: How much of the Prelude is supported?

- **Operators**: everything (24/24)
- **Types**: everything except `IO` (13/14)
- **Type Classes / Type Class Instances**: everything except `Read` and `Show` (13/15)
- **Functions**: everything (93/93)

6.14 Resolution of Function Names

6.14.1 Overview: Six Types of Functions

There are six different kinds of functions that are differentiated in the stepper:

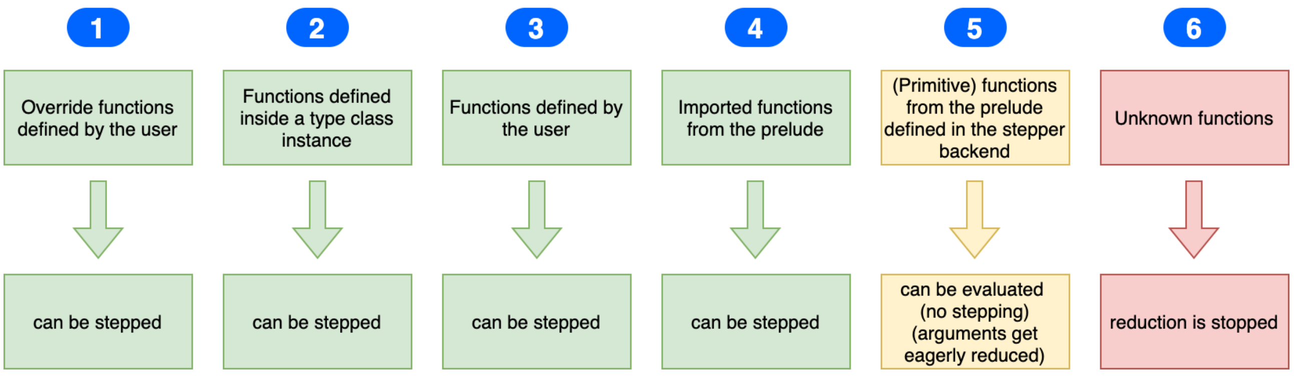

1
Override functions defined by the user
can be stepped
2
Functions defined inside a type class instance
can be stepped
3
Functions defined by the user
can be stepped
4
Imported functions from the prelude
can be stepped
5
(Primitive) functions from the prelude defined in the stepper backend
can be evaluated (no stepping) (arguments get eagerly reduced)
6
Unknown functions
reduction is stopped

Figure 26: The six different types of functions

When the stepper tries to find a function binding, it searches in the order indicated in the figure.

6.14.2 Override Functions defined by the User (1)

Every so often, it is useful to use different function definitions instead of the ones that are predefined in the prelude. That is why we implemented a way for the user to override function definitions from the prelude. To override a function, users just have to define a new function. This new function has the same name as the function that should be overwritten, but using the prefix `override’`. This can be useful, for example, if the original function is optimized for execution speed instead of readability.

Here is an example:

The reverse function of the prelude is defined in a way that is difficult to understand:

```
reverse :: [a] -> [a]
reverse =  foldl (flip (:)) []
```

Listing 79: Complicated reverse function from the prelude

For teaching purpose, it might be useful to replace the function with a slower but easier to read version:

```
override'reverse :: [a] -> [a]
override'reverse [] = []
override'reverse (x:xs) = (override'reverse xs) ++ [x]
```

Listing 80: Slower but easier reverse function

The overwritten `reverse` function is now automatically used everywhere instead of the original function.

6.14.3 Functions defined inside a Type Class instance (2)

In Haskell Core, application of functions defined inside a type class look like this:

```
customFunction @Int $fCustomTypeClassInt (I# 1#) (I# 1#)
```

Listing 81: Application of a function defined inside a type class

The parameter `$fCustomTypeClassInt` is a dictionary that contains implementations for every function required by the `CustomTypeClass`. In this example, the stepper would try to resolve `$fCustomTypeClassInt` and find the implementation for `customFunction`.

6.14.4 Functions defined by the User (3)

Functions defined by the user can be stepped because the stepper knows the body / definition of the function itself. Here is an example that shows how the stepping of such a user-defined function (`addOne`) looks like conceptually:

```
addOne 1
{-replace addOne with definition-}
(\x -> (+) x 1) 1
{-lamda reduction-}
(+) 1 1
{-apply (+)-}
2
```

Listing 82: Example stepping on a user-defined function

6.14.5 Imported Functions from the Prelude (4)

As described in Chapter 6.13, there is a custom prelude file containing many definitions. This custom prelude file is converted to Haskell Core at runtime (just like the input file from the user).

Stepping a prelude function looks exactly like stepping a function defined by the user itself.

6.14.6 Unsteppable Functions from the Prelude (5)

There are some functions from type classes that cannot be stepped. The stepper can still evaluate those functions as long as both the argument- and result-types are supported:

- An argument-type is supported if there is a converter function that converts an `Expr` into the expected argument type. For example, if a function expects an argument to have the type `Bool`, there has to be a converter function like `exprToBool :: Expr Var => Bool`. Note that such a conversion can potentially lead to an error. For example, this might be the case if the expression does not contain a representation of a `Bool` instance. However, such errors are not expected to happen because such type errors are already checked by GHC before the stepper logic itself is executed.
- A result-type is supported if there is a converter function that converts the result back into an `Expr`. For example, if a function returns a type `Bool`, there has to be a converter function like `boolToExpr :: Bool => Expr Var`.

This downside with this kind of function is that partial function application is not possible. Furthermore, every argument passed to the function has to be eagerly reduced into a literal. This is strict behavior and thus not the same evaluation order that GHC would choose. This is not ideal but still better than stopping the reduction altogether. (See Chapter 6.8 for more details about the collection and reduction of application arguments)

Here is a code snipped that shows how such unparsable / unsteppable functions are evaluated. Note how converter functions are used:

```haskell
evaluateUnsteppableFunctionWithArguments :: String -> [Expr Var] -> Expr
↪  Var
evaluateUnsteppableFunctionWithArguments "+" [x, y] = (+) x y
evaluateUnsteppableFunctionWithArguments ">" [x, y] = boolToCoreExpression
↪  ((>) x y)
evaluateUnsteppableFunctionWithArguments "negate" [Lit (LitNumber _ x)] =
↪  integerToCoreExpression (negate x)
{-...-}
```

Listing 83: Excerpt of the evaluateUnsteppableFunctionWithArguments function

Here is the conceptual output of the stepper when it encounters such an unparsable function:

```
(magicalUnsteppableFunction (((+) 1) 2)) ((+ 3) 4)
{-strict/eager evaluate the first argument-}
(magicalUnsteppableFunction 3) ((+ 3) 4)
{-strict/eager evaluate the second argument-}
(magicalUnsteppableFunction 3) 7
{-evaluate magicalUnsteppableFunction (showing substepps is not supported
↪  for this function)-}
42
```

Listing 84: Example evaluation on an unsteppable function

6.14.7 Unknown Functions (6)

Unknown functions that are not defined by the user, and that are not supported by the stepper per default, cause the stepper to stop.

Here is what the output looks like (conceptually) for such an unsupported function:

```
unknownFunction 1
{-unknownFunction is not supported-}
```

Listing 85: Stepper output for an unsupported expression

6.15 Support for Infinite Lists

The stepper supports infinite lists like `[1..]`. Infinite lists can be used as function parameters, for example `take 3 [1..]`.

To support infinite lists, it was necessary to implement a custom version of the `Enum` typeclass. The original definition from the prelude would have resulted in an infinite loop when the substeps are printed.

6.16 Detection of Infinite Loops

Of course, infinite loops can still occur if the final result of a reduction is an infinite data structure. The stepper detects such infinite loops and stops the output before an error occurs. Here is an example where further reduction to normal form would lead to an infinite loop. In this case, the stepper stops the reduction as soon as the expression is in head normal form (but not in normal form):

```
: @Integer
  0
  (enumFrom
     @Integer $fEnumInteger (succ @Integer $fEnumInteger 0))

{-reduction is complete-}
```

Listing 86: Detection of infinite loops

6.17 Haskell-Like Pretty Printing

6.17.1 Pretty Printer

Haskell Core code is not completely difficult to understand for someone who knows the Haskell language. But sometimes, it is still exhausting to read because the interesting part of an expression is cluttered with additional information such as types.

To make the stepper more beginner-friendly, we added an optional "Haskell-like" output. If this option is used, the stepper prints out Core expressions in a way that it looks like original Haskell Code (and usually is indeed valid Haskell Code). The pretty printer transforms the Core expression in several ways to make it easier to read:

- Remove type information
- Use infix operators
- Convert nested lamdas into lamdas with multiple parameters
- Print lists like `[1, 2, 3]` instead of `:  1 (:  2 (:  3 []))`
- Visualize tuples and case-patterns like Haskell

The users can choose in which syntax they want to see the output. The next subchapters show comparisons of the output.

The Haskell-Like Pretty Printer for Core is a separate module which could also be useful for other projects.

6.17.2 Example Expression

Original User Input This is what the user enters:

```
mapComparison = map negate [1, 2, 3] < [1, 2, 3]
```

Listing 87: Pretty print example 1: Original User Input

GHC's Pretty Printer for Core This is the same expression, printed in Core:

```
mapComparison =
        < @[Integer]
          ($fOrd[] @Integer $fOrdInteger)
          (map
             @Integer
             @Integer
             (negate @Integer $fNumInteger)
             (: @Integer 1 (: @Integer 2 (: @Integer 3 ([] @Integer)))))
                 (: @Integer 1 (: @Integer 2 (: @Integer 3 ([]
                 ↪  @Integer))))
```

Listing 88: Pretty print example 1: Core output

Our Haskell-Like Pretty Printer This is the same Core expression from above, printed by the Haskell-Like Pretty Printer:

```
mapComparison = (map (negate) [1, 2, 3]) < [1, 2, 3]
```

Listing 89: Pretty print example 1: Haskell-like pretty printed Core output

Remarks The output of the Haskell-Like Pretty Printer looks almost like the user's input, apart from some explicit brackets.

6.17.3 Example Expression 2

Original User Input This is what the user enters:

```haskell
monadExample = do
  a <- [1, 2, 3]
  b <- [3, 2, 1]
  return (a * b)
```

Listing 90: Pretty print example 2: Original User Input

GHC's Pretty Printer for Core This is the same expression, printed in Core:

```
monadExample =
  >>=
    @[]
    $fMonad[]
    @Int
    @Int
    (: @Int (I# 1#) (: @Int (I# 2#) (: @Int (I# 3#) ([] @Int))))
    (\ (a :: Int) ->
       >>=
         @[]
         $fMonad[]
         @Int
         @Int
         (: @Int (I# 3#) (: @Int (I# 2#) (: @Int (I# 1#) ([] @Int))))
         (\ (b :: Int) -> return @[] $fMonad[] @Int (* @Int $fNumInt a
         ↪  b)))
```

Listing 91: Pretty print example 2: Core output

Our Haskell-Like Pretty Printer This is the same Core expression from above, printed by the Haskell-Like Pretty Printer:

```haskell
monadExample = [1, 2, 3] >>= (\a -> [3, 2, 1] >>= (\b -> return (a * b)))
```

Listing 92: Pretty print example 2: Haskell-like pretty printed Core output

Remarks Haskell Core has no monadic do-syntax. Instead, the operator >>= is used. The Haskell-Like Pretty Printer output is valid Haskell as well.

6.18 Configurable Verbosity

6.18.1 Full Verbosity vs. Minimal Verbosity

The full output of a step by step reduction can become quite large. Sometimes even simple input expressions result in hundreds of pages in output and a large amount of sub-steps. Especially recursive statements, which are widely used in Haskell, increase the length of the output.

This is the reasons we made the verbosity of the output configurable. In the configuration of the backend, it can be decided for every possible reduction category if the result should be printed or not. To make the tool accessible, the user can choose between two verbosity levels: all sub-steps or only interesting sub-steps. When the user chooses "only interesting sub-steps", only the following reductions-types are shown:

- Evaluation of unsteppable functions
- Pattern Matching

Practice has shown that this minimal verbosity configuration leads to good and understandable results.

6.18.2 Example of minimal verbosity

The full reduction for the expression `maximum [1, 2, 3]` contains 26 sub-steps. If the minimal verbosity is turned on, only 7 “interesting“ sub-steps are shown:

```
maximum [1, 2, 3]

{- skipping 3 substeps -}

{- Replace with matching pattern -}
foldl1 (max) [1, 2, 3]

{- skipping 4 substeps -}

{- Replace with matching pattern -}
foldl (max) 1 [2, 3]

{- skipping 4 substeps -}

{- Replace with matching pattern -}
foldl (max) (max 1 2) [3]

{- skipping 4 substeps -}

{- Replace with matching pattern -}
foldl (max) (max (max 1 2) 3) []

{- skipping 4 substeps -}

{- Replace with matching pattern -}
max (max 1 2) 3

{- (Strict) reduce application argument -> Evaluate unsteppable
↪  function/operator max -}
max 2 3

{- Evaluate unsteppable function/operator max -}
3

{- reduction completed successfully -}
```

Listing 93: Example of minimal verbosity output

6.19 Notes for the Maintenance of this Project

6.19.1 Project Maintenance

There might be a continuation work based on this project, so we want to be transparent about the pitfalls we had to deal with and the workarounds we chose. This section should help developers in the future to work with the substitution stepper and to maintain it. We consulted experts of GHC Core but there does not seem to be simple solutions to replace those workarounds chosen:

6.19.2 Workaround: Creation of Boolean Data Constructors

To support some unsteppable functions from the prelude, it was sometimes necessary to convert basic Haskell types into their corresponding Core representation and the other way around. We were lucky that GHC contains many helper functions for the creation of literal expressions, such as `mkLitString` or `mkLitInt64`.

However, the `Bool` type does not have its own custom representation in Haskell Core (such as `Integer`, `Char`, and others). Instead, `True` and `False` are represented as type constructors.

The implementation of our function `boolToCoreExpression :: Bool -> CoreExpr` looks odd. It allows us to work with Bool values, but we felt like there must be a more elegant way to create custom type constructors in Core.

6.19.3 Workaround: Sporadic Usage of „show" to Extract some Information from Core Instances

In some cases, we need to extract specific information from a Core expression but did not find a function in GHC that helps to extract the information. In such cases, we used the `show` function for Core expressions and extracted the required information on a `String` level. This happens in these cases:

- To find out if a function refers to a class dictionary (such as `$EqInteger`), the stepper analyzes the name of the function. This is not the most elegant way, as it might lead to errors if the user defines functions where the name resembles a type class instances. Fortunately, this is unlikely.
- When the stepper looks up a specific binding from the list of bindings, it compares the binding name and the expression type to choose the right binding. The comparison happens on the `String`-representation of these types. This works but is not elegant. Unfortunately, the type `Type` does not derive from the `Eq` type class.
- To find out if a type is a tuple type, the stepper uses `String`-comparison. A tuple type

looks different depending on how large a tuple is. Valid tuple types are `(,)`, `(,,)`, `(,,,)` and so on. The stepper checks if the type starts with a `(` and ends with a `)` to decide if a type describes a tuple. Again, this works but is not elegant.

6.19.4 Workaround: Extracting Functions out of Type Class Dictionaries

The following example snippet shows the generated type dictionary `$fEqDirection` which contains functions from the `Eq` type class for the custom type `Direction`. It includes references to the implementations of `==` and `/=` for this specific type.

```
C:Eq @(Direction a) ($c== @a $dEq_a3FW) ($c/= @a $dEq_a3FW)
```

Listing 94: Class dictionary in Core, containing references to the operators == and /=

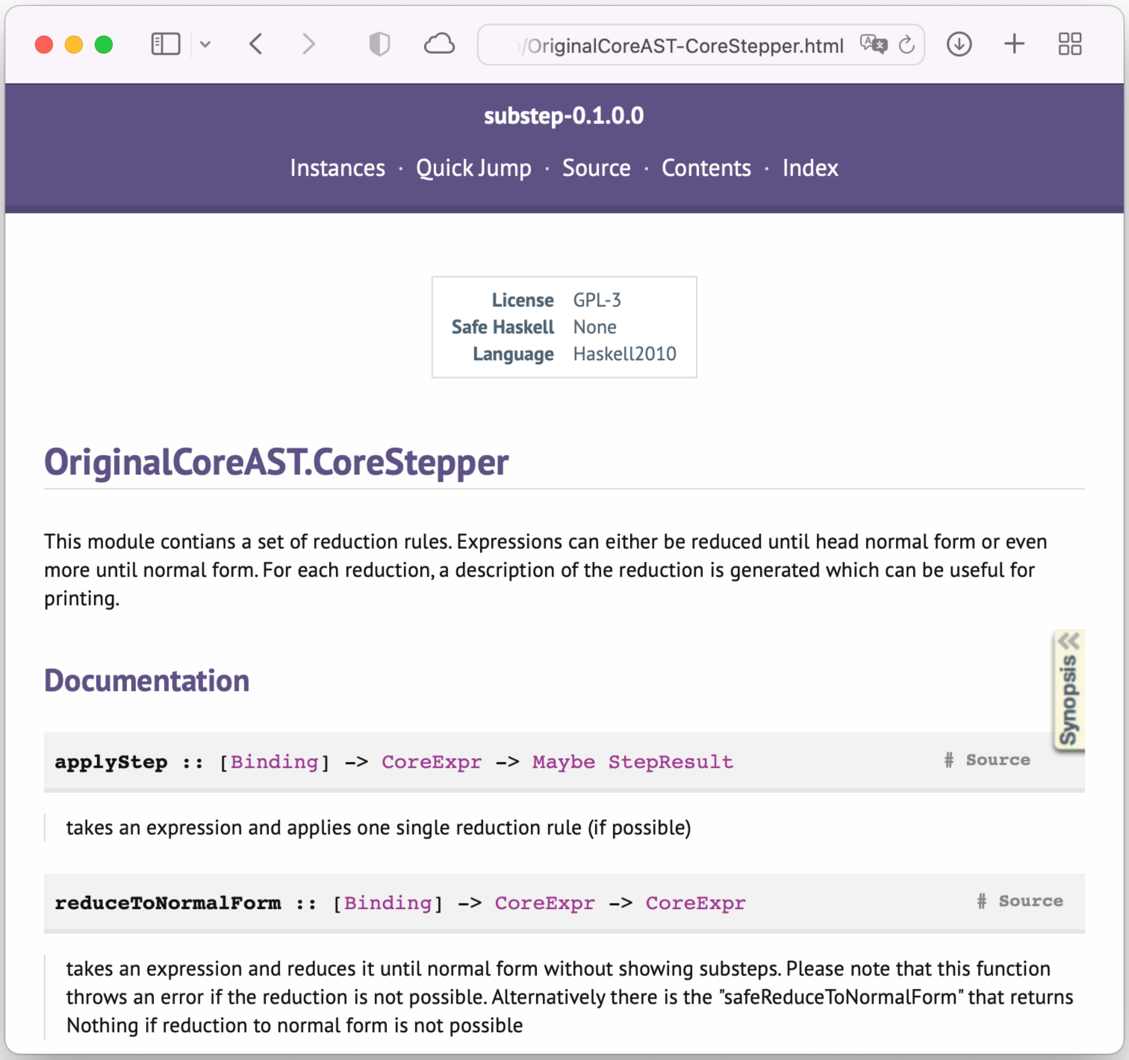
When the stepper needs to find the implementation for an operator or function (such as `==`) it compares the function name to the suffixes of the (automatically named) function from the class dictionary. In this example, the stepper would find the right function, which is `$c==`. This worked for every example we attempted. However, there is no guarantee that the generated function names follow this pattern, which could potentially result in a bug. Unfortunately, we haven't discovered a better way to solve this imperfection.

7 Testing and Documentation

7.1 Haddock Documentation

We documented the source code using Haddock Comments as well as inline commends where we thought it would be helpful for developers.

The Haddock Documentation can be downloaded from the product documentation GitLab Page: `http://haskell-substitution-stepper.pages.gitlab.ost.ch/product-docu/index.html`


/OriginalCoreAST-CoreStepper.html
substep-0.1.0.0
Instances · Quick Jump · Source · Contents · Index
Synopsis

License	GPL-3
Safe Haskell	None
Language	Haskell2010

OriginalCoreAST.CoreStepper

This module contians a set of reduction rules. Expressions can either be reduced until head normal form or even more until normal form. For each reduction, a description of the reduction is generated which can be useful for printing.

Documentation

```
applyStep :: [Binding] -> CoreExpr -> Maybe StepResult                # Source
```

takes an expression and applies one single reduction rule (if possible)

```
reduceToNormalForm :: [Binding] -> CoreExpr -> CoreExpr               # Source
```

takes an expression and reduces it until normal form without showing substeps. Please note that this function throws an error if the reduction is not possible. Alternatively there is the "safeReduceToNormalForm" that returns Nothing if reduction to normal form is not possible

Figure 27: Haddock documentation

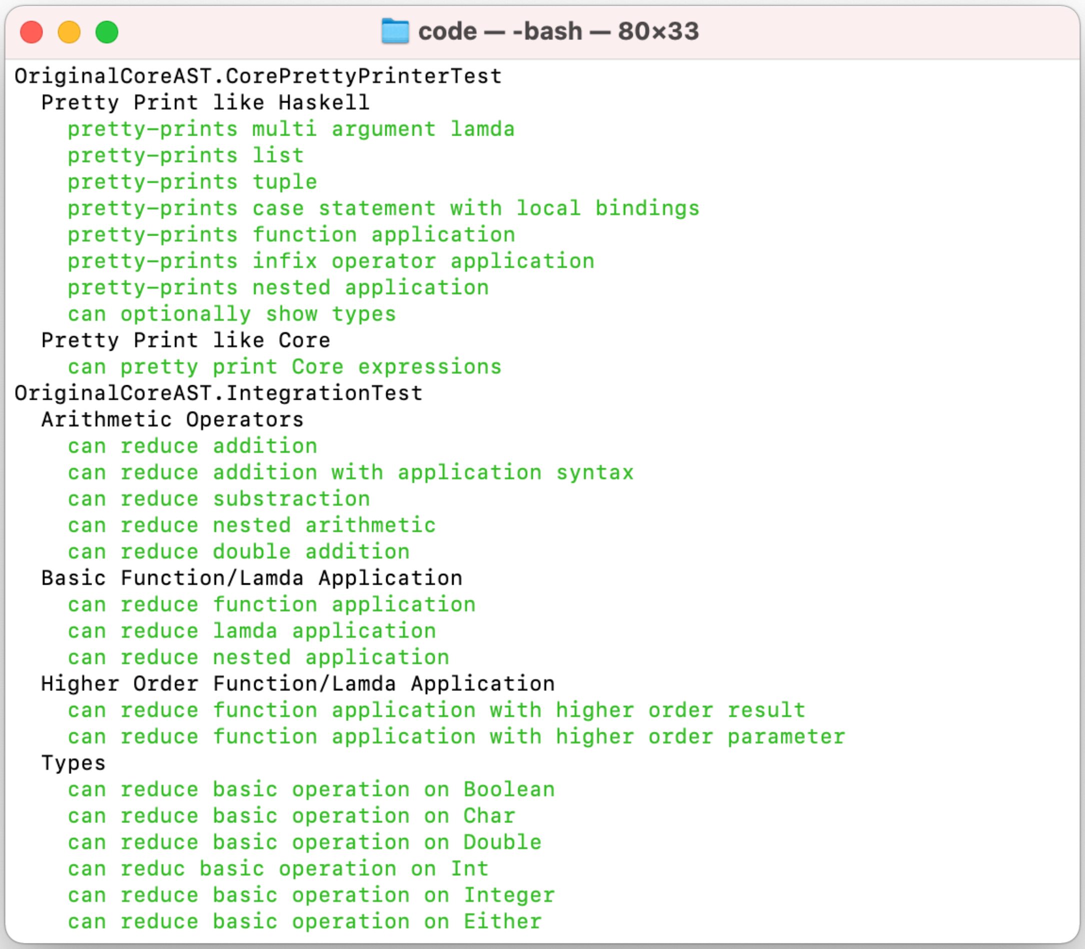
7.2 Test Suite

The testing framework Hspec and its integration with QuickCheck was used to write automatic unit and integration tests. The tests can be started with the following command:

```
stack --no-terminal test
```

Listing 95: Command to execute tests in the console

This is what part of the output looks like:



```
OriginalCoreAST.CorePrettyPrinterTest
  Pretty Print like Haskell
    pretty-prints multi argument lamda
    pretty-prints list
    pretty-prints tuple
    pretty-prints case statement with local bindings
    pretty-prints function application
    pretty-prints infix operator application
    pretty-prints nested application
    can optionally show types
  Pretty Print like Core
    can pretty print Core expressions
OriginalCoreAST.IntegrationTest
  Arithmetic Operators
    can reduce addition
    can reduce addition with application syntax
    can reduce substraction
    can reduce nested arithmetic
    can reduce double addition
  Basic Function/Lamda Application
    can reduce function application
    can reduce lamda application
    can reduce nested application
  Higher Order Function/Lamda Application
    can reduce function application with higher order result
    can reduce function application with higher order parameter
  Types
    can reduce basic operation on Boolean
    can reduce basic operation on Char
    can reduce basic operation on Double
    can reduc basic operation on Int
    can reduce basic operation on Integer
    can reduce basic operation on Either
```

Figure 28: Test output

The tests are also executed in the CI for every push. If there are failing tests, the whole pipeline fails.

7.3 Integration Tests

- **Total Amount of Integration tests**: 114 (a list of all the tests can be found in the appendix)

Each integration test defines an input expression in pure Haskell, as well as an expected output expression. Here is an example:

```haskell
lamdaApplicationInput = (\x y -> x+y) 1 2
lamdaApplicationExpectedOutput = 3
```

Listing 96: Input and expected output for an example integration test

Using the GHC library, both expressions are converted to Core during the test. Next, the input expression is reduced with the stepper until it is in head normal form. Finally, the equality of the reduced input expression and the expected output expression is checked.

The call to the GHC library is not pure, as it uses the `IO` monad. Therefore, it was a good idea to use the `before` hook from Hspec to pass all the data needed into the test instead of calling the GHC library from inside the tests. Like this, we did not have to work with monadic code inside the tests. All integration tests receive two variables to work with:

- a function `bindingFinder` which is used to get the Core expression for a given name
- a function `coreBindings` containing all the Core expressions

This is what a typical integration tests looks like:

```haskell
spec :: Spec
spec = before getBindingFinderWithCoreBindings $ do
  describe "Basic Function/Lamda Application" $ do
    it "can reduce function application" $ \(bindingFinder, coreBindings)
     ↪  -> do
      let input = bindingFinder "lamdaApplicationInput"
      let expectedOutput = bindingFinder "lamdaApplicationExpectedOutput"
      let actualOutput = reduceToHeadNormalForm coreBindings input

      {-expectation-}
      actualOutput `shouldBe` expectedOutput
```

Listing 97: A typical integration test

Note that the `shouldBe` function from Hspec can directly be used because we implemented the `Eq` type for `Expr b`. The equality comparison only works if both expressions are `Literals`.

7.4 Unit Tests

- **Amount of Unit tests**: 19 (a list of all the tests can be found in the appendix)

The integration tests check that expressions are successfully reduced from the initial form into normal form. The integration tests compare the reduced expression in normal form with the expected result to decide whether the reduction was successful or not. In contrast, the “Unit tests“ check if the individual reduction steps are applied correctly.

Technically speaking, those tests are integration tests as well. They trigger the GHC pipeline to obtain the input expression. Still, we refer to them as Unit tests because the intention is to test the function `applyStep` and not the GHC pipeline itself. This approach was chosen because it is almost impossible to create a valid and useful Core expression from scratch without using the GHC.

The function `applyStep` takes an expression as an argument and returns a reduced expression and a description of the reduction rule that was selected. The unit tests check if the right reduction rule was chosen for the given input expression and if the reduced expression has the expected format.

There are also separate unit tests which check the output format of the Haskell-Like Pretty Printer.

7.5 End to End tests

End to End tests were done manually. A Haskell file was defined that contains numerous Haskell expressions. Each expression uses a different aspect or syntactic feature of the Haskell language. We printed the reduction output for each expression and made sure that all the sub steps are correct.

7.6 Investigating Bugs

Sporadically, it was helpful for debugging to add tracing statements to the code when a specific problem had to be investigated.

Here is how an expression that makes use of tracing looks:

```haskell
tryFindBindingForString key [] = trace "no binding found" Nothing
```

Listing 98: Usage of tracing

7.7 Testing Conclusion

Automatic testing helped us to discover multiple bugs. For example, there was a silly bug regarding the pattern matching: The patterns in Core are not necessarily in the same order as the patterns in original Haskell. This led to the bug that sometimes during the stepping, the default-pattern (`_`) was used even though there were other matching patterns.

The tests not only helped to discover bugs, but they also helped to get an overview of the supported and unsupported syntax at any given point in time. We implemented not only tests for existing functionality, but also for the missing functionality using the `pendingWith` function from Hspec. Each pending test can be interpreted as a to-do item.

8 Result

8.1 Final product

The final product implements reduction rules for all the Core Constructors (except for `Tick` which is rarely used) which means that most expressions should be supported. As stated in the task description, users can define a custom context with functions, types, and type classes and choose an expression to step. If the output is printed with the Haskell-Like Pretty Printer, the result is a valid Haskell expression in most of the cases. For every substitution step taken, the chosen reduction rule is printed out as a comment. Those comments can also be suppressed if the user wishes so. Depending on the chosen verbosity, some sub-steps are skipped. The default verbosity is chosen in a way that the output looks like the reductions which are known from textbooks. In the backend, the verbosity can be fine-tuned for every reduction rule individually. The GHC library was used whenever possible: Mainly to convert the Haskell Input into Haskell Core, but also to extract information and manipulate Core expressions.

If there will be a continuation-work, it would make sense to connect the stepper backend to a graphical user interface. The backend is created in a manner that allows for an easy exchange of the user interface. This would make the tool more accessible for beginner users and would allow adding additional features – for example, to highlight subexpressions that change or to let the user select which part of an expression should be reduced next.

There is a user manual on the product documentation GitLab Page: `http://haskell-substitution-stepper.pages.gitlab.ost.ch/product-docu/index.html`

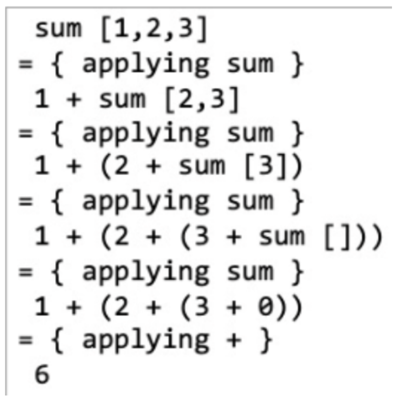
8.2 Demonstration

8.2.1 Examples

The task descriptions contained some example reductions from textbooks. In the next chapters, we compare those reductions with the output of the stepper for the same expression.

8.2.2 Example 1: sum (Core-Like Pretty Printing, Low Verbosity)

Here is the reduction printed in the textbook:



```
 sum [1,2,3]
= { applying sum }
 1 + sum [2,3]
= { applying sum }
 1 + (2 + sum [3])
= { applying sum }
 1 + (2 + (3 + sum []))
= { applying sum }
 1 + (2 + (3 + 0))
= { applying + }
 6
```

Figure 29: Textbook example 1

Here is the output of the stepper:

```
sum [1, 2, 3]

{- skipping 5 substeps -}

{- Replace with matching pattern -}
foldl ((+)) ((fromInteger 0) + 1) [2, 3]

{- skipping 4 substeps -}

{- Replace with matching pattern -}
foldl ((+)) (((fromInteger 0) + 1) + 2) [3]

{- skipping 4 substeps -}

{- Replace with matching pattern -}
foldl ((+)) ((((fromInteger 0) + 1) + 2) + 3) []

{- skipping 4 substeps -}

{- Replace with matching pattern -}
(((fromInteger 0) + 1) + 2) + 3

{- (Strict) reduce application argument -> (Strict) reduce application
↪  argument -> (Strict) reduce application argument -> Evaluate
↪  unsteppable function/operator fromInteger -}
((0 + 1) + 2) + 3

{- (Strict) reduce application argument -> (Strict) reduce application
↪  argument -> Evaluate unsteppable function/operator + -}
(1 + 2) + 3

{- (Strict) reduce application argument -> Evaluate unsteppable
↪  function/operator + -}
3 + 3

{- Evaluate unsteppable function/operator + -}
6

{- reduction completed successfully -}
```

Listing 99: Reduction of sum

As can be seen, the implementation of `sum` is different from the version in the textbook. However, we can define a function custom `sumList` that works like the example in the textbook:

```
sumList :: (Num a) => [a] -> a
sumList [] = 0
sumList (x:xs) = x + sumList xs
```

Listing 100: Definition of sumList

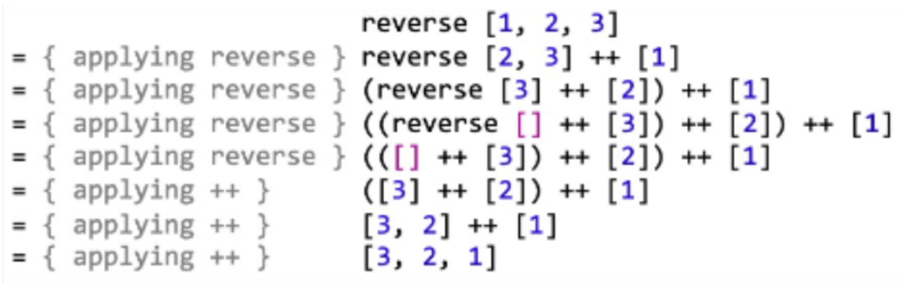
Now, the stepper output almost exactly looks like the example in the textbook:

```
sumList [1, 2, 3]

{- skipping 2 substeps -}

{- Replace with matching pattern -}
1 + (sumList [2, 3])

{- skipping 2 substeps -}

{- (Strict) reduce application argument -> Replace with matching pattern -}
1 + (2 + (sumList [3]))

{- skipping 2 substeps -}

{- (Strict) reduce application argument -> (Strict) reduce application argument -> Replace with matching pattern -}
1 + (2 + (3 + (sumList [])))

{- skipping 2 substeps -}

{- (Strict) reduce application argument -> (Strict) reduce application argument -> (Strict) reduce application argument -> Replace with matching pattern -}
1 + (2 + (3 + (fromInteger 0)))

{- (Strict) reduce application argument -> (Strict) reduce application argument -> (Strict) reduce application argument -> Evaluate unsteppable function/operator fromInteger -}
1 + (2 + (3 + 0))

{- (Strict) reduce application argument -> (Strict) reduce application argument -> Evaluate unsteppable function/operator + -}
1 + (2 + 3)

{- (Strict) reduce application argument -> Evaluate unsteppable function/operator + -}
1 + 5

{- Evaluate unsteppable function/operator + -}
6

{- reduction completed successfully -}
```

Listing 101: Reduction of sumList

8.2.3 Example 2: reverse (Core-Like Pretty Printing, Low Verbosity)

Here is the reduction printed in the textbook:



```
                        reverse [1, 2, 3]
= { applying reverse } reverse [2, 3] ++ [1]
= { applying reverse } (reverse [3] ++ [2]) ++ [1]
= { applying reverse } ((reverse [] ++ [3]) ++ [2]) ++ [1]
= { applying reverse } (([] ++ [3]) ++ [2]) ++ [1]
= { applying ++ }      ([3] ++ [2]) ++ [1]
= { applying ++ }      [3, 2] ++ [1]
= { applying ++ }      [3, 2, 1]
```

Figure 30: Textbook example 2

Again, the behavior of the `reverse` function from the prelude is different from the function from the textbook.

For a better comparison, we define a custom `reverseList` function:

```
reverseList :: [a] -> [a]
reverseList [] = []
reverseList (a : bc) = reverseList bc ++ [a]
```

Listing 102: Definition of reverseList

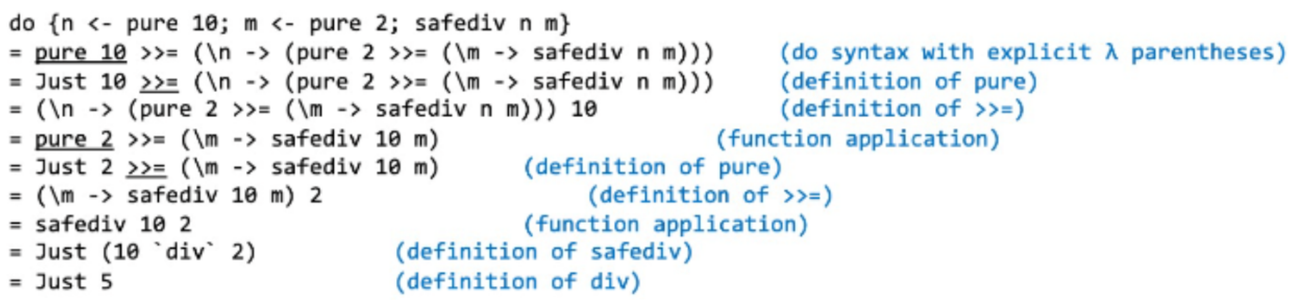
The resulting reduction looks very similar to the textbook version and is slightly less verbose in the “low verbosity“ configuration:

```
reverseList [1, 2, 3]

{- skipping 2 substeps -}

{- Replace with matching pattern -}
(reverseList [2, 3]) ++ [1]

{- skipping 20 substeps -}

{- Replace with matching pattern -}
3 : (([] ++ [2]) ++ [1])

{- reduction complete - reduce constructor arguments for better
↪  visualization -}
[3, 2, 1]

{- reduction completed successfully -}
```

Listing 103: Reduction of reverseList

8.2.4 Example 3: safediv (Core-Like Pretty Printing, Low Verbosity)

Here is the reduction printed in the textbook:



```
do {n <- pure 10; m <- pure 2; safediv n m}
= pure 10 >>= (\n -> (pure 2 >>= (\m -> safediv n m)))     (do syntax with explicit λ parentheses)
= Just 10 >>= (\n -> (pure 2 >>= (\m -> safediv n m)))     (definition of pure)
= (\n -> (pure 2 >>= (\m -> safediv n m))) 10              (definition of >>=)
= pure 2 >>= (\m -> safediv 10 m)                     (function application)
= Just 2 >>= (\m -> safediv 10 m)        (definition of pure)
= (\m -> safediv 10 m) 2                      (definition of >>=)
= safediv 10 2                           (function application)
= Just (10 `div` 2)             (definition of safediv)
= Just 5                        (definition of div)
```

Figure 31: Textbook example 3

This is the definition of the `safeDiv` function:

```haskell
safeDiv :: Integer -> Integer -> Maybe Integer
safeDiv x 0 = Nothing
safeDiv x y = Just (x `div` y)
```

Listing 104: Definition of safeDiv

The output of the stepper is identical, although not every step from the textbook is shown:

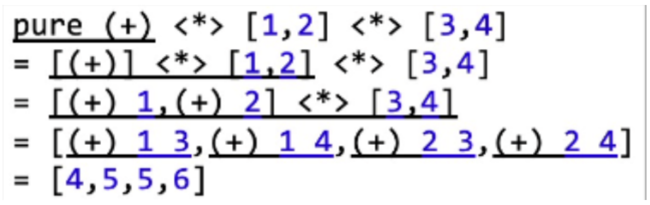
```
(pure 10) >>= (\n -> (pure 2) >>= (\m -> safeDiv n m))

{- skipping 5 substeps -}

{- Replace with matching pattern -}
(\n -> (pure 2) >>= (\m -> safeDiv n m)) 10

{- skipping 6 substeps -}

{- Replace with matching pattern -}
(\m -> safeDiv 10 m) 2

{- skipping 5 substeps -}

{- Replace with matching pattern -}
Just (div 10 2)

{- reduction complete - reduce constructor arguments for better
↪  visualization -}
Just 5

{- reduction completed successfully -}
```

Listing 105: Reduction of safeDiv

Like in all the other examples, all the reduction steps can be shown using a more verbose configuration.

8.2.5 Example 4: applicative

Here is the reduction printed in the textbook:



```
pure (+) <*> [1,2] <*> [3,4]
= [(+)] <*> [1,2] <*> [3,4]
= [(+) 1,(+) 2] <*> [3,4]
= [(+) 1 3,(+) 1 4,(+) 2 3,(+) 2 4]
= [4,5,5,6]
```

Figure 32: Textbook example 4

This last expression cannot elegantly be stepped with the stepper. The reason is that the `Applicative` type class is not part of the standard prelude.

8.2.6 Examples of the four verbosity levels

All the examples listed here use the lowest verbosity level because higher verbosity levels tend to expand over many pages. In the appendix, there is an example of the same expression reduced in all 4 verbosity levels.

8.3 Validation of Functional Requirements

Requirement	A user can enter arbitrary Haskell Code into the stepper as text input (not a simplified Version of Haskell).
Result	Requirement accomplished
Requirement	A user can provide multiple files at once, input files can reference other files.
Result	Requirement not accomplished. Only one single input file (containing an arbitrary number of expression) can be provided as input.
Requirement	A user can configure the verbosity of the stepper output.
Result	Requirement accomplished. User can choose four different verbosity levels.
Requirement	The stepper explains each reduction step that was taken.
Result	Requirement accomplished.
Requirement	The stepper follows the rules of laziness and non-strictness.
Result	Requirement partially accomplished. Rules of laziness and non-strictness are followed. Exception: Arguments of unsteppable functions are eagerly reduced as described in Chapter 6.14.6.
Requirement	A user can work with custom functions, custom types and custom type classes.
Result	Requirement accomplished.
Requirement	A user can work with functions, types, and type classes from the Standard Haskell Prelude where it is feasible.
Result	Requirement accomplished. Support for the Prelude is implemented except for IO, Show and Read.

8.4 Validation of Non-Functional Requirements

Requirement	Performance: Generating and Printing the reduction steps for a simple expression like “reverse [1, 2, 3, 4]“ does not take any longer than 10 seconds on the development machines.
Result	Requirement accomplished. Compiling a file printing (multiple) reduction takes about two seconds.
Requirement	Security: The software does not use dynamic loading with “eval“ to implement the reduction rules.
Result	Requirement accomplished. There is a custom “evaluate“ function that does not use dynamic loading.
Requirement	Constraints: Reduction are implemented in the the Haskell programming language.
Result	Requirement accomplished.
Requirement	Installability: There is a one-command installer for the software.
Result	Requirement accomplished. The software can be installed with stack (also see installation guide in project documentation)
Requirement	Learnability: The tool can be used and understood by someone without having read the full user manual.
Result	Requirement accomplished. All options and commands are explained in the helper page of the console tool.
Requirement	Fault tolerance: Expressions that cannot be stepped do not result in a crash of the application.
Result	Requirement accomplished.
Requirement	Fault tolerance: Infinite loops during reduction do not result in a crash of the application, instead the reduction is aborted.
Result	Requirement accomplished.
Requirement	Maintainability: The code is structured in a modular way. The user interface can potentially be exchanged without having to change the backend.
Result	Requirement accomplished.
Requirement	Modifiability/Changeability: The backend can be individually configured. Not every configuration option has to be shown to the user.
Result	Requirement accomplished.

Requirement	Testability: The CI pipeline fails if there are failing automated tests.
Result	Requirement accomplished.

8.5 Code Metrics

Here is a statistic about the code written for this project:

Language	**files**	**blank**	**comment**	**code**
Haskell	33	884	505	3853
YAML	4	36	116	121
Markdown	2	23	0	40
———	———	———	———	———
SUM:	39	943	621	4014

Table 3: Code Metrics

9 Conclusion

The goal of this thesis was to implement a tool hat can be used to visualize the execution of a functional program.

To achieve this, we first compiled a comprehensive overview and analysis of the Haskell programming language, its features, as well as the Glasgow Haskell Compiler pipeline and existing tools with similar goals.

Due to this research and after discussions with various Haskell experts, we concluded that it is more advantageous to implement the substitution stepper on GHCs intermediate language Haskell Core than on the original Haskell AST.

We developed a command line tool that can successfully step through most Haskell programs and produces an output that is very similar to textbook examples. The stepper output is customizable so that the level of detail and amount of information provided is appropriate to the user's needs.

In comparison to similar, previously existing tools, our Haskell Substitution Stepper supports a larger part of Haskell and is more closely coupled with the Glasgow Haskell Compiler. It is also easier to customize and extend.

Future work could include the integration of the stepper into GHCi or an IDE-plugin, for example for VS Code. Further possibilities are the implementation of an interactive GUI or expanding the supported Haskell language features. It would be interesting to provide an online version of the stepper to make it even more accessible without installation.

10 Appendix

10.1 Declaration of other People's Work

- **Code Snippets**: The source of unchanged code snippets is stated as a comment in the source code.
- **Documentation template**: This document is based on the following template by "maknesium" which is published under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Germany License: `https://github.com/maknesium/latex-vorlage-diplom-bachelor-masterarbeiten`
- **GitLab Page Template**: The template chosen for the GitLab Page was created by former students.
- **Documentation correction**: Many thanks to AnneMarie O'Neill for proofreading this text!

10.2 Example of the four verbosity levels

Input Expression

This is the input expression:

```haskell
maximum [1, 2, 3]
```

Listing 106: maximum function

Verbosity Level 1: Haskell-Like Pretty Printing, Skipping Substeps

This is the output for verbosity level 1:

```
maximum [1, 2, 3]

{- skipping 3 substeps -}

{- Replace with matching pattern -}
foldl1 (max) [1, 2, 3]

{- skipping 4 substeps -}

{- Replace with matching pattern -}
foldl (max) 1 [2, 3]

{- skipping 4 substeps -}

{- Replace with matching pattern -}
foldl (max) (max 1 2) [3]

{- skipping 4 substeps -}

{- Replace with matching pattern -}
foldl (max) (max (max 1 2) 3) []

{- skipping 4 substeps -}

{- Replace with matching pattern -}
max (max 1 2) 3

{- (Strict) reduce application argument -> Evaluate unsteppable
function/operator max -}
max 2 3

{- Evaluate unsteppable function/operator max -}
3
```

Verbosity Level 2: Haskell-Like Pretty Printing, No Skipping

This is the output for verbosity level 2:

```
maximum [1, 2, 3]

{- Reduce function of application -> Replace 'maximum' with
definition -}
(\$dOrd_a84C ->
  \ds_d8Ej ->
    case ds_d8Ej of {
      [] -> (error "Prelude.maximum: empty list");
      _ -> (foldl1 (max) ds_d8Ej)
    }
    where
      $dIP_a84F =
        (pushCallStack ("error", SrcLoc "main" "SteppablePrelude"
"src/SteppablePrelude.hs" 421 21 421 56) emptyCallStack))
  [1, 2, 3]

{- Reduce function of application -> Replace binding '$dIP_a84F'
with definition -}
(\ds_d8Ej ->
  case ds_d8Ej of {
    [] ->
      (error
        pushCallStack ("error", SrcLoc "main" "SteppablePrelude"
"src/SteppablePrelude.hs" 421 21 421 56) emptyCallStack
        "Prelude.maximum: empty list");
    _ -> (foldl1 (max) ds_d8Ej)
  })
  [1, 2, 3]

{- Lamda Application -}
case [1, 2, 3] of {
  [] ->
    (error
      pushCallStack ("error", SrcLoc "main" "SteppablePrelude"
"src/SteppablePrelude.hs" 421 21 421 56) emptyCallStack
      "Prelude.maximum: empty list");
  _ -> (foldl1 (max) [1, 2, 3])
}

{- Replace with matching pattern -}
foldl1 (max) [1, 2, 3]
```

```
{- Reduce function of application -> Replace 'foldl1' with
definition -}
(\ ->
  \f ds_d8Eg ->
    case ds_d8Eg of {
      [] -> (error "Prelude.foldl1: empty list");
      : x xs -> (foldl f x xs)
    }
    where
      $dIP_a84y =
        (pushCallStack ("error", SrcLoc "main" "SteppablePrelude"
"src/SteppablePrelude.hs" 297 21 297 55) emptyCallStack))
  max
  [1, 2, 3]

{- Reduce function of application -> Replace binding '$dIP_a84y'
with definition -}
(\f ds_d8Eg ->
  case ds_d8Eg of {
    [] ->
      (error
        pushCallStack ("error", SrcLoc "main" "SteppablePrelude"
"src/SteppablePrelude.hs" 297 21 297 55) emptyCallStack
        "Prelude.foldl1: empty list");
    : x xs -> (foldl f x xs)
  })
  max
  [1, 2, 3]

{- Lamda Application -}
(\ds_d8Eg ->
  case ds_d8Eg of {
    [] ->
      (error
        pushCallStack ("error", SrcLoc "main" "SteppablePrelude"
"src/SteppablePrelude.hs" 297 21 297 55) emptyCallStack
        "Prelude.foldl1: empty list");
    : x xs -> (foldl (max) x xs)
  })
  [1, 2, 3]
```

```
{- Lamda Application -}
case [1, 2, 3] of {
  [] ->
    (error
      pushCallStack ("error", SrcLoc "main" "SteppablePrelude"
"src/SteppablePrelude.hs" 297 21 297 55) emptyCallStack
      "Prelude.foldl1: empty list");
  : x xs -> (foldl (max) x xs)
}

{- Replace with matching pattern -}
foldl (max) 1 [2, 3]

{- Reduce function of application -> Replace 'foldl' with
definition -}
(\f z ds_d8Eb ->
  case ds_d8Eb of {
    [] -> z;
    : x xs -> (foldl f (f z x) xs)
  })
  max
  1
  [2, 3]

{- Lamda Application -}
(\z ds_d8Eb ->
  case ds_d8Eb of {
    [] -> z;
    : x xs -> (foldl (max) (max z x) xs)
  })
  1
  [2, 3]

{- Lamda Application -}
(\ds_d8Eb ->
  case ds_d8Eb of {
    [] -> 1;
    : x xs -> (foldl (max) (max 1 x) xs)
  })
  [2, 3]

{- Lamda Application -}
```

```
case [2, 3] of {
  [] -> 1;
  : x xs -> (foldl (max) (max 1 x) xs)
}

{- Replace with matching pattern -}
foldl (max) (max 1 2) [3]

{- Reduce function of application -> Replace 'foldl' with
definition -}
(\f z ds_d8Eb ->
  case ds_d8Eb of {
    [] -> z;
    : x xs -> (foldl f (f z x) xs)
  })
  max
  max 1 2
  [3]

{- Lamda Application -}
(\z ds_d8Eb ->
  case ds_d8Eb of {
    [] -> z;
    : x xs -> (foldl (max) (max z x) xs)
  })
  max 1 2
  [3]

{- Lamda Application -}
(\ds_d8Eb ->
  case ds_d8Eb of {
    [] -> (max 1 2);
    : x xs -> (foldl (max) (max (max 1 2) x) xs)
  })
  [3]

{- Lamda Application -}
case [3] of {
  [] -> (max 1 2);
  : x xs -> (foldl (max) (max (max 1 2) x) xs)
}
```

```
{- Replace with matching pattern -}
foldl (max) (max (max 1 2) 3) []

{- Reduce function of application -> Replace 'foldl' with
definition -}
(\f z ds_d8Eb ->
  case ds_d8Eb of {
    [] -> z;
    : x xs -> (foldl f (f z x) xs)
  })
  max
  max (max 1 2) 3
  []

{- Lamda Application -}
(\z ds_d8Eb ->
  case ds_d8Eb of {
    [] -> z;
    : x xs -> (foldl (max) (max z x) xs)
  })
  max (max 1 2) 3
  []

{- Lamda Application -}
(\ds_d8Eb ->
  case ds_d8Eb of {
    [] -> (max (max 1 2) 3);
    : x xs -> (foldl (max) (max (max (max 1 2) 3) x) xs)
  })
  []

{- Lamda Application -}
case [] of {
  [] -> (max (max 1 2) 3);
  : x xs -> (foldl (max) (max (max (max 1 2) 3) x) xs)
}

{- Replace with matching pattern -}
max (max 1 2) 3

{- (Strict) reduce application argument -> Evaluate unsteppable
function/operator max -}
```

```
max 2 3

{- Evaluate unsteppable function/operator max -}
3

{- reduction completed successfully -}
```

Verbosity Level 3: Core Pretty Printing, Skipping Substeps

This is the output for verbosity level 3:

```
maximum
  @Integer
  $fOrdInteger
  (: @Integer 1 (: @Integer 2 (: @Integer 3 ([] @Integer))))
{- skipping 5 substeps -}

{- Replace with matching pattern -}
foldl1
  @Integer
  (max @Integer $fOrdInteger)
  (: @Integer 1 (: @Integer 2 (: @Integer 3 ([] @Integer))))
{- skipping 5 substeps -}

{- Replace with matching pattern -}
foldl
  @Integer
  @Integer
  (max @Integer $fOrdInteger)
  1
  (: @Integer 2 (: @Integer 3 ([] @Integer)))
{- skipping 6 substeps -}

{- Replace with matching pattern -}
foldl
  @Integer
  @Integer
  (max @Integer $fOrdInteger)
  (max @Integer $fOrdInteger 1 2)
  (: @Integer 3 ([] @Integer))
{- skipping 6 substeps -}

{- Replace with matching pattern -}
foldl
  @Integer
  @Integer
  (max @Integer $fOrdInteger)
  (max @Integer $fOrdInteger (max @Integer $fOrdInteger 1 2) 3)
  ([] @Integer)
{- skipping 6 substeps -}

{- Replace with matching pattern -}
```

```
max @Integer $fOrdInteger (max @Integer $fOrdInteger 1 2) 3
{- (Strict) reduce application argument -> Evaluate unsteppable
function/operator max -}
max @Integer $fOrdInteger 2 3
{- Evaluate unsteppable function/operator max -}
3
{- reduction completed successfully -}
```

Verbosity Level 4: Core Pretty Printing, No Skipping

This is the output for verbosity level 4:

```
maximum
  @Integer
  $fOrdInteger
  (: @Integer 1 (: @Integer 2 (: @Integer 3 ([] @Integer))))
{- Reduce function of application -> Replace 'maximum' with
definition -}
(\ (@a) ($dOrd_a84C :: Ord a) ->
   let {
     $dIP_a84F :: HasCallStack
     [LclId,
      Unf=Unf{Src=<vanilla>, TopLvl=False, Value=False,
ConLike=False,
              WorkFree=False, Expandable=False, Guidance=IF_ARGS
[] 340 0}]
     $dIP_a84F
       = (pushCallStack
            (unpackCString# "error"#,
             SrcLoc
               (unpackCString# "main"#)
               (unpackCString# "SteppablePrelude"#)
               (unpackCString# "src/SteppablePrelude.hs"#)
               (I# 421#)
               (I# 21#)
               (I# 421#)
               (I# 56#))
            emptyCallStack)
          `cast` (Sym (N:IP[0] <"callStack">_N <CallStack>_N)
                  :: Coercible CallStack (?
callStack::CallStack)) } in
   \ (ds_d8Ej :: [a]) ->
     case ds_d8Ej of {
       __DEFAULT -> foldl1 @a (max @a $dOrd_a84C) ds_d8Ej;
       [] ->
         error
           @'LiftedRep
           @a
           $dIP_a84F
           (unpackCString# "Prelude.maximum: empty list"#)
     })
  @Integer
  $fOrdInteger
  (: @Integer 1 (: @Integer 2 (: @Integer 3 ([] @Integer))))
{- Lamda Application -}
(\ ($dOrd_a84C :: Ord a) ->
   let {
     $dIP_a84F :: HasCallStack
     [LclId,
      Unf=Unf{Src=<vanilla>, TopLvl=False, Value=False,
ConLike=False,
              WorkFree=False, Expandable=False, Guidance=IF_ARGS
[] 340 0}]
     $dIP_a84F
       = (pushCallStack
            (unpackCString# "error"#,
             SrcLoc
               (unpackCString# "main"#)
               (unpackCString# "SteppablePrelude"#)
               (unpackCString# "src/SteppablePrelude.hs"#)
               (I# 421#)
               (I# 21#)
               (I# 421#)
               (I# 56#))
            emptyCallStack)
         `cast` (Sym (N:IP[0] <"callStack">_N <CallStack>_N)
                 :: Coercible CallStack (?
callStack::CallStack)) } in
   \ (ds_d8Ej :: [a]) ->
     case ds_d8Ej of {
       __DEFAULT -> foldl1 @Integer (max @Integer $dOrd_a84C)
ds_d8Ej;
       [] ->
         error
           @'LiftedRep
           @Integer
           $dIP_a84F
           (unpackCString# "Prelude.maximum: empty list"#)
     })
  $fOrdInteger
  (: @Integer 1 (: @Integer 2 (: @Integer 3 ([] @Integer))))
{- Lamda Application -}
(let {
   $dIP_a84F :: HasCallStack
   [LclId,
    Unf=Unf{Src=<vanilla>, TopLvl=False, Value=False,
ConLike=False,
            WorkFree=False, Expandable=False, Guidance=IF_ARGS []
340 0}]
   $dIP_a84F
     = (pushCallStack
          (unpackCString# "error"#,
           SrcLoc
             (unpackCString# "main"#)
             (unpackCString# "SteppablePrelude"#)
             (unpackCString# "src/SteppablePrelude.hs"#)
             (I# 421#)
             (I# 21#)
             (I# 421#)
             (I# 56#))
          emptyCallStack)
       `cast` (Sym (N:IP[0] <"callStack">_N <CallStack>_N)
               :: Coercible CallStack (?callStack::CallStack)) }
in
 \ (ds_d8Ej :: [a]) ->
   case ds_d8Ej of {
     __DEFAULT -> foldl1 @Integer (max @Integer $fOrdInteger)
ds_d8Ej;
     [] ->
       error
         @'LiftedRep
         @Integer
         $dIP_a84F
         (unpackCString# "Prelude.maximum: empty list"#)
   })
  (: @Integer 1 (: @Integer 2 (: @Integer 3 ([] @Integer))))
{- Reduce function of application -> Replace binding '$dIP_a84F'
with definition -}
(\ (ds_d8Ej :: [a]) ->
   case ds_d8Ej of {
     __DEFAULT -> foldl1 @Integer (max @Integer $fOrdInteger)
ds_d8Ej;
     [] ->
       error
         @'LiftedRep
         @Integer
         ((pushCallStack
             (unpackCString# "error"#,
              SrcLoc
                (unpackCString# "main"#)
                (unpackCString# "SteppablePrelude"#)
                (unpackCString# "src/SteppablePrelude.hs"#)
                (I# 421#)
                (I# 21#)
                (I# 421#)
                (I# 56#))
             emptyCallStack)
          `cast` (Sym (N:IP[0] <"callStack">_N <CallStack>_N)
                  :: Coercible CallStack (?
callStack::CallStack)))
         (unpackCString# "Prelude.maximum: empty list"#)
   })
  (: @Integer 1 (: @Integer 2 (: @Integer 3 ([] @Integer))))
{- Lamda Application -}
case : @Integer 1 (: @Integer 2 (: @Integer 3 ([] @Integer))) of
{
  __DEFAULT ->
    foldl1
      @Integer
      (max @Integer $fOrdInteger)
      (: @Integer 1 (: @Integer 2 (: @Integer 3 ([] @Integer))));
  [] ->
    error
      @'LiftedRep
      @Integer
      ((pushCallStack
          (unpackCString# "error"#,
           SrcLoc
             (unpackCString# "main"#)
             (unpackCString# "SteppablePrelude"#)
             (unpackCString# "src/SteppablePrelude.hs"#)
             (I# 421#)
             (I# 21#)
             (I# 421#)
             (I# 56#))
          emptyCallStack)
       `cast` (Sym (N:IP[0] <"callStack">_N <CallStack>_N)
               :: Coercible CallStack (?callStack::CallStack)))
      (unpackCString# "Prelude.maximum: empty list"#)
```

```
}
{- Replace with matching pattern -}
foldl1
  @Integer
  (max @Integer $fOrdInteger)
  (: @Integer 1 (: @Integer 2 (: @Integer 3 ([] @Integer))))
{- Reduce function of application -> Replace 'foldl1' with
definition -}
(\ (@a) ->
   let {
     $dIP_a84y :: HasCallStack
     [LclId,
      Unf=Unf{Src=<vanilla>, TopLvl=False, Value=False,
ConLike=False,
              WorkFree=False, Expandable=False, Guidance=IF_ARGS
[] 340 0}]
     $dIP_a84y
       = (pushCallStack
            (unpackCString# "error"#,
             SrcLoc
               (unpackCString# "main"#)
               (unpackCString# "SteppablePrelude"#)
               (unpackCString# "src/SteppablePrelude.hs"#)
               (I# 297#)
               (I# 21#)
               (I# 297#)
               (I# 55#))
            emptyCallStack)
         `cast` (Sym (N:IP[0] <"callStack">_N <CallStack>_N)
                 :: Coercible CallStack (?
callStack::CallStack)) } in
   \ (f :: a -> a -> a) (ds_d8Eg :: [a]) ->
     case ds_d8Eg of {
       [] ->
         error
           @'LiftedRep
           @a
           $dIP_a84y
           (unpackCString# "Prelude.foldl1: empty list"#);
       : x xs -> foldl @a @a f x xs
     })
  @Integer
  (max @Integer $fOrdInteger)
  (: @Integer 1 (: @Integer 2 (: @Integer 3 ([] @Integer))))
{- Lamda Application -}
(let {
   $dIP_a84y :: HasCallStack
   [LclId,
    Unf=Unf{Src=<vanilla>, TopLvl=False, Value=False,
ConLike=False,
            WorkFree=False, Expandable=False, Guidance=IF_ARGS []
340 0}]
   $dIP_a84y
     = (pushCallStack
          (unpackCString# "error"#,
           SrcLoc
             (unpackCString# "main"#)
             (unpackCString# "SteppablePrelude"#)
             (unpackCString# "src/SteppablePrelude.hs"#)
             (I# 297#)
             (I# 21#)
             (I# 297#)
             (I# 55#))
          emptyCallStack)
       `cast` (Sym (N:IP[0] <"callStack">_N <CallStack>_N)
                 :: Coercible CallStack (?callStack::CallStack)) }
in
 \ (f :: a -> a -> a) (ds_d8Eg :: [a]) ->
   case ds_d8Eg of {
     [] ->
       error
         @'LiftedRep
         @Integer
         $dIP_a84y
         (unpackCString# "Prelude.foldl1: empty list"#);
     : x xs -> foldl @Integer @Integer f x xs
   })
  (max @Integer $fOrdInteger)
  (: @Integer 1 (: @Integer 2 (: @Integer 3 ([] @Integer))))
{- Reduce function of application -> Replace binding '$dIP_a84y'
with definition -}
(\ (f :: a -> a -> a) (ds_d8Eg :: [a]) ->
   case ds_d8Eg of {
     [] ->
```

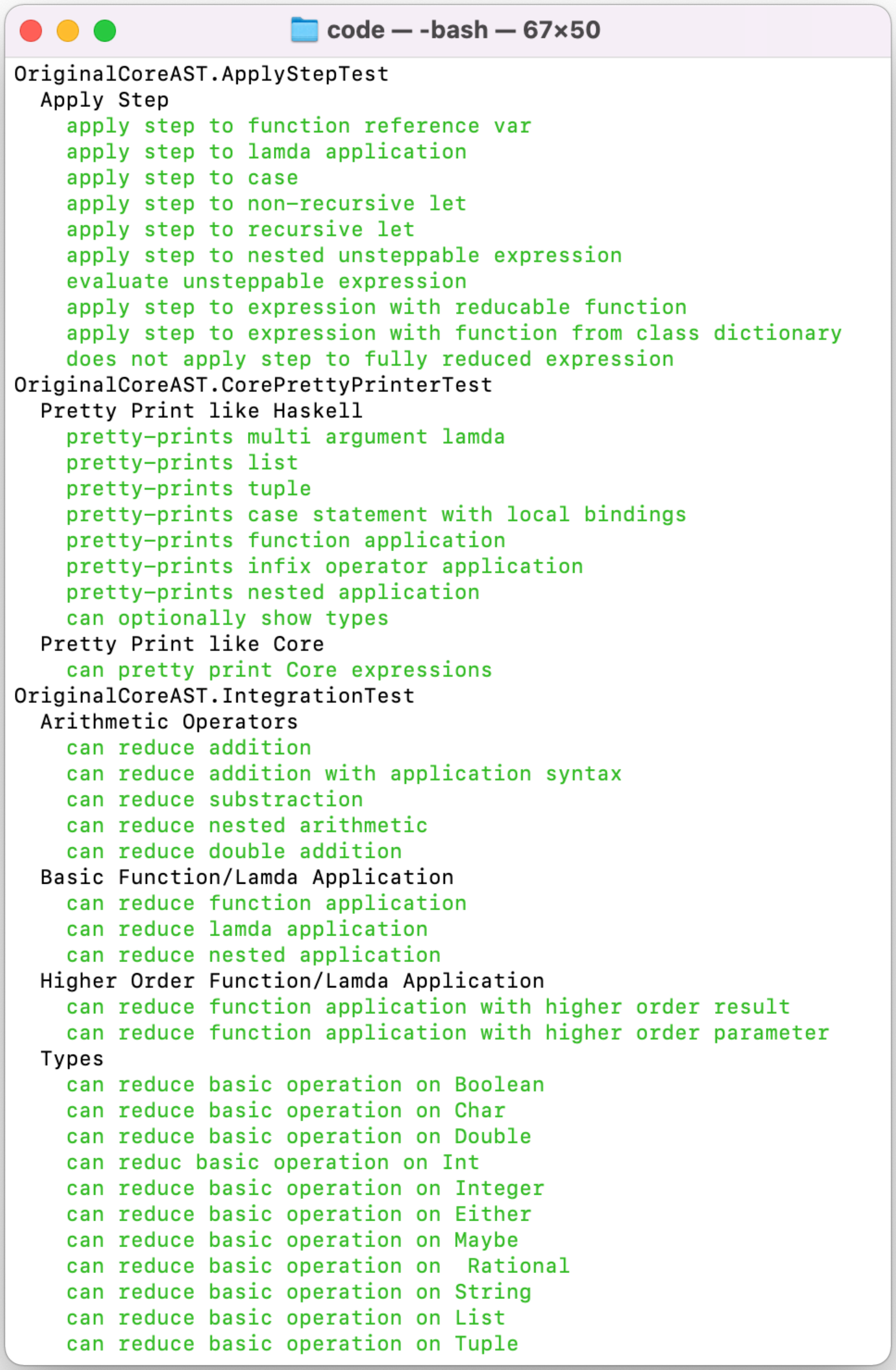
```
        error
          @'LiftedRep
          @Integer
          ((pushCallStack
              (unpackCString# "error"#,
               SrcLoc
                 (unpackCString# "main"#)
                 (unpackCString# "SteppablePrelude"#)
                 (unpackCString# "src/SteppablePrelude.hs"#)
                 (I# 297#)
                 (I# 21#)
                 (I# 297#)
                 (I# 55#))
              emptyCallStack)
           `cast` (Sym (N:IP[0] <"callStack">_N <CallStack>_N)
                   :: Coercible CallStack (?
callStack::CallStack)))
          (unpackCString# "Prelude.foldl1: empty list"#);
      : x xs -> foldl @Integer @Integer f x xs
   })
  (max @Integer $fOrdInteger)
  (: @Integer 1 (: @Integer 2 (: @Integer 3 ([] @Integer))))
{- Lamda Application -}
(\ (ds_d8Eg :: [a]) ->
   case ds_d8Eg of {
     [] ->
       error
         @'LiftedRep
         @Integer
         ((pushCallStack
             (unpackCString# "error"#,
              SrcLoc
                (unpackCString# "main"#)
                (unpackCString# "SteppablePrelude"#)
                (unpackCString# "src/SteppablePrelude.hs"#)
                (I# 297#)
                (I# 21#)
                (I# 297#)
                (I# 55#))
             emptyCallStack)
          `cast` (Sym (N:IP[0] <"callStack">_N <CallStack>_N)
                   :: Coercible CallStack (?
callStack::CallStack)))
         (unpackCString# "Prelude.foldl1: empty list"#);
     : x xs -> foldl @Integer @Integer (max @Integer
$fOrdInteger) x xs
   })
  (: @Integer 1 (: @Integer 2 (: @Integer 3 ([] @Integer))))
{- Lamda Application -}
case : @Integer 1 (: @Integer 2 (: @Integer 3 ([] @Integer))) of
{
  [] ->
    error
      @'LiftedRep
      @Integer
      ((pushCallStack
          (unpackCString# "error"#,
           SrcLoc
             (unpackCString# "main"#)
             (unpackCString# "SteppablePrelude"#)
             (unpackCString# "src/SteppablePrelude.hs"#)
             (I# 297#)
             (I# 21#)
             (I# 297#)
             (I# 55#))
          emptyCallStack)
       `cast` (Sym (N:IP[0] <"callStack">_N <CallStack>_N)
               :: Coercible CallStack (?callStack::CallStack)))
      (unpackCString# "Prelude.foldl1: empty list"#);
  : x xs -> foldl @Integer @Integer (max @Integer $fOrdInteger) x
xs
}
{- Replace with matching pattern -}
foldl
  @Integer
  @Integer
  (max @Integer $fOrdInteger)
  1
  (: @Integer 2 (: @Integer 3 ([] @Integer)))
{- Reduce function of application -> Replace 'foldl' with
definition -}
(\ (@a) (@b) (f :: a -> b -> a) (z :: a) (ds_d8Eb :: [b]) ->
   case ds_d8Eb of {
     [] -> z;
     : x xs -> foldl @a @b f (f z x) xs
   })
  @Integer
  @Integer
  (max @Integer $fOrdInteger)
  1
  (: @Integer 2 (: @Integer 3 ([] @Integer)))
{- Lamda Application -}
(\ (@b) (f :: a -> b -> a) (z :: a) (ds_d8Eb :: [b]) ->
   case ds_d8Eb of {
     [] -> z;
     : x xs -> foldl @Integer @b f (f z x) xs
   })
  @Integer
  (max @Integer $fOrdInteger)
  1
  (: @Integer 2 (: @Integer 3 ([] @Integer)))
{- Lamda Application -}
(\ (f :: a -> b -> a) (z :: a) (ds_d8Eb :: [b]) ->
   case ds_d8Eb of {
     [] -> z;
     : x xs -> foldl @Integer @Integer f (f z x) xs
   })
  (max @Integer $fOrdInteger)
  1
  (: @Integer 2 (: @Integer 3 ([] @Integer)))
{- Lamda Application -}
(\ (z :: a) (ds_d8Eb :: [b]) ->
   case ds_d8Eb of {
     [] -> z;
     : x xs ->
       foldl
         @Integer
         @Integer
         (max @Integer $fOrdInteger)
         (max @Integer $fOrdInteger z x)
         xs
   })
  1 (: @Integer 2 (: @Integer 3 ([] @Integer)))
{- Lamda Application -}
(\ (ds_d8Eb :: [b]) ->
   case ds_d8Eb of {
     [] -> 1;
     : x xs ->
       foldl
         @Integer
         @Integer
         (max @Integer $fOrdInteger)
         (max @Integer $fOrdInteger 1 x)
         xs
   })
  (: @Integer 2 (: @Integer 3 ([] @Integer)))
{- Lamda Application -}
case : @Integer 2 (: @Integer 3 ([] @Integer)) of {
  [] -> 1;
  : x xs ->
   foldl
     @Integer
     @Integer
     (max @Integer $fOrdInteger)
     (max @Integer $fOrdInteger 1 x)
     xs
}
{- Replace with matching pattern -}
foldl
  @Integer
  @Integer
  (max @Integer $fOrdInteger)
  (max @Integer $fOrdInteger 1 2)
  (: @Integer 3 ([] @Integer))
{- Reduce function of application -> Replace 'foldl' with
definition -}
(\ (@a) (@b) (f :: a -> b -> a) (z :: a) (ds_d8Eb :: [b]) ->
   case ds_d8Eb of {
     [] -> z;
     : x xs -> foldl @a @b f (f z x) xs
   })
  @Integer
  @Integer
  (max @Integer $fOrdInteger)
  (max @Integer $fOrdInteger 1 2)
  (: @Integer 3 ([] @Integer))
{- Lamda Application -}
```

```
(\ (@b) (f :: a -> b -> a) (z :: a) (ds_d8Eb :: [b]) ->
   case ds_d8Eb of {
     [] -> z;
     : x xs -> foldl @Integer @b f (f z x) xs
   })
  @Integer
  (max @Integer $fOrdInteger)
  (max @Integer $fOrdInteger 1 2)
  (: @Integer 3 ([] @Integer))
{- Lamda Application -}
(\ (f :: a -> b -> a) (z :: a) (ds_d8Eb :: [b]) ->
   case ds_d8Eb of {
     [] -> z;
     : x xs -> foldl @Integer @Integer f (f z x) xs
   })
  (max @Integer $fOrdInteger)
  (max @Integer $fOrdInteger 1 2)
  (: @Integer 3 ([] @Integer))
{- Lamda Application -}
(\ (z :: a) (ds_d8Eb :: [b]) ->
   case ds_d8Eb of {
     [] -> z;
     : x xs ->
       foldl
         @Integer
         @Integer
         (max @Integer $fOrdInteger)
         (max @Integer $fOrdInteger z x)
         xs
   })
  (max @Integer $fOrdInteger 1 2) (: @Integer 3 ([] @Integer))
{- Lamda Application -}
(\ (ds_d8Eb :: [b]) ->
   case ds_d8Eb of {
     [] -> max @Integer $fOrdInteger 1 2;
     : x xs ->
       foldl
         @Integer
         @Integer
         (max @Integer $fOrdInteger)
         (max @Integer $fOrdInteger (max @Integer $fOrdInteger 1
2) x)
```

```
        xs
   })
  (: @Integer 3 ([] @Integer))
{- Lamda Application -}
case : @Integer 3 ([] @Integer) of {
  [] -> max @Integer $fOrdInteger 1 2;
  : x xs ->
    foldl
      @Integer
      @Integer
      (max @Integer $fOrdInteger)
      (max @Integer $fOrdInteger (max @Integer $fOrdInteger 1 2)
x)
      xs
}
{- Replace with matching pattern -}
foldl
  @Integer
  @Integer
  (max @Integer $fOrdInteger)
  (max @Integer $fOrdInteger (max @Integer $fOrdInteger 1 2) 3)
  ([] @Integer)
{- Reduce function of application -> Replace 'foldl' with
definition -}
(\ (@a) (@b) (f :: a -> b -> a) (z :: a) (ds_d8Eb :: [b]) ->
   case ds_d8Eb of {
     [] -> z;
     : x xs -> foldl @a @b f (f z x) xs
   })
  @Integer
  @Integer
  (max @Integer $fOrdInteger)
  (max @Integer $fOrdInteger (max @Integer $fOrdInteger 1 2) 3)
  ([] @Integer)
{- Lamda Application -}
(\ (@b) (f :: a -> b -> a) (z :: a) (ds_d8Eb :: [b]) ->
   case ds_d8Eb of {
     [] -> z;
     : x xs -> foldl @Integer @b f (f z x) xs
   })
  @Integer
  (max @Integer $fOrdInteger)
  (max @Integer $fOrdInteger (max @Integer $fOrdInteger 1 2) 3)
  ([] @Integer)
{- Lamda Application -}
(\ (f :: a -> b -> a) (z :: a) (ds_d8Eb :: [b]) ->
   case ds_d8Eb of {
     [] -> z;
     : x xs -> foldl @Integer @Integer f (f z x) xs
   })
  (max @Integer $fOrdInteger)
  (max @Integer $fOrdInteger (max @Integer $fOrdInteger 1 2) 3)
  ([] @Integer)
{- Lamda Application -}
(\ (z :: a) (ds_d8Eb :: [b]) ->
   case ds_d8Eb of {
     [] -> z;
     : x xs ->
       foldl
         @Integer
         @Integer
         (max @Integer $fOrdInteger)
         (max @Integer $fOrdInteger z x)
         xs
   })
  (max @Integer $fOrdInteger (max @Integer $fOrdInteger 1 2) 3)
  ([] @Integer)
{- Lamda Application -}
(\ (ds_d8Eb :: [b]) ->
   case ds_d8Eb of {
     [] -> max @Integer $fOrdInteger (max @Integer $fOrdInteger 1
2) 3;
     : x xs ->
       foldl
         @Integer
         @Integer
         (max @Integer $fOrdInteger)
         (max
            @Integer
            $fOrdInteger
            (max @Integer $fOrdInteger (max @Integer $fOrdInteger
1 2) 3)
            x)
         xs
```

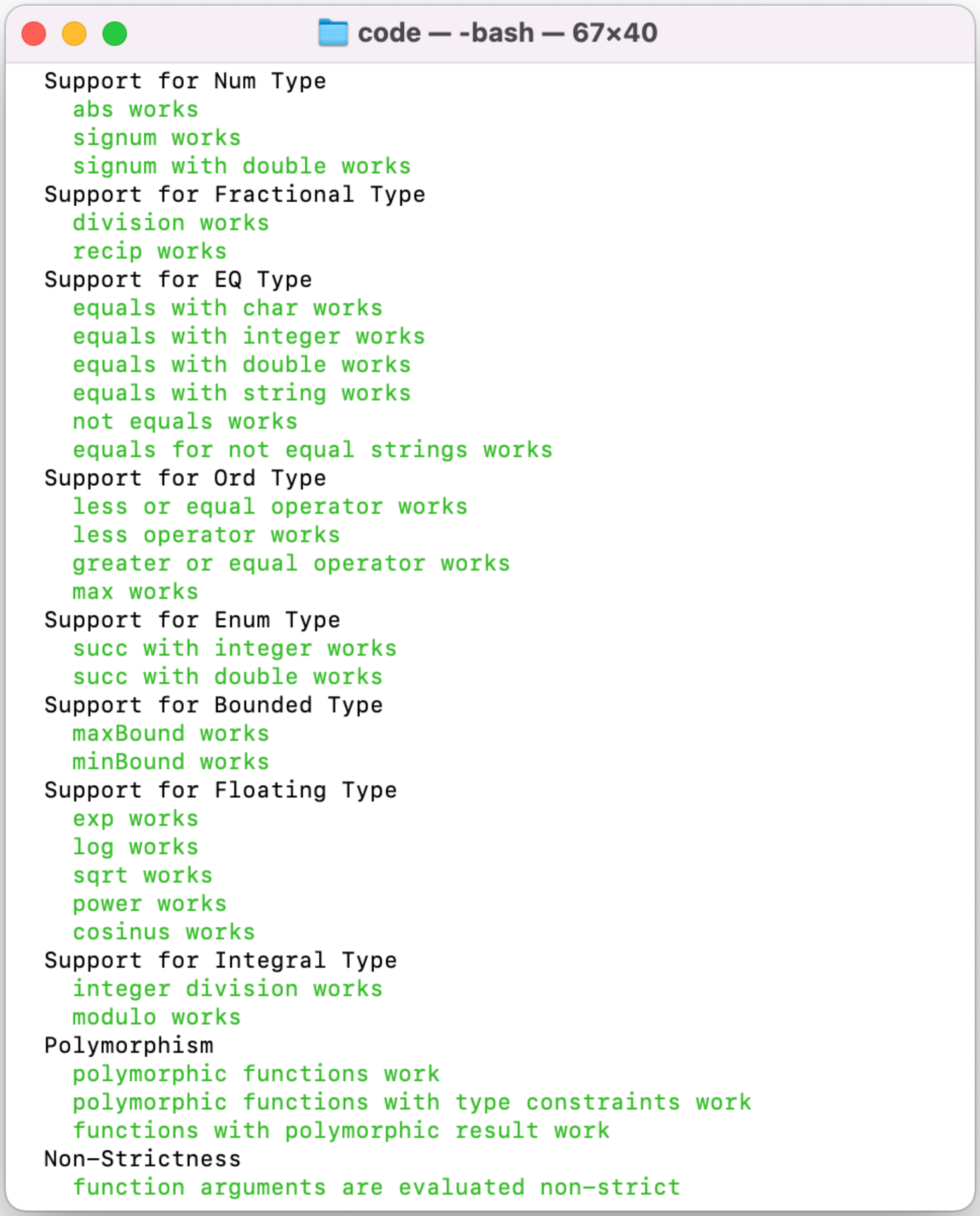
```
    })
  ([] @Integer)
{- Lamda Application -}
case [] @Integer of {
  [] -> max @Integer $fOrdInteger (max @Integer $fOrdInteger 1 2)
3;
  : x xs ->
    foldl
      @Integer
      @Integer
      (max @Integer $fOrdInteger)
      (max
         @Integer
         $fOrdInteger
         (max @Integer $fOrdInteger (max @Integer $fOrdInteger 1
2) 3)
         x)
      xs
}
{- Replace with matching pattern -}
max @Integer $fOrdInteger (max @Integer $fOrdInteger 1 2) 3
{- (Strict) reduce application argument -> Evaluate unsteppable
function/operator max -}
max @Integer $fOrdInteger 2 3
{- Evaluate unsteppable function/operator max -}
3
{- reduction completed successfully -}
```

10.3 Automated Testing Output



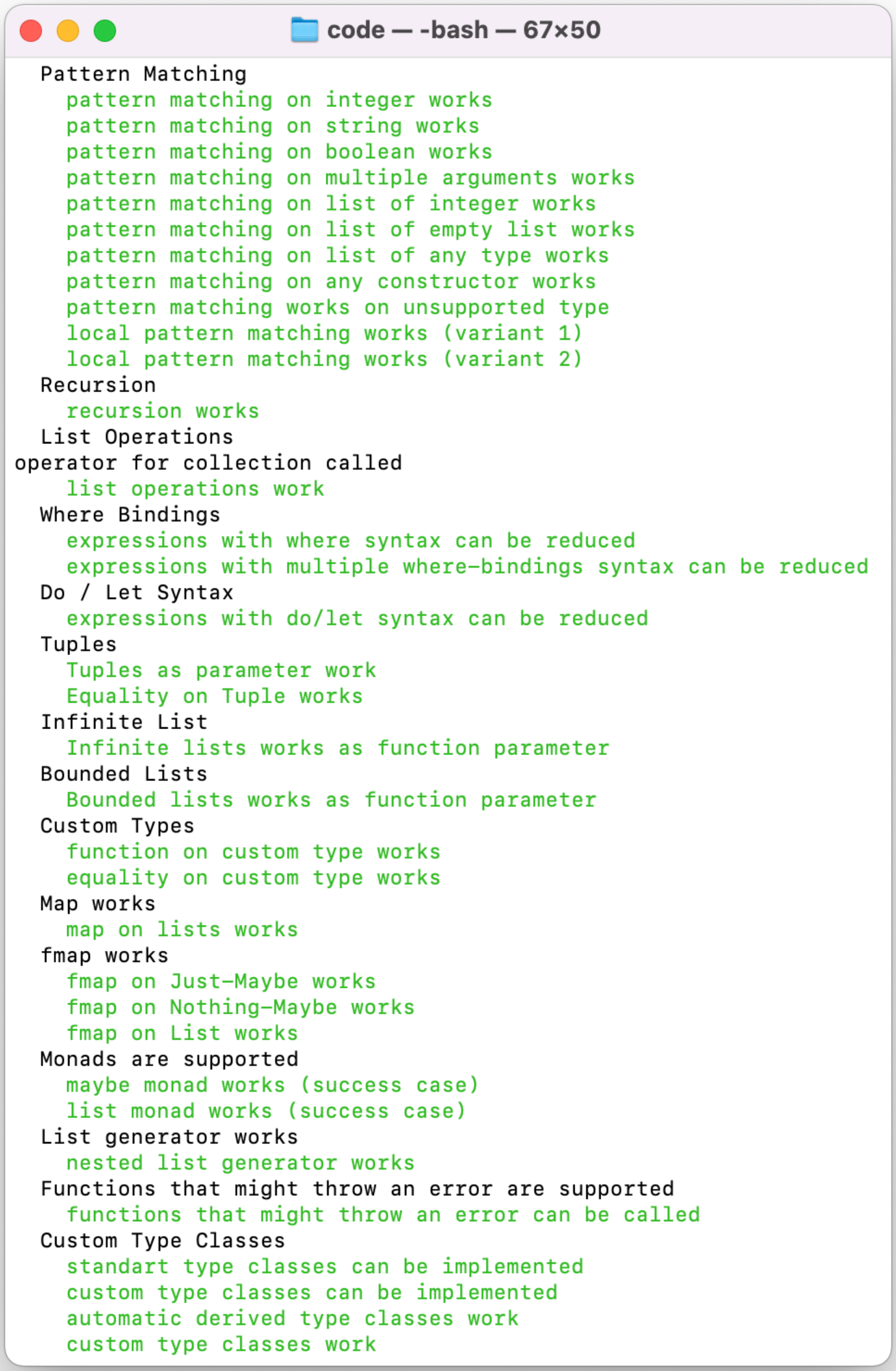
```
OriginalCoreAST.ApplyStepTest
  Apply Step
    apply step to function reference var
    apply step to lamda application
    apply step to case
    apply step to non-recursive let
    apply step to recursive let
    apply step to nested unsteppable expression
    evaluate unsteppable expression
    apply step to expression with reducable function
    apply step to expression with function from class dictionary
    does not apply step to fully reduced expression
OriginalCoreAST.CorePrettyPrinterTest
  Pretty Print like Haskell
    pretty-prints multi argument lamda
    pretty-prints list
    pretty-prints tuple
    pretty-prints case statement with local bindings
    pretty-prints function application
    pretty-prints infix operator application
    pretty-prints nested application
    can optionally show types
  Pretty Print like Core
    can pretty print Core expressions
OriginalCoreAST.IntegrationTest
  Arithmetic Operators
    can reduce addition
    can reduce addition with application syntax
    can reduce substraction
    can reduce nested arithmetic
    can reduce double addition
  Basic Function/Lamda Application
    can reduce function application
    can reduce lamda application
    can reduce nested application
  Higher Order Function/Lamda Application
    can reduce function application with higher order result
    can reduce function application with higher order parameter
  Types
    can reduce basic operation on Boolean
    can reduce basic operation on Char
    can reduce basic operation on Double
    can reduc basic operation on Int
    can reduce basic operation on Integer
    can reduce basic operation on Either
    can reduce basic operation on Maybe
    can reduce basic operation on  Rational
    can reduce basic operation on String
    can reduce basic operation on List
    can reduce basic operation on Tuple
```

Figure 33: Jira board (1/4)



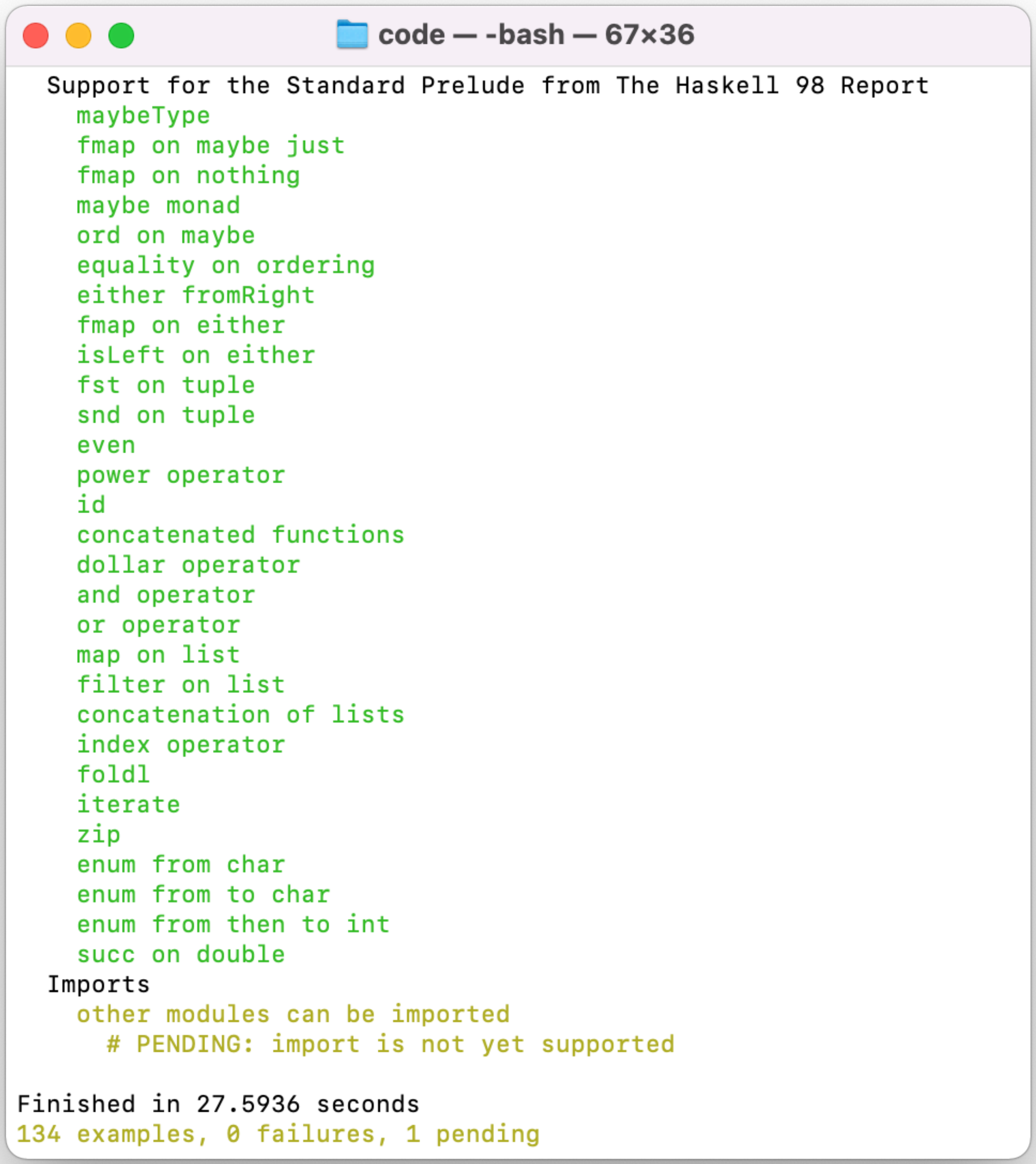
```
Support for Num Type
  abs works
  signum works
  signum with double works
Support for Fractional Type
  division works
  recip works
Support for EQ Type
  equals with char works
  equals with integer works
  equals with double works
  equals with string works
  not equals works
  equals for not equal strings works
Support for Ord Type
  less or equal operator works
  less operator works
  greater or equal operator works
  max works
Support for Enum Type
  succ with integer works
  succ with double works
Support for Bounded Type
  maxBound works
  minBound works
Support for Floating Type
  exp works
  log works
  sqrt works
  power works
  cosinus works
Support for Integral Type
  integer division works
  modulo works
Polymorphism
  polymorphic functions work
  polymorphic functions with type constraints work
  functions with polymorphic result work
Non-Strictness
  function arguments are evaluated non-strict
```

Figure 34: Testing output (2/4)



```
  Pattern Matching
    pattern matching on integer works
    pattern matching on string works
    pattern matching on boolean works
    pattern matching on multiple arguments works
    pattern matching on list of integer works
    pattern matching on list of empty list works
    pattern matching on list of any type works
    pattern matching on any constructor works
    pattern matching works on unsupported type
    local pattern matching works (variant 1)
    local pattern matching works (variant 2)
  Recursion
    recursion works
  List Operations
operator for collection called
    list operations work
  Where Bindings
    expressions with where syntax can be reduced
    expressions with multiple where-bindings syntax can be reduced
  Do / Let Syntax
    expressions with do/let syntax can be reduced
  Tuples
    Tuples as parameter work
    Equality on Tuple works
  Infinite List
    Infinite lists works as function parameter
  Bounded Lists
    Bounded lists works as function parameter
  Custom Types
    function on custom type works
    equality on custom type works
  Map works
    map on lists works
  fmap works
    fmap on Just-Maybe works
    fmap on Nothing-Maybe works
    fmap on List works
  Monads are supported
    maybe monad works (success case)
    list monad works (success case)
  List generator works
    nested list generator works
  Functions that might throw an error are supported
    functions that might throw an error can be called
  Custom Type Classes
    standart type classes can be implemented
    custom type classes can be implemented
    automatic derived type classes work
    custom type classes work
```

Figure 35: Testing output (3/4)



```
  Support for the Standard Prelude from The Haskell 98 Report
    maybeType
    fmap on maybe just
    fmap on nothing
    maybe monad
    ord on maybe
    equality on ordering
    either fromRight
    fmap on either
    isLeft on either
    fst on tuple
    snd on tuple
    even
    power operator
    id
    concatenated functions
    dollar operator
    and operator
    or operator
    map on list
    filter on list
    concatenation of lists
    index operator
    foldl
    iterate
    zip
    enum from char
    enum from to char
    enum from then to int
    succ on double
  Imports
    other modules can be imported
      # PENDING: import is not yet supported

Finished in 27.5936 seconds
134 examples, 0 failures, 1 pending
```

Figure 36: Testing output (4/4)

References

[1] G. Hutton, *Programming in Haskell*, 2nd ed. Cambridge University Press.

[2] “Sharing - HaskellWiki.” [Online]. Available: https://wiki.haskell.org/Sharing

[3] “Non-strict semantics.” [Online]. Available: https://wiki.haskell.org/Non-strict_semantics

[4] Lazy vs. Non-strict. Haskell Wiki. [Online]. Available: https://wiki.haskell.org/Lazy_vs._non-strict

[5] “Performance/Strictness - HaskellWiki.” [Online]. Available: https://wiki.haskell.org/Performance/Strictness

[6] B. O’Sullivan, J. Goerzen, and D. B. Stewart, *Real World Haskell: Code You Can Believe In*, 1st ed. O’Reilly.

[7] “Haskell/Graph reduction.” [Online]. Available: https://en.wikibooks.org/wiki/Haskell/Graph_reduction

[8] Vladislav Zavialov. Haskell to Core: Understanding Haskell Features Through Their Desugaring. Serokell Software Development Company. [Online]. Available: https://serokell.io/blog/haskell-to-core

[9] S. L. P. Jones, “Implementing lazy functional languages on stock hardware: The Spineless Tagless G-Machine - Version 2.5,” vol. 2, pp. 127–202.

[10] Kirill Elagin. I know kung fu: Learning STG by example. GitLab. [Online]. Available: https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/generated-code

[11] David Luposchainsky, “ZuriHac 2016 - Low-Level Haskell: An Interactive Tour Through the STG.” [Online]. Available: https://www.youtube.com/watch?v=-MFk7PIKYsg

[12] Tom. Haskell-programs-how-do-they-run. [Online]. Available: http://h2.jaguarpaw.co.uk/posts/haskell-programs-how-do-they-run/

[13] Christoph Bauer. GHC Commentary: What the hell is a .Cmm file? GitLab. [Online]. Available: https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/rts/cmm

[14] Haskell 2010 Language Report. [Online]. Available: https://www.haskell.org/onlinereport/haskell2010/

[15] S. Najd and S. P. Jones. Trees That Grow. [Online]. Available: http://arxiv.org/abs/1610.04799

[16] T. Olmer, B. Heeren, and J. Jeuring, “Evaluating Haskell expressions in a tutoring environment,” vol. 170, pp. 50–66. [Online]. Available: http://arxiv.org/abs/1412.4879

[17] B. Millwood, “Stepeval: Project post-mortem.” [Online]. Available: https://github.com/bmillwood/stepeval/blob/4fb70ae412f35c95f4f83c0229d3a05b6a027524/README.md

[18] The GHCi Debugger. [Online]. Available: https://downloads.haskell.org/~ghc/6.10-latest/docs/html/users_guide/ghci-debugger.html

[19] T. Furukawa, Y. Cong, and K. Asai, “Stepping OCaml,” vol. 295, pp. 17–34. [Online]. Available: http://arxiv.org/abs/1906.11422

[20] “Haskell Core Specification.” [Online]. Available: https://github.com/ghc/ghc/blob/ef92a0095cee1f623fba1c285c1836e80bf16223/docs/core-spec/core-spec.pdf

[21] (04.12.21) Compiler/GHC/Core.hs · Glasgow Haskell Compiler / GHC. gitlab.haskell.org. [Online]. Available: https://gitlab.haskell.org/ghc/ghc/blob/master/compiler/GHC/Core.hs

[22] A Haskell Compiler. Stanford Secure Computer Systems group. [Online]. Available: https://www.scs.stanford.edu/11au-cs240h/notes/ghc.html

[23] "Into the Core - Squeezing Haskell into Nine Constructors by Simon Peyton Jones." [Online]. Available: https://www.youtube.com/watch?v=uR_VzYxvbxg

[24] Haskell Report 2010: Standard Prelude. [Online]. Available: https://www.haskell.org/onlinereport/haskell2010/haskellch9.html#x16-1710009

[25] Haskell Report 2010: Specification of Derived Instances. [Online]. Available: https://www.haskell.org/onlinereport/haskell2010/haskellch11.html

[26] A. Pang, "[Haskell] Overriding Prelude functions," Mon Jul 5 03:59:45 EDT 2004. [Online]. Available: https://mail.haskell.org/pipermail/haskell/2004-July/014307.html