$\boxed{\text{Bachelor Thesis}}$

Project Documentation

Haskell Substitution Stepper

Semester: Spring 2023

Date: 2023-06-23 13:04:07+02:00

Author: Carlo Del Rossi
Project Advisor: Prof. Dr. Farhad D. Mehta
Co-Examiner: Dr. Joachim Breitner

School of Computer Science
OST Eastern Switzerland University of Applied Sciences

Abstract

With functional programming languages becoming more widespread and being taught at Universities, many programmers will eventually get in contact with them. Since functional programs can be very different from imperative ones, especially due to the heavy reliance on recursion, this could be very interesting for all the people that are not yet familiar with Haskell's concepts.

While functional languages like Haskell have debuggers, they are not as user-friendly and don't offer as much insight as debuggers for imperative languages. The internal state of the program is usually not displayed very comprehensibly, which leads to the not being very useful for learning processes.

The goal of the Haskell Substitution Stepper is to provide the user with the capability to step through a Haskell program and to be able to see what happens in the background when a function is executed. This is supposed to solve the usability problems that the regular Haskell debugger has.

For this, we came up with an application where the user can specify a function that he would like to step through and the application would load this function and then use Haskell's rules to step through. The user can see the whole internal state of the term that is being stepped through and the highlighting of the changes makes it easy to see what happens in each step. The user can choose between different modes of derivation and can control the flow of the derivation. The addition of helpful commands also allows the user to skip certain parts of the derivation that might not be interesting to them.

Glossary

Alternative one of the subterms that the scrutinee is matched against. An alternative defines a pattern that the scrutinee is being matched against and a replacement for the scrutinee, in case the scrutinee matches the pattern

Binding The combination of a function- or variable name and the definition. The definition is 'bound' to the name/identifier.

Core An intermediate language that Haskell is compiled to. It is a simplified language compared to core but it can do anything that Haskell can do too.

Derivation A step-by-step reduction of a term, often resulting in normal form

Desugaring The process of removing certain Haskell features to create a simpler term to derive. Can be thought of removal of 'syntactic sugar'.

LHS Left-Hand-Side

RHS Right-Hand-Side

Redex A (sub)term that is reducible. I.e. a reduction rule can be applied to the term successfully.

Reduction A rule-based step applied to a term that changes the structure of that term

Scrutinee A term examined in a case expression

Sharing A mechanism used by Haskell to prevent re-evaluation of the same term.

WHNF Weak Head Normal Form

Contents

Part I

Introduction

Chapter 1

Management Summary

Das Management Summary (auch Lay Summary) richtet sich an ein breites Publikum und an das Management, welches in der Regel über keine Fachkenntnisse im bearbeiteten Thema verfügen. Das Management Summary soll kurz und verständlich beschreiben, worum es bei der Arbeit geht und welche Ergebnisse erzielt wurden. Die Sprache soll knapp, klar und stark untergliedert sein. Der Umfang beträgt in der Regel 2-3 (max. 5) Seiten. Bilder sind hier im Gegensatz zum Abstract erwünscht. Beispiel Gliederung für Management Summary:

1.1 Initial Situation

Some functional languages, like Haskell, don't come with debuggers that are very user-friendly. While Haskell does have a debugger, it is fairly hard to use and it is not obvious, how some information is obtained.

The debugger seems to jump around, and when entering a function, it seems like the context of the function call is lost. It is not easy to see the values that are used when calling a function, as they need to be fetched manually.

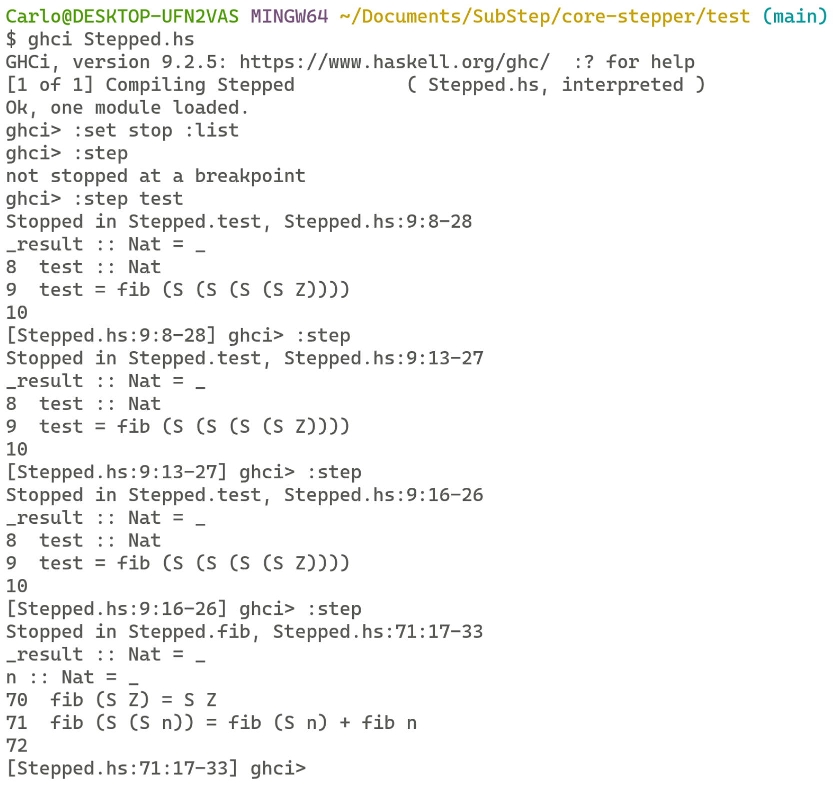
To overcome these problems, the Substitution Stepper was built as an alternative, that makes it much easier to visualize the state of the program.

Some sketches of how the Substitution Stepper could work are shown in **??**

1.2 Implementation

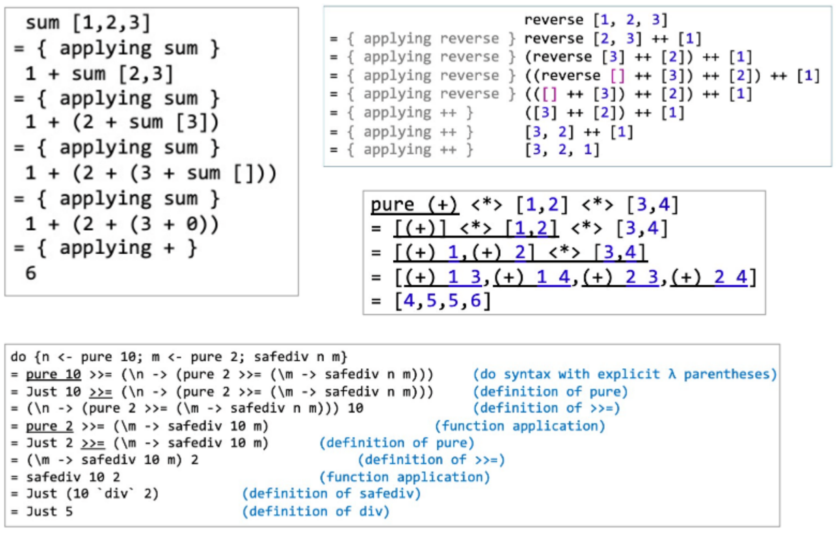
For the stepping the terms, Haskell is compiled down to Core, an intermediate langauge, that simplifies many Haskell features to allow for better optimization and easier execution.

While it is easier to step Core, it also comes with challenges translating it back to Haskell for the user.



```
Carlo@DESKTOP-UFN2VAS MINGW64 ~/Documents/SubStep/core-stepper/test (main)
$ ghci Stepped.hs
GHCi, version 9.2.5: https://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Stepped          ( Stepped.hs, interpreted )
Ok, one module loaded.
ghci> :set stop :list
ghci> :step
not stopped at a breakpoint
ghci> :step test
Stopped in Stepped.test, Stepped.hs:9:8-28
_result :: Nat = _
8  test :: Nat
9  test = fib (S (S (S (S Z))))
10
[Stepped.hs:9:8-28] ghci> :step
Stopped in Stepped.test, Stepped.hs:9:13-27
_result :: Nat = _
8  test :: Nat
9  test = fib (S (S (S (S Z))))
10
[Stepped.hs:9:13-27] ghci> :step
Stopped in Stepped.test, Stepped.hs:9:16-26
_result :: Nat = _
8  test :: Nat
9  test = fib (S (S (S (S Z))))
10
[Stepped.hs:9:16-26] ghci> :step
Stopped in Stepped.fib, Stepped.hs:71:17-33
_result :: Nat = _
n :: Nat = _
70  fib (S Z) = S Z
71  fib (S (S n)) = fib (S n) + fib n
72
[Stepped.hs:71:17-33] ghci>
```

Figure 1.1: Screenshot of the usage of the GHCi debugger. It is not clear what is going on and what the steps do.



```
 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 1.2: Some examples of how the Substitution Stepper could work.

1.3 Results

The result of this thesis is a command-line application that allows the stepping of Haskell code. Most Haskell code can be stepped without problems, however, there are some limitations, since it is currently not possible to step functions using the Prelude, which is the standard library for Haskell functions.

Besides the easier readability of the Substitution Stepper over the debugger, it also comes with many useful features. The user can step any term that they want in any order. And there are different modes available, also including an automatic mode that does the derivation automatically and shows the result when it's done.

Chapter 2

Scope

This chapter contains information about the scope of this thesis and details which parts of the source code were created during this thesis and which parts were pre-existing.

Originally, the focus of this thesis was completely on improving the usability of the Substitution Stepper. However, during the duration of the project, some issues came up, which showed that a reimplementation of the stepping mechanism might be beneficial.

Thus, the scope of the thesis grew and included the reimplementation of the stepping mechanism. Features like sharing were dropped in favor of simplicity.

Almost the whole source code was written as part of the thesis, only excluding the code that is contained within the CoreStepper folder.

2.1 Task Description

On the following pages, the task description is included, further outlining the scope of the thesis.

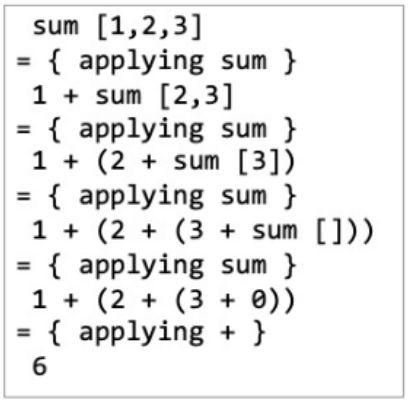
Improving the Usability of the Haskell Substitution Stepper

Task Description

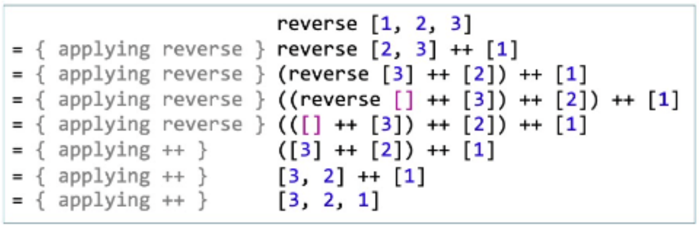
1. Setting

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.

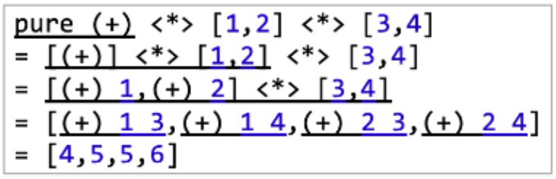
The functional programming paradigm does not have the concept of a program counter or internal state. Other techniques are therefore required to visualise the execution of functional programs. Executing a program in a functional programming language is typically viewed as evaluating an expression using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



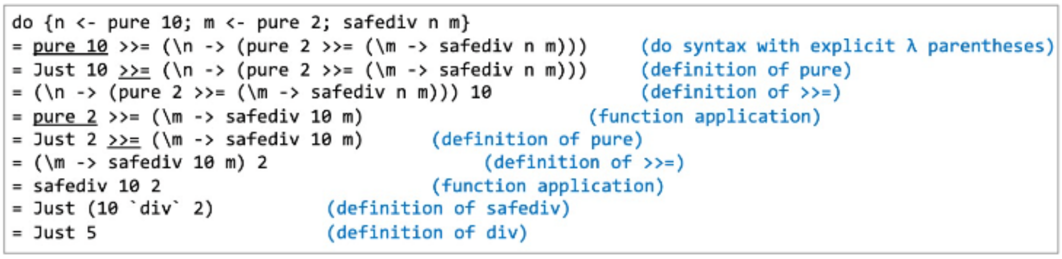
```
 sum [1,2,3]
= { applying sum }
 1 + sum [2,3]
= { applying sum }
 1 + (2 + sum [3])
= { applying sum }
 1 + (2 + (3 + sum []))
= { applying sum }
 1 + (2 + (3 + 0))
= { applying + }
 6
```



```
                       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)
```

Derivations such as these are used when teaching functional programming and reasoning about functional programs. Having tool support for generating such derivations could greatly help learning programming in and debugging programs written in the functional style.

The Haskell Substitution Stepper (HaskSubStep) is a tool that proposes to interactively generate such derivations. Although the current version of the tool is capable of generating derivations of a large subset of Haskell expressions, the current user interface requires improvements in order for the tool to be used effectively by its target audience.

2. Goals

The main aim of this project is to improve the usability of HaskSubStep. It plans to do this by proposing an appropriate concept for user interaction for HaskSubStep and implementing it as part of the tool in the language Haskell.

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

1. The tool must be usable for a beginner with little experience in functional programming. Its user interface must be carefully designed to this end.
2. The project should start with an initial study to consider how people could use HaskSubStep and conclude with a concept on the best way(s) to interact with HaskSubStep.
3. Since derivations may be long, 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. Since individual expressions at each step of a derivation may be long, the tool should allow hiding parts of large expressions that remain the same, or allow highlighting subexpressions that change between steps.
5. The user interaction may be implemented as a command-line interface (CLI), or a graphical user interface (GUI), or both.
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. Contributions to other parts of HaskSubStep as also possible in case this is required for satisfying the main aim of the project, or in case there are remaining resources.
9. The results may be used in the current run of the functional programming course at the OST.
10. The results will be released under the GPLv3 license.

Further refinements and modifications to this list that serve the main aim of this project are possible during its course.

3. Deliverables

- All artefacts (source code, web pages, accepted pull requests, etc.) required to achieve the goals of this project.
- Product documentation in English that is relevant to the use and further development of the results (e.g., requirements, domain model, architecture description, code documentation, user manuals, etc.) in a form that can be developed further and is amenable to version control (e.g., LaTeX or Markdown). Ideally, all product documentation should be contained withing the artefacts required to achieve the goals of this project as stated in above.
- Project documentation that is separate from the product documentation that briefly, but precisely 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.

4. Stakeholders

Partner: Software Engineering and Programming Language Lab, IFS Institute for Software, OST
Student: Carlo Del Rossi
Supervisor: Farhad Mehta.

5. Other Project Details

Type of project: Bachelor Thesis Project (de: Bachelorarbeit)
Duration: 20.02.2023 – 16.06.2023
Workload per student: 12 ECTS (1 ECTS = approx. 30 Hours)

Part II

Analysis

Chapter 3

Problem Analysis

This chapter outlines the topics that needed to be researched before the implementation of the Substitution Stepper could start. It describes the findings and problems that were encountered and describes how they could be solved.

3.1 Understanding Core

Before Haskell code is transformed into byte- or machine code, identifiers are renamed to unique identifiers, types are checked and the code is desugared and simplified. These operations result in Core code, which is the code that the Substitution Stepper is built to step. This section highlights the advantages and drawbacks of stepping Core code, shows the differences between Haskell code and Core code, and explains what the different expressions mean in Core.

3.1.1 Why Core?

Core has many advantages over Haskell code, as it is already desugared (i.e. some high-level features are removed) and the function identifiers have been made unique. Since many Haskell features are simplified it is a lot easier to implement the stepping. For example, the effort that needs to be put in to find the right class instance for a datatype is reduced. Another example that is simplified is infix operators and functions. These are translated to prefix function notation, which makes it easier to step since all of the function applications and operations are uniform. Function bindings are translated into lambdas, which are easier to handle, and multi-argument lambdas are translated into nested single-argument lambdas which are also simpler to apply. These are only a few of the simplifications that are made during the desugaring process.

However, these simplifications can also be drawbacks, as they make the pretty printing of the Core expressions back into regular Haskell expressions quite a bit more challenging.

3.1.2 Core Expressions

This subsection contains information about the structure of the Core `Expr` datatype, which is used to represent Core code.

```
data Expr b
= Var   Id
| Lit   Literal
| App   (Expr b) (Expr b)
| Lam   b (Expr b)
| Let   (Bind b) (Expr b)
| Case  (Expr b) b Type [Alt b]
| Cast  (Expr b) CoercionR
| Type  Type
| Tick  CoreTickish (Expr b)
| Coercion Coercion
```

Figure 3.1: The Expr datatype

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

type Alt = (AltCon, [Var], Expr)
```

Figure 3.2: The datatype used for different options for Case Expressions

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

Figure 3.3: The datatype used for let bindings

Var A `Var` corresponds to a fully evaluated expression and can - by itself - not be further reduced. An example of this is an argument-free constructor of a datatype, like `Empty` for the datatype `data List a = Cons a (List a) | Empty`

Lit A `Lit` corresponds to a literal value, like a number or a string.

App An `App` corresponds to an application. An application can be used to apply an argument to a constructor, like `App (App (Var Cons) (Lit 16)) Empty`, which corresponds to `[16]` or to apply an argument to a lambda, like `Lam n ((Var n))`, which corresponds to the `id` function.

Lam A `Lam` corresponds to a lambda. The first argument to the `Lam` constructor is the identifier of the variable that should be replaced inside the body and the second argument is the body of the lambda. Lambdas in Core can only have one argument, which can be seen when looking at the constructor. It only contains one binding that will be replaced inside of the expression. Because of this, multi-argument lambdas are split up into multiple nested lambdas during the desugaring process.

Let A `Let` expression can be either a recursive or a non-recursive `Let` binding. The first argument to the constructor is said binding, the second is the subterm in which this binding is defined.

Case The `Case` constructor is used for case expressions. The scrutinee is the first argument to the constructor, and the fourth argument is a list of alternatives that can be selected depending on the scrutinee. The second and third arguments are not important for the Substitution Stepper. Similar to the lambdas,`Case` expressions are also simplified in Core. They only look at the top-level constructor in an expression. So a Haskell case expression that looks like the following: `case someList of Cons a (Cons b _) -> a + b; ...` will be translated to something that looks more like this: `case someList of Cons a nestedList -> (case nestedList of Cons b _ -> a + b; ...;); ...;`

Cast The `Cast` constructor is used to cast certain expressions to another type, but this is not relevant for the Substitution Stepper.

Type The `Type` constructor is used to add types to polymorphic data structures or case statements. The Substitution Stepper ignores types, however, so they are also not relevant.

Tick `Ticks` can be used to annotate the source code and they are relevant for this project since they are responsible for the coloration and highlighting of expressions.

Coercion The `Coerction` constructor is also not relevant for the Substitution Stepper.

AltCon `AltCons` are used to indicate the type of alternative for a `Case` expression. Depending on the `AltCon` different kinds of expressions will be matched. The `DataCon` or `Literal` contained in the `DataAlt` or `LitAlt` respectively contain the constructor or literal that is being matched against. Thus by comparing the content of the `AltCon` and the scrutinee, one can see if the alternative matches or not.

DataAlt A `DataAlt` is used for alternatives matching a datatype constructor. For example `DataAlt Empty` could be used to match the empty list in a case expression like `case Empty of Empty -> True; Cons _ _ False;`. In this example, both of the alternatives should be `DataAlts` since the scrutinee is a datatype constructed via a constructor.

LitAlt A `LitAlt` is used for alternatives matching a literal. This is used for scrutinees that are numbers or strings, like `case someNumber of 42 -> True; _ False;`.

DEFAULT The `DEFAULT AltCon` is used to match anything. If the `DEFAULT AltCon` is reached, it matches regardless of what the scrutinee is and what comes after it.

Alt The `Alt` is a datatype that contains one single alternative for the `Case` expression. It consists of the `AltCon`, indicating what kind of match is performed and containing the value that the scrutinee should match. Additionally, in the case of a `DataAlt`, it also contains a list of variables that match the arguments to the constructor. These can be used in the expression that the scrutinee is being replaced with. An example of that would be: `case someList of Empty -> 0; Cons n _ -> n;`. In this example, the variable `n` would appear in the previously mentioned list and will be used to determine the result. The last element contained in the `Alt` datatype is the resulting expression, in case that particular alternative is selected.

3.1.3 Translating Core back to Haskell

While the simplicity of Core is a big advantage during the stepping process, it can also be somewhat of an inconvenience, especially if the goal is to make Core look like regular Haskell. This becomes especially apparent when dealing with `Case` expressions that contain additional, nested `Case` expressions in the alternatives, or when dealing with nested lambdas that each have only one value. The nesting is not only not very aesthetically pleasing but it also makes the code harder to read.

To fix this, the module responsible for pretty printing treats expressions like these in a special way.

Multi-Argument Lambdas

Fixing the nested single-step lambdas is pretty simple and does not require much code, as can be seen in Figure 3.4. If a lambda is nested inside another lambda, the binding variable is displayed, followed by a recursive call to `prettyLam` which prints the rest of the lambda. Once the last nested lambda is reached, the arrow is displayed followed by the expression in the last lambda. In the end, the backslash needs to be prepended to the lambda, which is done by the function that called `prettyLam`.

```haskell
prettyLam :: (Pretty a, Show a) => Expr a -> Doc AnsiStyle
prettyLam (Lam b (Lam bn ex)) = pretty b <+> prettyLam (Lam bn ex)
prettyLam (Lam b ex) = pretty b <+> pretty "->" <+> prettyEx False ex
prettyLam _ = undefined
```

Figure 3.4: The function displaying nested lambdas as one

Case Applications

Fixing the layout of the `Case` expressions inside of alternatives is a bit trickier. It is achieved by using the prettyAlt function:

```
prettyAlt :: (Pretty a, Show a) => Doc AnsiStyle -> Alt a -> Doc AnsiStyle
prettyAlt prefix (Alt (DataAlt dc) bs (Case _ _ _ as))
    | null bs = vsep $
        map (prettyAlt (prefix <+> pretty dc)) as
    | otherwise = vsep $
        map (prettyAlt (prefix <+> pretty dc <+>
        (hsep . map pretty . filter (isUpper . head . show))
        bs)) as
prettyAlt prefix (Alt (DataAlt dc) bs ex)
    | null bs   =
        hsep (prefix <+> pretty dc : [pretty "->", prettyEx False ex]) <> pretty ";"
    | otherwise =
        parens (prefix <+> pretty dc <+> hsep (map pretty bs))
        <+> pretty "->" <+> prettyEx False ex
prettyAlt prefix (Alt (LitAlt lit) [] ex) =
    prefix <+> viaShow lit <+> pretty "->" <+> (group . prettyEx False) ex <> pretty ";"
prettyAlt prefix (Alt DEFAULT [] ex) =
    prefix <+> pretty "_ ->" <+> (group . prettyEx False) ex <> pretty ";"
...
```

Figure 3.5: The function responsible for pretty printing alternatives

The function `prettyAlt` in Figure 3.5 checks if the expression that it contains is a `Case` expression. If that is the case, it pretty-prints the constructors contained in the alternative, before calling itself recursively on the alternatives of the nested case. This concatenates all the constructors of the nested alternatives and allows them to look like one alternative for the top-level case expression. Otherwise, it prints the variables that are being matched against, the arrow separating the LHS from the RHS and pretty prints the RHS. In the case of a `LitAlt`, the literal is printed before the arrow and the RHS. In the case of `DEFAULT`, the underscore is printed before the arrow and the RHS.

The following Core `Case` from Figure 3.6 expression can be taken as an example:

First, the `DataCon` from the outermost alternative is printed out, which corresponds to `Cons`. Since the alternative contains another `Case` expression, the `map` function prepends the `Cons` before every option in the nested `Case`. The `prettyAlt` function is called recursively and in this execution the `a` that was passed as argument is prepended before the alternatives in the nested case statement. Putting it all together, the three alternatives are combined into one single case expression, with the options being `Emtpy -> 0`, `Cons a (Cons b _) -> 2`, and `Cons a Empty -> 1`. The resulting options can

```
case someList of
    Empty -> 0
    Cons a restList -> case restList of
                        Cons b \_ -> 2
                        Empty -> 1
```

Figure 3.6: Example of a case expression nested in an alternative.

then be displayed with the outermost case expression, which finally results in the more Haskell-like case expression seen in Figure 3.7:

```
case someList of
   Empty -> 0
   Cons a (Cons b _) -> 2
   Cons a Empty -> 1
```

Figure 3.7: The pretty-printed version of the previous example.

However, in order to stay consistent in both the displaying and derivation of case expressions, the Substitution stepper also derives these case expressions in one step, instead of first deriving the outer and then the inner one. Since it is both necessary for the pretty representation of the case expressions, and probably also intuitive for the user, I felt like this adjustment was justified.

Infix Operators

In Core, function- and operator applications are changed to prefix form. For most people this is unusual and it can also make it hard to read. Due to this, the Substitution Stepper is built to display operators in infix form. Since operators can be distinguished easily from functions due to the fact that function names start with an alphabetic character, it is not too hard to fix this. If an operator is recognized, the pretty printing sequence is flipped for the operator and the first argument, so `+ a b` becomes `a + b` by flipping the argument `a` the operator `+`. This can be seen by looking at the interaction of the following four functions:

3.2 Reduction Types

After studying Core and the previous implementation it turned out that in order to step Core, roughly four types of reductions need to be implemented. In this section, they are listed and explained.

```haskell
prettyEx' :: (Pretty a, Show a) => Bool -> Expr a -> Doc AnsiStyle
prettyEx' needsParens (App a b)
    | isOperation a = if needsParens
        then parens $ (prettyEx False . flipApp) a <+> prettyEx False b
        else (prettyEx False . flipApp) a <+> prettyEx False b
    | ...
...
```

Figure 3.8: The pretty-printing function decides how to print an application based on the result of `isOperation`

```haskell
isOperation :: Expr a -> Bool
isOperation (App a _) = isOperator a
isOperation _ = False

isOperator :: Expr a -> Bool
isOperator (Var x) = case show x of
    '$':'c':s -> not . isAlpha $ head s
    _ -> not . isAlpha $ head (show x)
isOperator (Tick _ x) = isOperator x
isOperator _ = False
```

Figure 3.9: Helper functions, checking if an application is an operator.

```haskell
flipApp :: Expr a -> Expr a
flipApp (App a b) = App b a
flipApp _ = error "flipApp can only be called on Apps!"
```

Figure 3.10: Helper function that is responsible for flipping the first argument and the operator.

3.2.1 Delta Reductions

Delta reductions are one of the simpler types of reductions. In the case of Core, a delta reduction corresponds to replacing a function name occurring somewhere in the term with its definition.

Given the following term and definitions, a delta reduction could look like this:

It can be seen that the function name `fib` has been replaced with its definition, which in this case is a lambda taking one argument. The body of the lambda contains the logic defined in Figure 3.11.

These delta reductions are created based on derivation rules, which are derived from the top-level bindings found in the Haskell module.

```
data Nat = Z | S Nat

fib :: Nat -> Nat -- Fibonacci
fib Z = S Z
fib (S Z) = S Z
fib (S (S nat)) = fib (S nat) + fib nat
```

Figure 3.11: Definition of Nat and the function fib.

```
example :: Nat
example = fib (S (S Z))
```

Figure 3.12: Example term

```
example = (\n -> case n of
       Z -> S Z
       S Z -> S Z
       S (S nat) -> fib (S nat) + fib nat) S (S Z)
```

Figure 3.13: The resulting term when applying a delta reduction on the previously shown example term.

3.2.2 Beta Reductions

Beta reductions correspond to the replacement of variables with another expression. Beta reductions consist of a subterm and a binding variable, which indicates which variable should be replaced in the contained term. In Core, beta reductions are mostly depicted as lambdas, which take the form (`Lam b ex`), where the `b` defines the binding variable and `ex` contains the subterm in which the replacement should happen.

Beta reductions are mainly used for two purposes. The first one is to apply the arguments to a subexpression, like applying an argument to a function definition. For this purpose, lambdas are used.

The second main purpose is in case expressions. For example, case expressions that capture a variable via the matching of constructors often use this variable again on the RHS of the alternative. Thus this captured variable replaces a certain identifier within the RHS of the application. For this purpose, beta reductions are used as well.

Continuing the example from Figure 3.11 we can apply the lambda to the argument `S (S Z)`. Doing this beta reduction will result in the following expression:

The lambda has now disappeared and instead of the `n` in the `case` expression the argument `S (S Z)` is contained there now. We can tell by comparing the alternatives to the scrutinee, that the third alternative has to be the one that is selected. The first two alternatives don't match, while the third one matches, if we substitute `nat` with Z.

```
example = case S (S Z) of
    Z -> S Z
    S Z -> S Z
    S (S nat) -> fib (S nat) + fib nat
```

Figure 3.14: After applying a beta reduction in the form of a lambda.

However, just replacing the scrutinee with `fib (S nat) + fib nat` won't do the job, since the `nat` on the LHS occurs in the RHS. This reveals the second application of beta reductions. The variable `nat` needs to be replaced on the RHS with the value that it captured, which in this case is `Z`. This is done by doing a beta reduction as well, yielding the term `fib (S Z) + fib Z`.

3.2.3 Case Applications

Case applications are probably the most complicated type of reductions currently performed by the Substitution Stepper. As seen in the example shown in 3.14, reducing a case expression involves more steps than the other types of reductions do.

The first step consists of pattern-matching the scrutinee with the patterns in the alternatives, until one is found that matches. In some cases, it is fairly easy, for example when matching a literal or the default pattern _, since for literals, it is an easy equality comparison and in the case of _ no comparison at all.

When dealing with custom datatypes, the constructors need to be equality-compared and the arguments to the constructor also need to be compared or captured. As seen in Figure 3.14, the Substitution Stepper needs to match the two leading `S` constructors, and then match the `nat` variable with `Z` and capture the value in the variable. This can become even more complex for constructors that have multiple arguments since more variables need to be captured.

The last step performed by the case application is the previously described beta reduction that replaces the variables on the RHS with the captured values.

3.2.4 Let Applications

The last of the four reduction types can be split up into two responsibilities concerned with handling `Let` expressions.

Applying a Let Binding

The first one is the replacement of `Vars` with the definition of previously defined let bindings. This also comes in two flavors, recursive and non-recursive let bindings.

Non-Recursive Let Bindings An example of non-recursive bindings can be seen in Figure 3.15.

```
let y = Z
let x = S Z
case x of
    Z -> 0
    S _ -> do
        let z = 1
        z
```

Figure 3.15: An example for a non-recursive let expression

It is clear that the case expression cannot be reduced yet, as it is not in WHNF and the scrutinee `x` cannot be matched against the options. First, the let expression needs to be applied to the scrutinee `x`. The application of a let expression can be seen as a delta reduction. It could also be seen as a beta reduction in the case of non-recursive bindings, but as soon as recursive bindings are considered, problems could arise when treating it as a simple beta reduction.

So when trying to apply a let binding to a specific subterm in the Substitution Stepper, all let bindings that are ancestors of the subterm are collected. From the binding, reduction rules are extracted analogous to the reduction rules extracted from the top-level bindings at the start. The Substitution Stepper then tries to see which rule is applicable and as soon as one is found it is applied to the subterm, just like a delta reduction.

While 3.15 is a bit artificial, it explains very well what happens with non-recursive let bindings when they are applied. When trying to reduce the scrutinee `x`, the Substitution Stepper collects the ancestor let bindings `y = Z` and `x = S Z`, while ignoring `z = 1`, as it is not an ancestor of the scrutinee.

Then, the Substitution Stepper tries to do a delta reduction with a rule that replaces a variable with value `y` with `Z`, which fails, since the scrutinee has value `x`, not `y`. In the next step, it tries to do a delta reduction with a rule that replaces a variable with value `x` with `S Z`. This delta-reduction succeeds and yields the term shown in 3.16

```
let y = Z
let x = S Z
case S Z of
    Z -> 0
    S _ -> do
        let z = 1
        z
```

Figure 3.16: The let has been successfully applied to `x`

Recursive Let Bindings An example of recursive bindings can be seen in Figure 3.17.

```
let inf = S inf -- infinity
case inf of
    Z -> 0
    S Z -> 1
    S (S Z) -> 2
    S (S (S _)) -> 3
```

Figure 3.17: An example for a recursive let binding

Recursive let bindings can be very similar to non-recursive ones, but in the case of a `case` expression like this, they have to be treated quite differently. In Figure 3.17, the binding needs to be applied to the scrutinee three times before it can be matched.

```
let inf = S inf -- infinity
case S (S (S inf)) of
    Z -> 0
    S Z -> 1
    S (S Z) -> 2
    S (S (S _)) -> 3
```

Figure 3.18: The same term after applying the binding three times.

When looking at Figure 3.18 it should be clear now, that the last alternative is the only one that matches the scrutinee, thus the term results in the value `3`.

The workings behind the scenes are not obvious when looking at the pretty, Haskell-like, representation of the Core case expressions. Thus it might be better to consider Figure 3.19 to understand what is going on here.

```
let inf = S inf -- infinity
case inf of
    Z -> 0
    S a -> case a of
        Z -> 1
        S b -> case b of
            Z -> 2
            S _ -> 3
```

Figure 3.19: The same example for a recursive binding but this time displayed with the desugared syntax.

When looking at Figure 3.19 it becomes more clear why the binding needs to be applied exactly three times. The first time it is applied to match `S a`. `S inf` is matched with `S a`, capturing the value `inf` in `a`.

```
let inf = S inf -- infinity
case inf of
    Z -> 1
    S b -> case b of
        Z -> 2
        S _ -> 3
```

Figure 3.20: The same example after applying the binding once and reducing the outermost case expression

The same step can be repeated, replacing the `inf` with `S inf` and matching it against `S b`, which in turn captures the `inf` in `b` again.

```
let inf = S inf -- infinity
case inf of
    Z -> 2
    S _ -> 3
```

Figure 3.21: The same example after applying the binding twice and reducing the two outermost case expressions

By repeating the step one more time `S inf` is matched with `S _`, yielding the result `3`.

This example also shows one of the advantages of desugaring the Haskell code. It is a lot easier to handle these nested case expressions as opposed to the first example seen in 3.17.

The reason why the application of let bindings does not work with a simple beta reduction - at least with the implementation of the Substitution Stepper - is that due to limitations with the underlying subterm datatype, the beta reduction is done recursively. If a recursive binding is inserted during the beta reduction, the beta reduction will call itself again and replace the previously inserted recursive binding again. This will go on and on and thus create an infinite recursive loop.

Removing a Let Binding

The second responsibility associated with let bindings is the removal of let bindings that are no longer needed. There are different ways to handle this.

For the Substitution Stepper, I have chosen a very simple approach that removes a let expression if the term that it contains is simply a `Var` and the value of the `Var` is not equal to the identifier of the binding or a `Lit`. This allows the Substitution Stepper to be sure that the let binding cannot be applied to anything, thus it can be removed.

Taking figure 3.16 as an example and reducing the case expression yields the following result:

```
let y = Z
let x = S Z
let z = 1
z
```

Figure 3.22: The example from 3.16 where the case expression has been reduced

The binding for z can be applied to `z`, which yields the following result:

```
let y = Z
let x = S Z
let z = 1
1
```

Figure 3.23: The example after applying the let binding to `z`

Once this term is reached, the binding `z = 1` cannot be applied to anything anymore as the expression contained is the literal `1`. When this is the case, the let binding for z can be removed, leaving only the bindings for x and y. The same process is applied to the bindings for x and y. Neither of them can apply to the literal `1`, thus the binding for x is removed first, followed by the binding for y, leaving behind the final term and result `1`.

Chapter 4

Requirements

4.1 Functional Requirements

The following table lists the identified functional requirements. The MVP includes FR1, FR3, FR4, FR5, and FR6.

FR	Description
FR 1	The user can specify functions that should not be stepped through and instead derived in a single step.
FR 2	The user can control the derivation flow in such a way, that they can skip the derivation for certain redexes interactively.
FR 3	The user can step through the reduction line by line and control which subterm should be reduced next.
FR 4	The user can make the derivation run through to the end without requiring any interaction.
FR 5	The tool supports a verbose variant, that indicates which function has been applied.
FR 6	The user can import user-defined files and step through functions defined in these files.

4.2 Non-Functional Requirements

This section contains the NFRs, which are prioritized from 1 to 3, where 1 is the highest priority and 3 is the lowest priority.

Nr.	**Description**	**Measurement**	**Prio**
NFR1	The tool is usable for people with little experience in functional programming.	The derivation of a user-defined function takes max. one import command and one step command. The function is defined in one file and depends only on other functions in the same file.	1
NFR2	The tool provides an open API or CLI, to open it up to additional UIs.		3

Part III

Design

Chapter 5

Architecture

This chapter highlights the architecture of the Substitution Stepper as well as the reasoning for some big decisions.

5.1 Overview

The Substitution Stepper consists of multiple logically grouped folders that each have a specific responsibility. The most important folders/modules are shown in Figure 5.1, though not all of them are shown in full detail as that would clutter up the diagram too much and would not provide a lot of value.

5.1.1 Main application

The main application is responsible for initiating the derivation. The module AppConfig is used by the main application to extract the relevant arguments from the command line, based on which the derivation is initiated. The AppConfig module is also responsible for adding default values for arguments that were not passed to the application.

5.1.2 CoreStepper

The CoreStepper module, which was developed outside of the scope of this thesis, takes Haskell source code and compiles it to GHC Core code. All bindings from the stepped module are collected and returned. The binding for the stepped function is fetched as well and passed to the Derivation module via the main application.

5.1.3 Derivation

The module `Derivation.DerivationUtils` is used by the main application to prepare the GHC Core code for stepping. The function `instantiateDerivation` is used to create all the delta reduction rules and to get the term that will be derived. With this done, the respective derivation mode can be initialized. Based on which derivation mode is desired by the user, the control flow is passed on to the `Derivation.Automatic`,

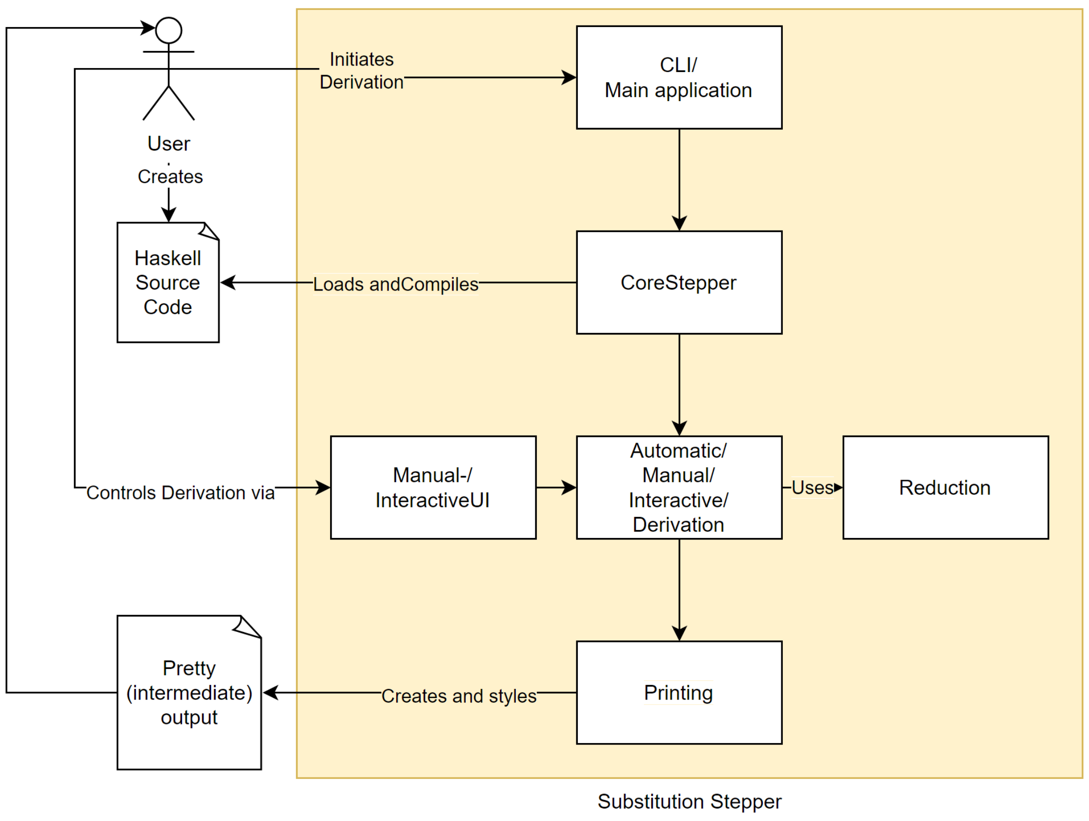


Figure 5.1: Architecture of the Substitution Stepper

`Derivation.Manual` or `Derivation.Interactive` module. The stepping functionality, defined in the `Derivation.Stepper` module, relies on the contents of the `Derivation.Reduction` folder to execute the steps. If the derivation is executed in manual or interactive mode, the `UI.ManualUI` or `UI.InteractiveUI` module is used for the interaction with the user.

5.1.4 Printing

The modules contained in the `Printing` folder are used to display the GHC Core code to the user in a prettier and more Haskell-like way.

5.2 Architectural Decisions

5.2.1 Derivation Logic

For the derivation logic, I have decided to go for a self-written set of derivation rules over the derivation logic that was provided by the `CoreStepper` module. This gives me more control over the rules I want to feature, and makes it easier for me to add functionality that is missing in the `CoreStepper` module like classes and instance functions. However, it comes at the cost of having to put in more time to get a working set of derivation rules. Since I value the flexibility that my self-written implementation gives me, I chose the former option.

5.2.2 Application Type

For the application type, I have decided to go for a command-line application over a GUI application like for example a VSCode extension. The advantage of a command-line application is that it is simpler to implement since there is no need to familiarize myself with the details of how to create a VSCode extension or similar. On the other hand, it takes more effort to make the interaction with the command line intuitive and it takes some thought about how to best display the information so that the user understands what is going on. Since I value the time-saving aspect of the first option, especially because this Thesis is done as individual work, I decided to go for a command-line application.

5.2.3 Sharing

Haskell's lazy evaluation makes use of sharing, so the same expression doesn't have to be reduced multiple times despite using the leftmost outermost strategy. For the Substitution Stepper, however, I have decided to ignore this feature. This removes the need for a heap containing all the evaluated expressions. Since no heap is needed it is also easier to display the stepped terms and probably also less confusing for people who are new to Haskell. Of course, this has the drawback of not being entirely accurate to the derivation strategy used by Haskell. Despite the slight inaccuracy I went for the option without sharing, as I value the simplicity of an approach without sharing more than the accuracy and I think that it has more didactic value for Haskell newbies.

Chapter 6

UI Design

This chapter contains the initial sketches for the design of the derivation UI with a CLI tool.

The derivation uses the following function definitions:

```
data Nat = S Nat | Z

double :: Nat -> Nat
double Z = Z
double (S n) = S (S (double n))

(+) :: Nat -> Nat -> Nat
Z + n = n
(S m) + n = m + (S n)

fib :: Nat -> Nat
fib Z = S Z
fib (S Z) = S Z
fib (S (S n)) = fib (S n) + fib n
```

6.1 Base Variant

The base variant of the tool automatically steps through the derivation without user interaction.

```
> step double (S (S Z))

      S (S (double (S Z)))

      S (S (S (S (double Z))))

      S (S (S (S (Z))))
```

6.2 Verbose Variant

The verbose variant is similar to the base variant. The difference is that the verbose variant includes information about which function was applied during the step. The previous derivation looks like the following in verbose mode:

```
> step -v double (S (S Z))
        = { applying double }
          S (S (double (S Z)))
        = { applying double }
          S (S (S (S (double Z))))
        = { applying double }
          S (S (S (S (Z))))
```

6.3 Hiding Variant

The hiding variant allows the user to define functions, that should be derived in one step, thus hiding the details of the derivation. The hiding variant can be used in combination with either the verbose or base variant. An example of this can be seen in the following derivation (regular on the left side and hiding on the right side):

```
> step -v fib (S (S (S Z)))        | > step -v -s (+) fib (S (S (S Z)))
    ={ applying fib }              |     ={ applying fib }
      fib (S (S Z)) + fib (S Z)    |       fib (S (S Z)) + fib (S Z)
    ={ applying fib }              |     ={ applying fib }
      fib (S Z) + fib Z + fib (S Z) |       fib (S Z) + fib Z + fib (S Z)
    ={ applying fib }              |     ={ applying fib }
      S Z + fib Z + fib (S Z)      |       S Z + fib Z + fib (S Z)
    ={ applying fib }              |     ={ applying fib }
      S Z + S Z + fib (S Z)        |       S Z + S Z + fib (S Z)
    ={ applying (+) }              |     ={ skipping (+) }
      Z + S (S Z) + fib (S Z)      |
    ={ applying (+) }              |
      S (S Z) + fib (S Z)          |       S (S Z) + fib (S Z)
    ={ applying fib }              |     ={ applying fib }
      S (S Z) + S Z                |       S (S Z) + S Z
    ={ applying (+) }              |     ={ skipping (+) }
      S Z + S (S Z)                |
    ={ applying (+) }              |
      Z + S (S (S Z))              |
    ={ applying (+) }              |
      S (S (S Z))                  |       S (S (S Z))
```

As can be seen in the above derivation, the right side saves 3 trivial applications of (+). This can help to make the derivation shorter and hide trivial or uninteresting derivations.

6.4 Manual Variant

The manual variant is similar to the previous variants. The output can be the same as any of the previous variants, the difference here is that the user can step through the derivation manually. So after entering the step command, the first line of the derivation is displayed. The user can then advance the derivation with the enter key or other commands step by step. With this variant, the user can skip parts of the derivation interactively and slowly step through.

In a second step, if possible, the redexes should be highlighted, allowing the user to skip certain redexes and go straight to the interesting derivations. In this example, the redexes are underlined with the tilde symbol and assigned numbers, which can be used in commands like goto, which skips all derivation steps until the specified redex is reached.

```
> step -m -v fib (S (S (S Z)))
          fib (S (S (S Z)))
> enter
        ={ applying fib }
          fib (S (S Z)) + fib (S Z)
          ~~~~~~1~~~~~~   ~~~~2~~~~ -- Multiple redexes are marked
> goto 2                            -- Skip redex 1 and go directly to 2
        ={ goto 2 }
          S (S Z) + fib (S Z)
> enter
        ={ applying fib }
          S (S Z) + S Z
> enter
        ={ applying (+) }
          S Z + S (S Z)
> enter
        ={ applying (+) }
          Z + S (S (S Z))
> enter
        ={ applying (+) }
          S (S (S Z))
```

Interactive commands could include commands like the following:

- enter: advance one line

- goto x: skip ahead to redex x
- step x: advance x lines
- q/quit: cancel stepping/exit the stepper
- continue: continue the stepping until the end is reached

Part IV

Implementation

Chapter 7

Application Implementation

This chapter shows some interesting parts of the implementation of the application.

7.1 Subterm

The subterm datatype is at the heart of the application. It is probably the most important datatype, as it is used all across the application.

7.1.1 Strengths of the Subterm Datatype

The subterm datatype has been designed to always contain a valid subterm. An alternative to the subterm datatype would have been to simply use a term in combination with the position. If the term and position aren't managed together though and it would be possible to change them independently, it would be possible to create term/position combinations that are not valid.

One can take a binary tree as an example that is supposed to keep track of mathematical operations. For example the expression `1 + 2`. The whole term would have position `[]`, the number 1 would have position `[0]` and the number 2 would have position `[1]`. If one were to evaluate the term independently from the positions, One would be left with the term `3`, where only the position `[]` is valid. However, one might still have access to the subterms at position `[0]` and `[1]`. This would pose a problem, as they are not valid positions anymore. If one were to use one of the invalid positions it could cause errors. To prevent these errors, one might be tempted to wrap the whole thing into the Maybe Monad or use checks before using these subterms. This would complicate things and would definitely not be a nice or elegant solution.

A more elegant solution is to wrap the term and position in a datatype, the Subterm datatype in this case. By not letting other Modules access the constructor of the Subterm datatype, independent mutation of the term or position can be prevented.

A Subterm datatype can only be created using certain functions defined in the TermClass module. This ensures that all Subterms are valid. Thus, they can be used without having to worry about any errors arising.

Through the many functions that are defined in the TermClass module, reduction rules can be defined easily using f.ex. the replace function. It takes only two simple functions in order to be able to implement the Subterm datatype. This makes the Subterm datatype very versatile and universally usable.

7.1.2 Weaknesses of the Subetrm Datatype

Though the Subterm datatype has many strengths and is generally a very elegant implementation of the subterm concept, it also comes with a couple of drawbacks. These drawbacks are fairly minor though and could be patched during future development.

The first thing that is a bit inconvenient is that every term structure implementing the Term class has to also implement the Highlightable class, if stepping capabilities are desired. For this purpose, each underlying term structure needs to implement annotations or another concept to be able to store the information needed to highlight/style the terms.

This could be avoided if the Subterm datatype would support annotations out of the box. It might be possible to find a solution, where a dictionary containing the position as key and the annotation(s) as value could be added to a term implementing the Term class. This might add a bit of extra effort that is needed to manage the annotations, but on the other hand, it would come with many benefits. Another approach using the Term class could also be possible, like for example adding a function that determines whether a given subterm is an annotation and skipping all annotations somehow when determining the position of another subterm.

As already mentioned, the annotations - and thus the highlighting - would only need to be implemented once. This would reduce the effort that is needed to add additional languages that could be stepped, as the implementation is already there for any datatype implementing the Term class.

It is also important to note that the implementation of the annotations as it is now is not trivial. Something that needs to be made sure of is that adding an annotation does not mess with the position of a term. If the annotation is added in the middle of the term somewhere, it is possible that it moves the positions of all subterms that come after the annotation. Taking the previous example again and adding a random annotation shows the problem: Going from `1 + 3` to `1 + some_annotation(3)` can move the position of the number `3`. While the other positions stay the same, the position of the number `3` now changes from `[1]` to `[1,0]`, while the position of the annotation now would be `[1]`.

This can cause problems when trying to combine annotations for the same term. A single Subterm value can only point to one single subterm in the term because of how the Subterm datatype is implemented. But often, multiple subterms in the same term need to be highlighted. For example, the diff and the selection in the manual and interactive modes need to be highlighted in the same term. To be able to do that the annotations need to be combined. If one of the annotations changes the position of the actual subterm, the other annotation might be referring to a subterm that does not exist, which could cause errors. This problem is shown in 7.1, where the same term is annotated in two different ways and the annotations contain conflicting positions for the same subterm.

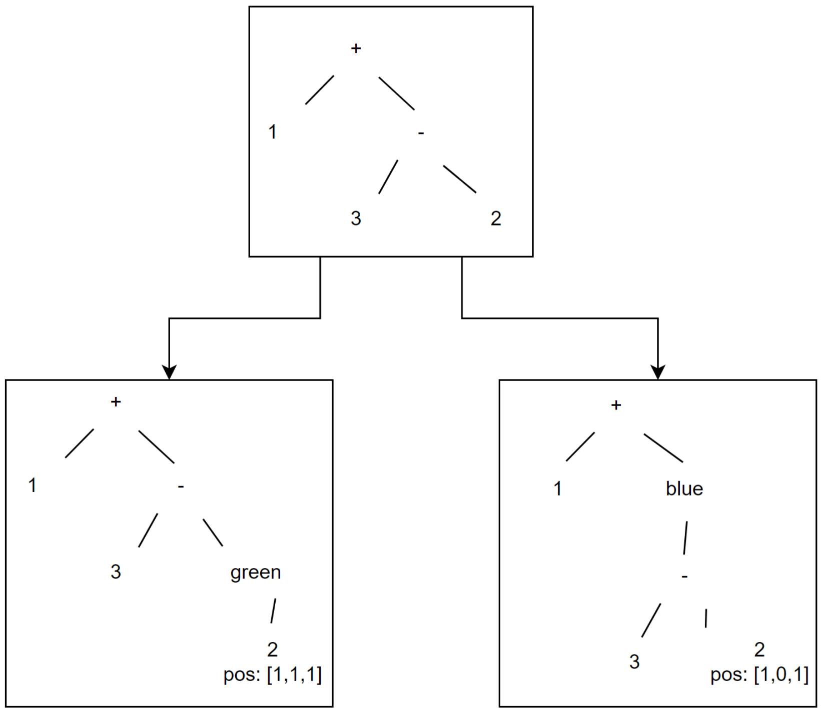


Figure 7.1: The same term can be annotated in two conflicting ways. On the left side, the position of the number 2 is [1,1,1] while on the right side, it is [1,0,1] and the position [1,1] does not exist at all.

For this reason, the annotations must not change the positions of any subterm in the term. Implementing a solution that does not do that might not be obvious initially.

In the Core implementation of the Term class, this is handled by making the annotation appear to be the last child of its parent term, while the children of the annotation appear to be children of the annotation's parent as well. This preserves the position of the annotation's children, while still making the Term class aware of the annotation's existence.

The impact that this change has is shown in 7.2. It is important to note that it might look like with this implementation one cannot tell which child the annotation refers to, but this is only a problem when looking at the subterm in this form. Internally in the Core representation, it is still well-defined which child the annotation refers to.

The second thing that is also a bit inconvenient is that a Subterm value can only point to one subterm in the term. Especially for beta reductions, it would be convenient if one could for example select multiple subterms with a filter condition.

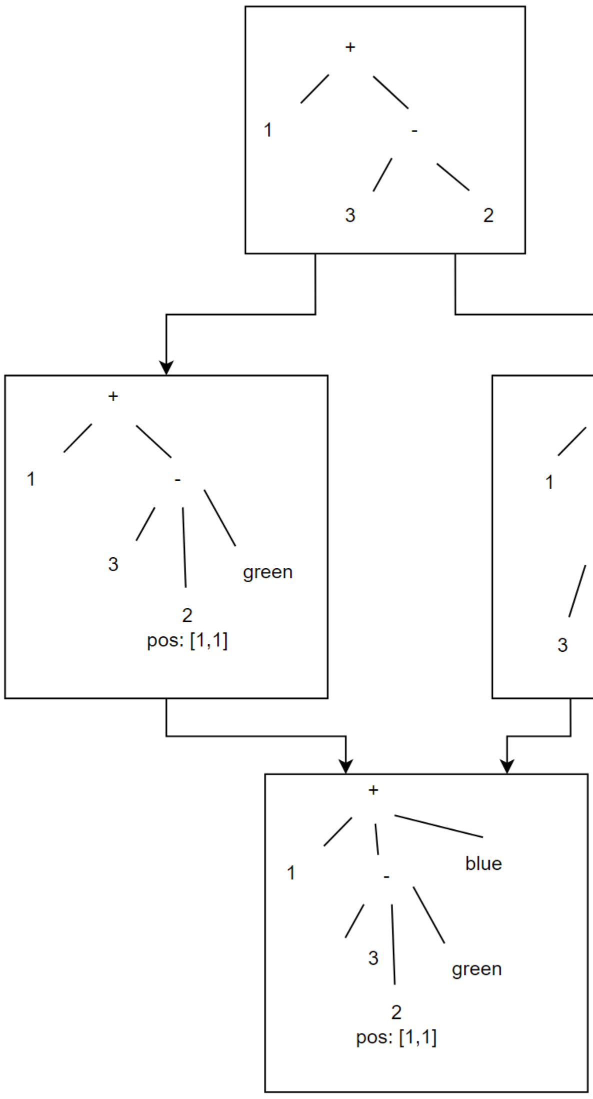


Figure 7.2: The same term, again annotated in two different ways, this time however they are not conflicting and can be combined.

As it is right now, to fully beta-reduce a term, the subterms need to be replaced individually. When attempting a beta reduction, all subterms matching the binding variable are gathered from the term. This yields a list of subterms as a result. To replace all occurrences, the first subterm is selected and the beta reduction is performed on that subterm. The other subterms are left untouched. Then, on the replaced subterm, all subterms matching the binding variable are gathered again and the process is repeated until the binding variable does not occur anymore. This is somewhat cumbersome, but there is no way of replacing multiple subterms in a single term due to the implementation of the Subterm datatype.

A solution for this problem could be, as mentioned before, the capability to select all subterms meeting a certain criterion inside of the subterm and replace them all in one go. This approach could be implemented more efficiently than the current approach.

7.2 Derivation

The Derivation datatype is responsible for keeping track of the history of the derivation.

7.2.1 Initial Approach

Originally, it was planned to be a tree data structure, but it was dropped in favor of a list for simplicity and because making it a tree did not have any benefits. One of the first versions of the Term and Derivation datatypes can be seen in 7.3.

```
data Term
    = Redex String [Term] (Maybe Term)
    | Value String deriving Show

data Derivation = Term [(Derivation, RewriteRule)]
```

Figure 7.3: One of the first drafts for the Derivation Type

The Term datatype turned out to not work well at all and was thus discarded quickly. The Derivation data type on the other hand was considered for a longer time. The idea was to keep track of every possible way to derive a term. So once a branch of the tree was discovered through stepping, the information would be saved even if the user decided to step back. If the user would go back down the same path at a later point, the information would be available already and no computation would be necessary.

The plan was to count all redexes in the current term and to create an entry in the list for each of the redexes, thus creating a tree. And as soon as one of the redexes was selected by the user, the respective entry in the list would be adjusted.

In the end, however, a simpler approach was taken, as the overhead for re-calculating a step is not that big and thus the original datatype did not offer too much of a benefit. In addition to that, the tree datatype would grow exponentially if all paths were discovered and would unnecessarily take up space.

Another reason why it was dropped is that it did not seem like a very likely use case, as a user would probably mostly be interested in just stepping one path and not all of them.

7.2.2 Final Version

The final version of the Derivation datatype is a lot simpler than the original tree structure. It is in essence a stack data structure that only keeps track of the current history and discards the current term if the user decides to step back. Since the current history is most likely the only thing the user cares about, removing all other branches seemed to make a lot of sense. The final version of the Derivation datatype can be seen in 7.4

```
data Derivation t where
    Derive :: Reducible t => t -> [DerivationStep t] -> Derivation t

type DerivationStep t = (RewriteRule t, RuleDescription, Subterm t)
```

Figure 7.4: The final and current version of the Derivation datatype.

The Derivation datatype consists of the original term, which is the first argument to the constructor, and a list of derivation steps, which is the second argument. The reason the starting term is stored separately from the other derivation steps is that it does not require the additional information that the DerivationStep datatype provides, like RewriteRules or RuleDescriptions. Also, the starting term should not specify a certain subterm, which is why it is just of type t and not Subterm t.

7.2.3 DerivationStep

The DerivationStep datatype is closely linked with the Derivation datatype. It is used to specify the rule that has been applied, a description of the rule that has been used, and the resulting term.

The RuleDescription is currently just a simple datatype as it does not contain a lot of information. It is used to display the justifications for the steps. The RuleDescription datatype could be expanded to contain more (meta)information, like line numbers of the function definition, etc. but this was not necessary yet, nor was there a concrete use case for this.

The resulting term is of type Subterm t and the subterm that is pointed to is the subterm that has been substituted when applying the rule. This allows for easy highlighting of the diff. All that needs to be done is to annotate that Subterm with the style for diffs. When the term is printed, the diff will show up via annotation.

7.3 RewriteRule

The RewriteRule datatype is used to create rules for the derivation at runtime. While most rules are defined at compile time already, like beta reductions and case applications which always follow one specific pattern, some rules cannot be determined at compile time. The most important examples of RewriteRules are delta reductions. To create these RewriteRules for delta reductions, the top-level bindings from the module are collected and turned into RewriteRules.

A RewriteRule is just a function that takes a Subterm and returns a Maybe, which might contain the term that results from the application of the rule. The definition of RewriteRule can be seen in 7.5

```
type RewriteRule t = (Subterm t -> Maybe (t, RuleDescription))
```

Figure 7.5: The RewriteRule datatype.

To create a rule for a specific delta reduction, the Substitution Stepper goes through all top-level bindings and creates RewriteRules. If the subterm that the rule is applied to matches the identifier of the binding, the definition of the binding is returned, otherwise, Nothing is returned. This makes sure that the rule for a delta reduction can only be applied to matching identifiers.

Another example of a RewriteRule is a Let binding, which are treated as delta reductions that can only be applied to children of the Let binding.

7.4 Reducible

The Reducible class is the main class when it comes to the stepping of terms. It provides an interface that allows the Substitution Stepper to derive any term that implements this class. To implement the class, two functions are needed.

7.4.1 `reduce`

The `reduce` function is used to derive one step of a term. It takes a set of RewriteRules and a Subterm and tries to derive the specified subterm. If a reduction can be applied to the subterm, the resulting term and RuleDescription are returned, otherwise, Nothing is returned.

In the case of Core, the `reduce` function first tries to do a case application, then a beta reduction, then a delta reduction, then resolving a Let binding, and lastly applying a Let binding to a variable. If none of these work, Nothing is returned, otherwise, the first type of reduction that yields a result is returned.

7.4.2 `isWHNF`

The second function required to implement the Reducible class is `isWHNF`. This function is needed to implement two different types of derivation strategies.

The first type of strategy strictly follows the ordering defined by the strategy. The second type of strategy follows the ordering partially. First, all terms that are not contained within a term that is in WHNF are looked at and the strategy is applied to only these terms. If, however, there are no reducible terms outside of terms that are in WHNF, the terms contained in these WHNF terms are considered too.

This is supposed to prevent infinite loops that could arise when using the leftmost innermost strategy. An example of this is the lambda
`x y -> x + y` with the definiton of `+` being `0 + x = x; (y + 1) + x = y + (x + 1)`.

If the leftmost innermost strategy is followed strictly, the definition of `plus` will be expanded infinitely, resulting in infinite recursion, because the `+` operation is nested more deeply than the application of the lambda.

If, however, terms inside of terms that are in WHNF are not considered, then the definition of `+` could not be expanded any further. This is because the lambda is in WHNF, thus the lambda needs to be applied to the arguments first before the definition of the `+` can be expanded.

This limitation still does not prevent all infinite loops though. If a term does not have a normal form, this approach will not terminate.

7.4.3 Strategies

The ReducibleClass also contains the derivation strategies that can be used by the Substitution Stepper to step through terms. However, in retrospect, this approach does not work well since the derivation strategy cannot be generalized for any given type of term. The strategies that are currently defined order the subterms according to their numbering if the term is traversed infix. This, unfortunately, does not work for Core as it is not infix.

Despite this flaw, the leftmost outermost and call-by-name strategies work reliably for Core.

7.5 Normalizable

The Normalizable class is an extension of the Reducible class. Its use case is to be able to find a sequence of 'atomic' steps, in order to be able to skip the uninteresting parts of a derivation.

Taking Core as an example, in the case of a delta reduction, the Normalizable implementation counts the number of lambdas nested in the function definition, and then tries to find the same amount of applications surrounding the lambda, thus allowing to do delta- and beta reductions, as well as case applications in one single step, hiding the details.

This hides the details of the stepping that most users probably would not want to see either way.

7.6 Highlightable

The Highlightable class is responsible for annotating terms with styles, so that the terms can then be highlighted during pretty printing.

As discussed in the section about Subterms, it might have been more advantageous to leave the annotating to the Subterms. With the current implementation, the Highlightable class has 4 functions that need to be implemented.

7.6.1 `annotateTerm`

The function annotateTerm is responsible for annotating Subterms with styles. In the implementation for Core, source note Ticks are used to add styles to the Expr datatype.

Since the styles implement Show and Read, they can easily be added as source notes and then transformed back into actual styles.

7.6.2 `combineAnnotations`

The function combineAnnotations is used to combine the annotations of two subterms of the same root term. This needs to be done due to the previously mentioned limitation that it is not possible to annotate two different subterms in the same Subterm datatype.

As a workaround, the combineAnnotations function checks whether the root terms of the Subterm datatypes are equal, and if that is the case the annotations contained in the two root terms are combined. This allows the Substitution Stepper to annotate both the selection and the diff in the same term.

7.6.3 `highlight`

The highlight function takes a (potentially) annotated term and pretty-prints it. The reason that this function is needed, is that the regular pretty function does not support colors or any other form of highlighting. As the signature of the pretty function is `pretty :: t -> Doc ann`, the highlight function, however, wants to have the signature `highlight :: t -> Doc AnsiStyle`.

The AnsiStyle annotation provides the coloring/underlining functionality that is used by the Substitution Stepper. So the highlight function needs to call a function different from pretty, that supports these AnsiStyle annotations.

To prevent code duplication, in the case of Core, a function called prettyEx is defined that produces results of type Doc AnsiStyle. The highlight function can then just call the prettyEx function, while the pretty function calls `unAnnotate . prettyEx`, turning the `Doc AnsiStyle` result back to a `Doc ann` result.

This way the pretty printing logic only needs to be implemented once.

7.6.4 `unAnnotateTerm`

The function unAnnotateTerm removes all annotations for a certain term, as the name suggests.

7.7 User Interaction

The user interaction was mainly done using the Haskeline package for writing and reading output/input and the System.Console.ANSI package to manipulate the CLI so that the output looks presentable and not overloaded.

7.7.1 Haskeline CompletionFunc

The Haskeline CompletionFunc is usually used to complete file paths and words via the tab key while inputting on the CLI. The Substitution Stepper uses the CompletionFuncs a bit differently though. Instead of completing any input, the tab key cycles through the available redexes.

The detour via the CompletionFuncs was taken because there is no other easy way to get tab key or arrow key input from the CLI. And since using the tab key is much more convenient and intuitive compared to typing in a command or something, this approach was selected.

A drawback here is that there is no easy key to cycle the redexes in reverse order, as Haskeline only supports the tab key for the completion of words. For reverse-cycling of the redexes the 'backTab' command can be typed in.

7.7.2 Removal of Old Prompts

To remove clutter on the UI, some commands remove the previous prompt before printing the new one.

A very good example of this is the tab command, which changes the selected redex.

Normally, the same term would be printed out under the previous one, with the only difference being the coloring of the selected subterm. Since this can clutter up the UI when changing the selection a lot, the functionality to remove the previous prompt was introduced.

Due to this it now looks like the green-colored selection jumps around in the same term, giving the illusion of a UI that can update itself. In the background, a certain amount of lines are deleted on the CLI and re-printed with a different coloring.

The same approach was also chosen when a command is entered, that is not valid or when error messages are printed. This avoids the constant moving of the CLI when there are no changes in the term itself.

7.7.3 Printing under the Cursor

Some commands that can be used when deriving manually produce output. This output can be either an error message or just a regular 'success' message, that conveys information to the user. To keep the history of terms above the cursor stable and uninfluenced by these messages, the messages are printed under the cursor.

This also greatly simplifies the erasing of the messages. While the prompts can be easily erased, even if they are above the cursor, since their length is known to the application, the length of the messages is not known. Thus there is no easy way to delete them if they are written above the cursor.

If they are written below the cursor, however, the application can simply clear the CLI until the end of the screen is reached. A solution to this is discussed in section 12.3.4. Unfortunately, there was not enough time to implement the suggested changes properly.

To be able to print under the cursor, the message-outputting functions print a newline, moving the cursor down one line.

Before the message is printed, the number of lines that the message will occupy is calculated. Then the message is then printed. After this, the cursor is moved back up to where it was originally. This gives the illusion of messages being printed under the cursor.

From the number of lines occupied by the message, the application can calculate how many lines the cursor needs to be moved up.

7.7.4 Adaptive Line Width

To be able to take advantage of the full terminal, the UIUtils module provides a function that creates a LayoutOpts value, which is used by the pretty printer.

The LayoutOpts value contains the length of a line in the CLI. This allows the pretty printer to format the terms and messages in a way that allows them to fill up the entire screen. Line breaks are chosen smartly by the prettyprinter, so that words should not spill over to the next line, avoiding ugly line breaks.

The function used to get the width of the CLI is not guaranteed to work on any terminal, as it is not part of the ANSI standard functionality.

In case the width of the CLI cannot be retrieved, 80 characters per line is chosen as the default value, as most terminals should be at least this width.

7.8 Commands

For the user interaction, the Command datatype is used. The list of commands is easily extensible, thus allowing for the easy addition of new commands. The Command datatype can be seen in figure 7.6

Each command has access to the DerivationState and can thus modify the derivation freely. Commands cannot perform IO actions, but they can notify the user about errors and pass messages via the CommandResult.

If the CommandResult is `Left someDoc`, it means that an error has occurred and the message is displayed in red. If an error occurs, the DerivationState should not be changed.

If the CommandResult is `Right maybeDoc`, then the command was successful and depending on the content of `maybeDoc` there can be a message that should be conveyed to the user.

```
type CommandResult = Either (Doc AnsiStyle) (Maybe (Doc AnsiStyle))

data Command t = Command
    { name               :: String
    , argsDescription    :: String
    , usageDescription   :: String
    , command            :: [String] -> State (DerivationState t) CommandResult
    , continueAfter      :: Bool
    , replaceLine        :: Bool
    , printDerivation    :: Bool
    }
```

Figure 7.6: The Command datatype.

While the Commands can also not read input via IO actions, the user can pass arguments to the Command via a list of strings. It is the Command's responsibility to parse/read the arguments correctly and without producing an error that terminates the application.

The user is able to call commands by typing in their names, followed by a space-delimited list of arguments if needed.

The remaining fields in the Command datatype give the Command control over what happens after the command is executed.

For example, the continueAfter flag will decide if the derivation continues after the command is executed. If it is set to False, the derivation will stop. This is used for the quit command, which simply aborts the derivation.

The replaceLine flag decides whether the previous prompt should be removed before the new one is printed, as described in the previous section about the UI. The tab command is one of the examples that make use of that flag.

The printDerivation flag decides whether the derivation should be printed out. The flag only takes effect if the derivation stops after the command is executed. An example of a Command that uses this flag is the continue command. Since the command continues derivation automatically, the user cannot see what is happening, thus it needs to be printed at the end so the user can see it.

All the current commands can be executed using a combination of these flags, however, if new commands are added the list of flags might have to be extended.

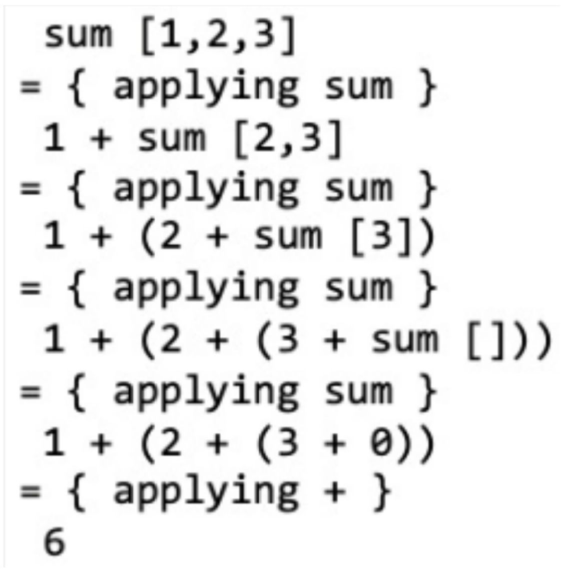
The reasoning behind implementing user interaction this way, is, as mentioned before, to be able to easily add new commands. The inspiration for this was taken from the Command software design pattern.

Chapter 8

Examples from the Task Description

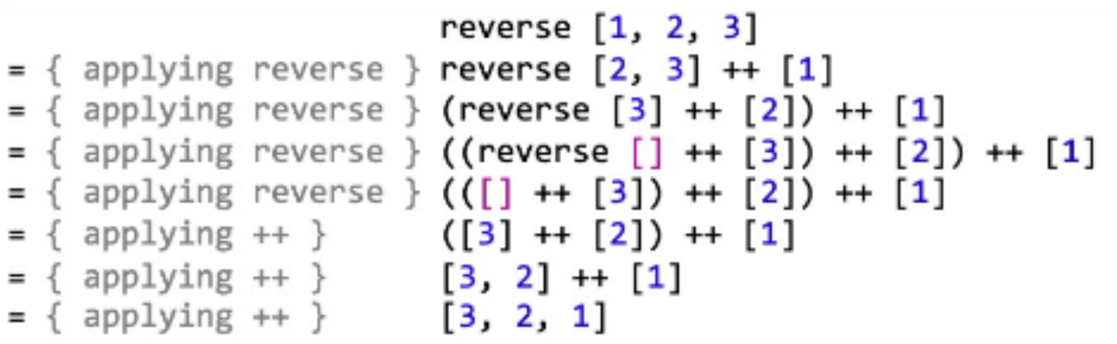
The examples showing how the Substitution Stepper steps some of the functions shown in the task description. Due to the length of the examples, they can be found in the appendix in Chapter 13

The examples that are shown are similar to the following examples from the task description, however, some adjustments had to be made due to the limitations of the stepper.



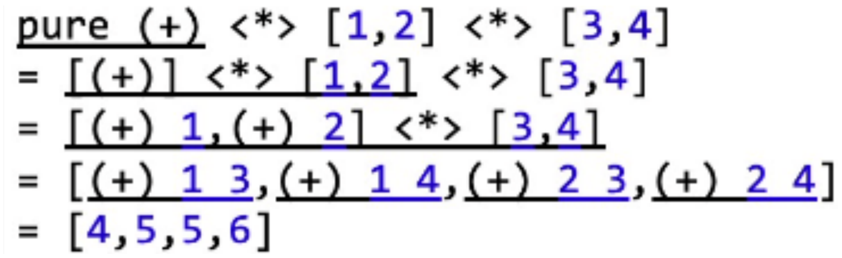
```
 sum [1,2,3]
= { applying sum }
 1 + sum [2,3]
= { applying sum }
 1 + (2 + sum [3])
= { applying sum }
 1 + (2 + (3 + sum []))
= { applying sum }
 1 + (2 + (3 + 0))
= { applying + }
 6
```

Figure 8.1: Example 1



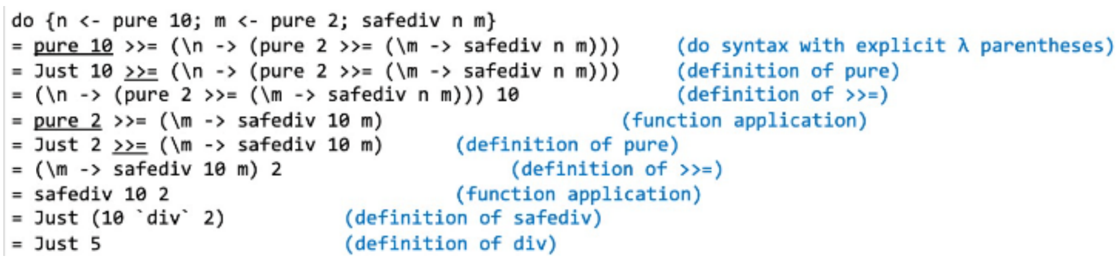
```
                          reverse [1, 2, 3]
= { applying reverse }    reverse [2, 3] ++ [1]
= { applying reverse }    (reverse [3] ++ [2]) ++ [1]
= { applying reverse }    ((reverse [] ++ [3]) ++ [2]) ++ [1]
= { applying reverse }    (([] ++ [3]) ++ [2]) ++ [1]
= { applying ++ }         ([3] ++ [2])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

Figure 8.2: Example 2



```
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 8.3: Example 3



```
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 8.4: Example 4

Chapter 9

Extensibility of the Substitution Stepper

A secondary goal when creating the Substitution Stepper was to create a stepping application that can be used for more than just Haskell. For this, the stepping functionality was kept as generic as possible. This chapter describes how one could approach the addition of a new language that should be stepped.

9.1 Implementing the Show/Pretty Class

The Substitution Stepper requires the terms to be pretty printable via the `pretty` function defined in the Prettyprinter package, as well as transformable into strings via show.

The requirement for Show could be lifted, but that would require removing the showTerm command. That should however not have too big of an impact on the usability.

9.2 Implementing the Term Class

The first step when adding a new language to step, should be to implement the Term Class, as it is used everywhere across the code.

When adding a structure that keeps track of annotations, it is important that the annotation does not change the position of the contained terms. Other than that there is not much to it and implementing the Term class should be quite simple.

9.3 Implementing the Reducible Class

The Reducible class needs to be implemented, in order to be able to step the term.

9.4 Optional: Implementing the Normalizable Class

The Normalizable class can be useful if the derivation of a term should hide some steps that are not really interesting. But it is not necessary to do that. If this functionality is not desired, it might be best to implement `getNormalizableTerm`, but rather than performing some changes to the term, it can be wrapped in a Just and returned. Alternatively, the getNormalizableTerm function can also just always return Nothing.

9.5 Optional: Implementing Highlightable

The Highlightable is also not essential to the derivation. However, it is used to color the selection and diff during the manual derivation. If only the automatic derivation functionality is desired, then the Highlightable class can be implemented with bogus functions, like `annotateTerm = _ -> id`, `combineAnnotations = _ -> id`, `highlight = pretty`, and `unAnnotateTerm = id`.

9.6 Setting up the Derivation

Once the above classes have been implemented, the derivation can be set up and then executed.

At first, if any type of delta reduction is supported, the rules need to be created. The rules should have the following type: `type RewriteRule t = (Subterm t -> Maybe (t, RuleDescription))`.

Once all rules are created, the functions that execute the derivation can be called. These function are: `doAutomaticDerivation`, `doManualDerivation`, and `doInteractiveDerivation`.

When all of this is done, the new language is ready to be stepped.

9.7 Conclusion

It is fairly simple to add different languages to the Substitution Stepper. The most challenging task is probably to implement the Reducible class and maybe create the RewriteRules, which depends heavily on the language added. The other parts are quite simple.

Since there is not as much code to be written for the addition of a new language, a lot of time can be saved if a similar project is attempted again.

Thus, I would say that the secondary goal of being extendible has been reached.

Chapter 10

Quality Measures

This chapter describes the various quality measures that were put in place to make sure that the code is as clean as possible and working as expected.

10.1 Coding Guidelines

- General
  - Code needs to be committed frequently with descriptive commit messages.
  - Side effects should be avoided whenever possible.
  - Comments may only be used if they are helpful.
  - Global variables should be avoided if possible.
  - Exceptions are preferred over error codes.
  - Nesting should not be deeper than 2 levels.
  - No duplicate code. (DRY)
  - Functions should be kept short (less than 11 lines).
- Formatting
  - Line length should not exceed 120 characters.

10.2 Test Concept

Due to the limited amount of time available, a proper test concept could not be created. However, to ensure that the most important functionalities (i.e. the stepping functionalities) work, some tests have been created.

The test log can be seen in 10.1. For the testing, the hspec package has been used.

10.2.1 Unit Tests

The `caseSpec`, `deltaSpec`, and `betaSpec` suites perform unit tests on the respective reduction functions. These tests make sure that the reduction functions operate as expected.

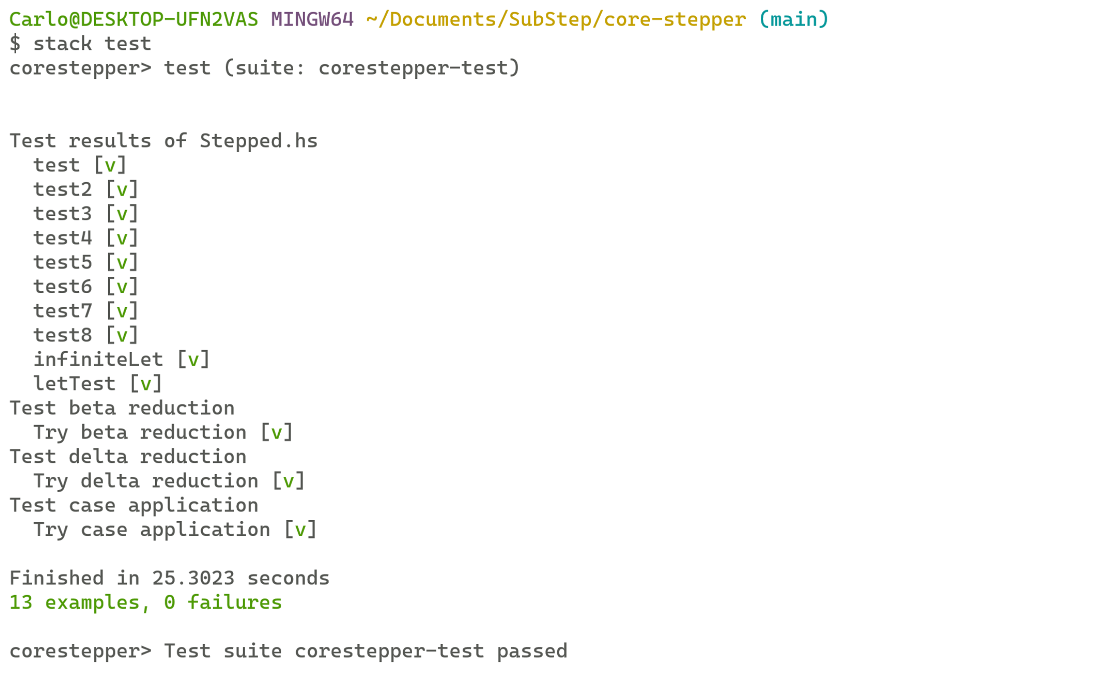
10.2.2 System Tests

The `resultSpec` suite contains some system tests that compare the output of some test functions to the expected output. These tests make sure that the reductions happen as expected.

10.2.3 Difficulties during Testing

During the testing process, some difficulties were encountered. The main problem is that the identifiers for `Var`s are not always the same. They can change between the execution of tests, which makes the tests not repeatable.

To fix this issue, instead of doing a full comparison, only the parts between the `Var`s are compared. The whole term is still tested, but it is broken up into smaller comparisons instead of one large comparison comparing the whole term.



```
Carlo@DESKTOP-UFN2VAS MINGW64 ~/Documents/SubStep/core-stepper (main)
$ stack test
corestepper> test (suite: corestepper-test)


Test results of Stepped.hs
  test [v]
  test2 [v]
  test3 [v]
  test4 [v]
  test5 [v]
  test6 [v]
  test7 [v]
  test8 [v]
  infiniteLet [v]
  letTest [v]
Test beta reduction
  Try beta reduction [v]
Test delta reduction
  Try delta reduction [v]
Test case application
  Try case application [v]

Finished in 25.3023 seconds
13 examples, 0 failures

corestepper> Test suite corestepper-test passed
```

Figure 10.1: The test log

Part V

Discussion

Chapter 11

Results

In this chapter, the final result is analyzed and the requirements are checked.

11.1 Functional Requirements

11.1.1 FR 1

The user can specify functions that should not be stepped through and instead derived in a single step.

The first requirement has not been met, as I have not found a possible way to reliably implement this functionality. This requirement is closely related to normalization, as the point is to skip all applications of a certain function until it does not contain that function anymore.

An example that was created as a possible use case for this requirement was the + operator.

```
data Nat = S Nat | Z

(+) :: Nat -> Nat -> Nat
Z + n = n
S m + n = m + S n

ex = (S (S (S Z))) + (S Z)
```

Figure 11.1: The starting term for the example of FR 1

The idea here was that since the + operator is very trivial and boring to step through due to the big amount of recursion just to add two numbers together, that it should be skipped instead and directly return the result `(S (S (S (S Z)))` in this case).

However, due to the recursive nature of the definition of the + operator, I found this to be not possible to do nicely.

While it would work for this particular example, it would not work for a term that does not have a normal form, since then the `+` operator could end up being expanded infinitely.

Looking at the term `pure + <*> [S Z] [S (S Z)]` shows the problem in this case. If the function `pure` is reduced, the Substitution Stepper will eventually get to the `+` inside of it and try to skip it. But because it is not applied to enough arguments, it does not have a normal form. Instead, the `+` will be expanded over and over again, causing an endless loop (see figure 11.2).

```
pure +

[+]

[
    \left right -> case left of
        Z -> right
        S m -> m + S right
]

[
    \left right -> case left of
        Z -> right
        S m -> case m of
            Z -> S right
            S m’ -> m’ + S (S right)
]
...
```

Figure 11.2: Endless recursion caused due to the term `pure +` not having a normal form

11.1.2 FR2

The user can control the derivation flow in such a way, that they can skip the derivation for certain redexes interactively.

This requirement has been met. It corresponds to the call command. Using this command the user can skip case applications and beta reductions. This allows the user to ignore the mostly uninteresting parts of the derivation.

Since there are not many cases where a user would want to see all the detailed steps including beta reductions and case application, this setting has been made the default setting for the interactive derivation modes.

In addition to the call command, there is also a command named normalize, which is taking the skipping of steps even further. Due to the recursive nature of the normalize command, however, it is not used as a default command anywhere, as it could easily cause an infinite loop if applied to a term that does not have a normal form, similarly to the problem described in the first FR.

The following two examples from the Substitution Stepper show the difference between using the call command (figure 11.3) and the regular step command (figure 11.4).

```
$ stack exec -- corestepper-exe -m manual -F ../test/Stepped.hs -f test6
(S (S Z)) (+) S (S Z)


Please enter a command:
{- Normalize (((\ds_d2Mn n_a20X -> case ds_d2Mn of
    Z -> n_a20X;
    (S m_a20Y) -> m_a20Y + S n_a20X
)) (S (S Z))) (S (S Z)) -}


(S Z) (+) S (S (S Z))
```

Figure 11.3: Derivation using the call command

Comparing the two examples shows that using the call command, the user can skip 4 steps (1x delta reduction, 2x beta reduction, 1x case application) without losing any valuable information.

```
$ stack exec -- corestepper-exe -m manual -F ../test/Stepped.hs -f test6
(S (S Z)) (+) S (S Z)


Please enter a command:
{- DeltaReduction + -}

(((\ds_d2Mn n_a20X -> case ds_d2Mn of
    Z -> n_a20X;
    (S m_a20Y) -> m_a20Y + S n_a20X
)) (S (S Z))) (S (S Z))


Please enter a command:
{- BetaReduction ds_d2Mn -}

((\n_a20X -> case S (S Z) of
    Z -> n_a20X;
    (S m_a20Y) -> m_a20Y + S n_a20X
)) (S (S Z))


Please enter a command:
{- BetaReduction n_a20X -}

case S (S Z) of
    Z -> S (S Z);
    (S m_a20Y) -> m_a20Y + S (S (S Z))


Please enter a command:
{- Case Application -}

(S Z) (+) S (S (S Z))
```

Figure 11.4: Derivation using the regular step command

11.1.3 FR 3

The user can step through the reduction line by line.

This requirement has been met as well. It corresponds to the manual and interactive modes that are available when using the Substitution Stepper. Through coloring the user can see which subterm is currently selected to be derived next and using the tab key the user can shift the selection to another subterm. Additionally, the application indicates which redex is selected and how many redexes there are in total, which makes it easier for the user to navigate the selection.

```
$ stack exec -- corestepper-exe -m manual -v 1 -F ../test/Stepped.hs -f test
(fib) (S (S (S (S Z))))

(Subterm 1/1)


Please enter a command:
((fib) (S (S (S Z)))) + fib (S (S Z))

(Subterm 2/3)


Please enter a command:
((fib (S (S Z))) + fib (S Z)) (+) fib (S (S Z))

(Subterm 1/5)
```

Figure 11.5: Deriving the left occurrence of `fib` first.

```
$ stack exec -- corestepper-exe -m manual -v 1 -F ../test/Stepped.hs -f test
(fib) (S (S (S (S Z))))

(Subterm 1/1)


Please enter a command:
(fib (S (S (S Z)))) + (fib) (S (S Z))

(Subterm 3/3)


Please enter a command:
(fib (S (S (S Z)))) (+) (fib (S Z)) + fib Z

(Subterm 1/5)
```

Figure 11.6: Deriving the right occurrence of `fib` first.

Figures 11.5 and 11.6 show the same term being derived. However, in the second step, the user chooses two different redexes, which leads to the two third steps looking different.

11.1.4 FR 4

The user can make the derivation run through to the end without requiring any interaction.

The fourth FR is met as well. There are two options for the user to let the derivation run automatically.

First, the user can use the automatic mode to step through the derivation without requiring any input. The user can specify a strategy which is then used by the Substitution Stepper to determine the sequence of reductions.

```
$ stack exec -- corestepper-exe -m automatic -F ../test/Stepped.hs -f testAdd
Derive (S Z) + S Z
--------Step 1---------
RewriteRule: DeltaReduction +

(((\ds_d4e0 n_a3sr -> case ds_d4e0 of
    Z -> n_a3sr;
    (S m_a3ss) -> m_a3ss + S n_a3sr
)) (S Z)) (S Z)

--------Step 2---------
RewriteRule: BetaReduction ds_d4e0

((\n_a3sr -> case S Z of
    Z -> n_a3sr;
    (S m_a3ss) -> m_a3ss + S n_a3sr
)) (S Z)
```

```
--------Step 3---------
RewriteRule: BetaReduction n_a3sr

case S Z of
    Z -> S Z;
    (S m_a3ss) -> m_a3ss + S (S Z)


--------Step 4---------
RewriteRule: Case Application

Z + S (S Z)

--------Step 5---------
RewriteRule: DeltaReduction +

(((\ds_d4e0 n_a3sr -> case ds_d4e0 of
    Z -> n_a3sr;
    (S m_a3ss) -> m_a3ss + S n_a3sr
)) Z) (S (S Z))

--------Step 6---------
RewriteRule: BetaReduction ds_d4e0

((\n_a3sr -> case Z of
    Z -> n_a3sr;
    (S m_a3ss) -> m_a3ss + S n_a3sr
)) (S (S Z))

--------Step 7---------
RewriteRule: BetaReduction n_a3sr

case Z of
    Z -> S (S Z);
    (S m_a3ss) -> m_a3ss + S (S (S Z))


--------Step 8---------
RewriteRule: Case Application

S (S Z)
```

Figure 11.7: Deriving a term using the interactive mode.

When using the manual derivation mode, the user also has the choice to make the derivation continue automatically. This can be seen in figure 11.8, where after the first manual step the derivation is continued automatically.

```
$ stack exec -- corestepper-exe -m manual -v 2 -F ../test/Stepped.hs -f testAdd
(S Z) (+) S Z

(Subterm 1/1)


Please enter a command:
{- Normalize (((\ds_d2MJ n_a21a -> case ds_d2MJ of
    Z -> n_a21a;
    (S m_a21b) -> m_a21b + S n_a21a
)) (S Z)) (S Z) -}


Z (+) S (S Z)

(Subterm 1/1)


Please enter a command:



Derive Z + S (S Z)
--------Step 1---------
RewriteRule: DeltaReduction +

(((\ds_d2MJ n_a21a -> case ds_d2MJ of
    Z -> n_a21a;
    (S m_a21b) -> m_a21b + S n_a21a
)) Z) (S (S Z))

--------Step 2---------
RewriteRule: BetaReduction ds_d2MJ

((\n_a21a -> case Z of
    Z -> n_a21a;
    (S m_a21b) -> m_a21b + S n_a21a
)) (S (S Z))
```

```
--------Step 3---------
RewriteRule: BetaReduction n_a21a

case Z of
    Z -> S (S Z);
    (S m_a21b) -> m_a21b + S (S (S Z))

--------Step 4---------
RewriteRule: Case Application

S (S Z)

Derivation Finished.
```

Figure 11.8: Deriving a term using manual mode and then executing the continue command.

11.1.5 FR 5

The tool supports a verbose variant, that indicates which function has been applied.

This requirement has been met as well. There are up to 4 verbosity levels, depending on the derivation mode.

The automatic mode has two verbosity levels that satisfy the requirement. Verbosity level 1 does not show the justifications for the reduction that was applied, while verbosity level 2 does.

The manual mode has four verbosity levels. The first two levels skip certain intermediate derivation steps according to FR 2, while the two last levels do not skip anything. In addition to that, the even verbosity levels show the justifications while the odd levels do not.

The interactive mode has two verbosity levels. The justifications will always be shown, but the step size is configurable. Verbosity levels 1 and 2 skip certain derivation steps, just like in the manual derivation, while verbosity levels 3 and 4 do not skip anything. So effectively, there are only 2 levels, even though 4 can be specified.

The following 4 figures 11.9, 11.10, 11.11, and 11.12 show the same term being derived in manual mode. Only the first step is shown and each of the figures has a different verbosity level.

```
$ stack exec -- corestepper-exe -m manual -F ../test/Stepped.hs -f test -v 1
(fib) (S (S (S (S Z))))

(Subterm 1/1)


Please enter a command:
(fib (S (S (S Z)))) (+) fib (S (S Z))

(Subterm 1/3)
```

Figure 11.9: A step using verbosity level 1. Some intermediate steps are skipped and there are no justifications.

```
$ stack exec -- corestepper-exe -m manual -F ../test/Stepped.hs -f test -v 2
(fib) (S (S (S (S Z))))

(Subterm 1/1)


Please enter a command:
{- Normalize ((\ds_d2Na -> case ds_d2Na of
    Z -> S Z;
    S ds_d2Ni Z -> S Z;
    S ds_d2Ni (S n_a219) -> (fib (S n_a219)) + fib n_a219
)) (S (S (S (S Z)))) -}


(fib (S (S (S Z)))) (+) fib (S (S Z))

(Subterm 1/3)
```

Figure 11.10: A step using verbosity level 2. Some intermediate steps are skipped and justifications are included.

```
$ stack exec -- corestepper-exe -m manual -F ../test/Stepped.hs -f test -v 3
(fib) (S (S (S (S Z))))

(Subterm 1/1)


Please enter a command:
((\ds_d2Na -> case ds_d2Na of
    Z -> S Z;
    S ds_d2Ni Z -> S Z;
    S ds_d2Ni (S n_a219) -> (fib (S n_a219)) + fib n_a219
)) (S (S (S (S Z))))

(Subterm 1/4)
```

Figure 11.11: A step using verbosity level 3. No steps are skipped and no justifications are displayed.

```
$ stack exec -- corestepper-exe -m manual -F ../test/Stepped.hs -f test -v 4
(fib) (S (S (S (S Z))))

(Subterm 1/1)


Please enter a command:
{- DeltaReduction fib -}


((\ds_d2Na -> case ds_d2Na of
    Z -> S Z;
    S ds_d2Ni Z -> S Z;
    S ds_d2Ni (S n_a219) -> (fib (S n_a219)) + fib n_a219
)) (S (S (S (S Z))))

(Subterm 1/4)
```

Figure 11.12: A step using verbosity level 4. No steps are skipped and justifications are displayed.

11.1.6 FR 6

The user can import user-defined files and step through functions defined in these files.

This requirement is partially met. While files cannot be imported per se, a file containing a Haskell module can be specified as an argument to the Substitution Stepper. However, it is not currently possible to specify multiple modules. Due to this limitation, the requirement is only considered met partially.

```
stack exec -- corestepper-exe -F ../test/Stepped.hs -f test
```

Figure 11.13: Executing the Substitution Stepper only requires passing two simple arguments

11.2 Non-Functional Requirements

11.2.1 NFR 1

The tool is usable for people with little experience in functional programming. The derivation of a user-defined function takes max. one import command and one step command. The function is defined in one file and depends only on other functions in the same file.
The first non-functional requirement has been met. Stepping a function automatically can be done very easily. The only thing that needs to be done is to execute the executable and passing file path and function name like this:

There is no further interaction required when using the automatic derivation, so it is very simple to use. There is no experience needed in order to use the application.

11.2.2 NFR 2

The tool provides an open API or CLI, to open it up to additional UIs.
I would also say that the second NFR is met. The different derivation modes can be accessed easily.

To use the Substitution Stepper with a different UI, the UI needs to be added to the `Manual.hs` and `Interactive.hs` files in the `src/Derivation` folder. The `Automatic.hs` file does not require any changes at all since it does not perform any user interaction.

To use the automatic derivation with a different UI, the `doAutomaticDerivation` function can be called from the new UI. The printing can then be done via the `prettyDerivation` function.

To use the manual derivation with a different UI there are two options. The easier option would be to rewrite the `ManualUI` module in the `src/UI` folder. The functions in there can be adjusted to the new UI for different displaying or prompting of terms and commands.

Should that not offer enough customizability, another UI module could be written, which could then be worked into the `Manual` module in the `src/Derivation` folder. If it behaves similarly to the `ManualUI` module, then the functions in the `Manual` module can just be changed to call the new UI rather than the old one. Only if it is very different from the original UI in how it works would the replacement of the UI be a bit of a bigger task.

Using the interactive derivation with a different UI can be achieved the same way as for the manual derivation but instead of modifying the `ManualUI` or `Manual` modules, the `InteractiveUI` and `Interactive` modules need to be adjusted.

Since in most cases, it should not be too much work to adjust the UI as described above, I would say that the requirement is met.

11.3 Limitations

11.3.1 Prelude

One of the biggest limitations of the Substitution Stepper currently is that the Prelude is not supported. This includes datatypes like String and Int as well as all the functions that are defined in the prelude. However, instances like Monad, Applicative, etc. can still be implemented and work. Most of the prelude can be implemented easily, which would also be a useful exercise for students of the Functional Programming class. However, there are some types that cannot be implemented too easily, like numerical datatypes and operations on them.

The reason for this limitation has to do with the CoreStepper project, which this application is based on. It is currently not possible to compile expressions and functions that contain Prelude functionality to Core.

11.3.2 IO

It is currently unclear if the Substitution Stepper would be able to step impure IO functions. This is partially due to the fact that there is no mechanism to perform the input or output operations with, even if the limitation with the Prelude functionality would be resolved.

11.4 Known Issues

11.4.1 LeftmostInnermost Strategy

The leftmost innermost derivation strategy does not work as expected. Due to the fact that function applications and operations are not saved infix in the Core representation, the implementation for the leftmost innermost derivation Strategy is a bit faulty.

The implementation assumes that the arguments to a function/operation are a (in)direct child of the function/operation. This is not the case in Core. The second argument for the operation is a sibling of the operation in Core representation, not a child. This causes the operation to be perceived as nested more deeply than the argument, which can cause infinite recursion in the worst case when used with the automatic derivation mode.

This issue still needs to be looked into, to get the ordering right.

11.4.2 Errors when using Low Verbosity

When using a low verbosity level (1 or 2) during manual or interactive derivation, some terms containing let bindings might erroneously indicate that they cannot be derived, while in fact, they should be derivable.

The same applies when using the call command on a term containing a let binding, as in the background, stepping at low verbosity and using the call command are the same operation.

The reason for this issue is not known yet and needs to be looked into.

11.4.3 Normalization

The normalization command when using manual derivation does not work as intended yet. It does not always fully normalize the term and stops in seemingly random places. The cause behind this issue is not clear yet and needs to be looked into.

11.4.4 Invisible Beta Reductions

Some functions in Core require types as arguments. This is due to the explicit typing used in Core. Some generic functions need to take types as arguments so the explicit typing works in later derivation steps aswell. I did not want to display types, because I think that they just clutter up the display. This, however, causes some lambdas to be applied to "invisible" arguments. In fact, they are applied to the types, but the types are invisible to the user.

This might be confusing, but it is necessary. While one could remove the types from the terms, there are lambdas that expect a type as an argument and there is no way to distinguish these lambdas from other lambdas. So if one were to remove the type but leave the lambda in there, it would result in a lambda that cannot be further reduced, since it is missing its argument.

Luckily, this is only a problem when using the most verbose modes during stepping. If a mode with lower verbosity is used, this problem will not be noticeable at all.

Chapter 12

Conclusion & Next Steps

12.1 Requirements

In the previous chapter 11, the requirements were analyzed and checked whether they have been met or not. As table 12.1 shows, most of the requirements were met.

FR1 probably requires an approach different from what has been attempted and FR6 could probably be implemented without too much of a struggle. Since the CoreStepper is already capable of loading a module and returning all the bindings in there, it is reasonable to assume that one could use that multiple times on different modules to get all bindings defined across all the modules.

Requirement	**Status**
FR 1	Not met
FR 2	Met
FR 3	Met
FR 4	Met
FR 5	Met
FR 6	Partially met
NFR 1	Met
NFR 2	Met

Table 12.1: Summary of which requirements have been met and which have not.

Originally the MVP was planned to contain FR1, FR3, FR4, FR5, and FR6. Despite FR1 not being met, I would consider the MVP being reached. I would say that FR1 is not as important as FR2. So I would redefine the MVP to contain FR2, FR3, FR4, FR5, and FR6.

This has a couple of reasons. First of all, the two requirements are very similar. In essence, they have the same responsibility. Both requirements include functionality that happens in the background. However, the functionality described by FR1 does not happen automatically. The user needs to manually add functions that should be skipped. This makes the usage of FR1 more complicated.

On the other hand, in order to make use of the functionality described in FR2 the user does not have to do anything. For both the interactive and manual derivation modes, the skipping of uninteresting steps is the default behavior. This, in my opinion, is already a big advantage over the functionality described in FR1.

Secondly, when the requirements for the Substitution Stepper were created, FR1 was counted towards the MVP because it seemed easier to implement than FR2, which lead to it being prioritized more. During the implementation, it became clear that FR2 would be easier to implement than FR1, which is why the priorities have shifted.

For these reasons, I have decided to implement FR2 before FR1, even though originally FR1 was planned to be part of the MVP, not FR2.

12.2 Reflection

The goal of this thesis was to improve the usability of the Substitution Stepper. In the end, the thesis was not only limited to usability but included also a bit of a redesign of the stepping functionality. Due to change and expansion of scope, some requirements that were originally created could not be met.

During this thesis, advances, like the ability to step instance functions, were made, thus reducing the limitations of the Substitution Stepper.

The redesign of the stepping functionality also came with the advantage of being theoretically able to step other languages that are not Haskell, like the Lambda Calculus. It is possible now to add new steppable languages with relatively low effort, since the user interaction does not depend on the stepped language. Things like the highlighting of subterms and selection of subterms to derive don't have to be implemented for new languages. This can greatly reduce the effort that is needed to execute a project similar to this in the future.

Despite not all the requirements being met, the result of the thesis is a usable application that allows the user to step through Haskell code. The most important requirements for a working product were met and I am confident that the application would prove useful to someone learning Haskell. For this reason I would say that the main goal of the thesis was reached. The following section will provide some ideas for the next steps that could further improve the Substitution Stepper.

While I consider this thesis a success, there are still some things that I would have adjusted in retrospect.

For starters, I would probably adjust the `Subterm` data structure a bit. The way annotations are handled currently is a bit more complicated than it needs to be. Instead of making the stepped terms handle annotations themselves, I would make the `Subterm` data structure handle the annotations.

By doing this, the `Highlightable` class would not have to be implemented by every single term structure that is being stepped. It could be implemented using only the `Subterm` data structure. Additionally, one would not have to worry about the annotations changing the actual position of the subterm, which is something that could happen if the annotations are not implemented properly.

In short: Making the annotations part of the `Subterm` could help reduce code duplication, and in general, the amount of code that needs to be written to add another term structure to the Substitution Stepper.

Another change I would do is to add some kind of UI interface that allows the modules responsible for the derivation to be decoupled from the UI modules. This change is described in subsection 12.3.3.

12.3 Next Steps

This section proposes adjustments or expansions of the Substitution Stepper that could be done when continuing development on this project. The adjustments/expansions are not listed in any particular order.

12.3.1 Support for the Prelude

One of the probably most impactful adjustments would be to add support for the Prelude. Since the limitation of not being able to use the data types defined in the prelude is quite a big handicap, it would make sense to work on this first.

Considering that especially the numerical data types are used often in the Functional Programming class to demonstrate functions like sum, foldl/foldr, etc. it would be very nice for the students using this tool during class to be able to use them. Another important concept of Haskell, which is currying, could not be demonstrated either, since tuples are not supported either. And from my experience, functions like (un)curry or flip were the functions that most students struggled with a bit, besides Monads and Applicatives.

Having to implement your own data type is a bit of a big hurdle at the start. Especially data types like `data Nat = S Nat | Z` are hard to understand when starting out with functional programming.

For these reasons, I would strongly recommend making this a priority when continuing development on the Substitution Stepper.

12.3.2 Collapsing/Shortening of (Sub)terms

Collapsing or shortening (sub)terms was originally also a desired functionality. However, it did not make the list of requirements as it was not prioritized highly enough. During the development of the Substitution Stepper there was not enough time to add additional requirements, which is why it did not make the list later either.

Despite this, being able to shorten a term would be very useful, especially since some terms can get very long.

An idea for how to implement this was to let the user manually choose a subterm and offering them a command with which they could replace the whole subterm with a custom name. This would be somewhat simple to implement as it does not require any 'intelligence' on the side of the application. In an earlier iteration of the Substitution Stepper the `mode` command was used to switch between the selection of redexes only or the selection of any arbitrary subterm. It would be imaginable to bring back this functionality and to add a `rename` or `replace` command that would replace the selected subterm with the specified name.

This could be a way to make the term shorter, however it might not be an ideal approach as it requires additional user interaction, which might be a bit cumbersome for the user.

Another problem with this kind of approach is that in long terms there is an exponentially larger amount of subterms, which could make the selection of subterms even more cumbersome if no better way is found besides using the tab key.

The collapsing or shortening of (sub)terms could maybe also be done automatically, however, I have not been able to come up with an idea on how to do that.

12.3.3 Simplification of the UI modules

To simplify code and prevent duplication, maybe a `UI` class could be created that is used in the `Derivation.Manual` and `Derivation.Interactive` modules. This could allow these modules to call UI functions while still remaining independent of the `ManualUI` and `InteractiveUI` modules. This would probably be better practice, as it would also make it easier to replace the UI, since the `Derivation.Manual` and `Derivation.Interactive` modules could be left unchanged.

I have tried to create a UI class that acts as an interface between the UI modules and the Derivation modules. I might have taken the wrong approach there, as I was not able to implement it in a sufficiently elegant way.

12.3.4 UI State

Adding something like a UI State to the UI types might also prove beneficial. The UI State could for example keep track of how many lines have been written, in order to be able to erase them safely and reliably.

The current implementation is not very elegant, as it re-computes the length of the previous prompt and then erases that amount of lines. This could potentially be an error-prone approach, while the approach using the UI State is probably safe.

I could also imagine the UI State keeping track of the current CompletionFunc. This might allow for the usage of multiple completion functions during runtime. For example, if no function is specified in the command line arguments, the application could prompt the user to input the function name, and provide completion support for the function names when doing so.

And as soon as the derivation process starts, the application could switch the completion function and provide the usual completion function that cycles the selection of redexes.

This change could probably be done in one go together with the simplification of the UI modules, improving the overall cleanliness of the UI code.

12.3.5 Improvements to Pretty Printing

There are still some improvements that could be made to the pretty printing of the Core expressions. Especially when it comes to operator precedence and the placement of line breaks in the CLI.

The Substitution Stepper currently disregards operator precedence completely as I have not gotten to look into that yet. The placement of line breaks is not perfected either, as sometimes it can look a bit confusing when the terms get too long.

Overall, this would be an improvement that is worth implementing. It would make it easier for users to read and understand what is going on.

12.3.6 Printing Terms via Show

During the development of the Substitution Stepper there was an idea that could help the readability of the terms greatly. The idea was to display data types that implement the Show class via show, instead of showing the constructors. This however was not possible due to the limitation that the Prelude - and thus strings - are not supported currently.

But even if this limitation is resolved, it might still not be possible to implement this idea. To be able to implement this idea, the Substitution Stepper needs to know the type of an expression before attempting to display it via show. Since the Substitution Stepper currently ignores types, some changes might be needed to enable this. If the Show instance is constrained to f.ex. another type needing to implement Show as well, the implementation of this feature might become very complex.

12.3.7 Implementation of different Languages

It might be interesting to implement additional languages that can be derived by the Substitution Stepper. For example, the Lambda Calculus language might be interesting for the Functional Programming class, as it is also part of the course. Considering that there is already a LambdaCalculator project used during the course, this might not be necessary. Though I could imagine that it might be convenient to have the capability to step Haskell and Lambda Calculus in the same application.

Part VI

Appendix

Chapter 13

Examples

This chapter contains some examples of how the Substitution Stepper steps some of the examples from the task description. The derivations are somewhat long still compared to the examples given in the task description.

13.1 Example 1

The first example sums up three elements in a list. Due to the limitations of the stepper, the Nat datatype was used instead of Int.

```
Carlo@DESKTOP-UFN2VAS MINGW64 ~/Documents/SubStep/core-stepper/test (main)
$ stack exec -- corestepper-exe -m manual -F Stepped.hs -f test2
sum ((Cons (S Z)) ((Cons (S (S Z))) ((Cons (S (S (S Z)))) Empty)))

(Subterm 1/1)


Please enter a command:
{- Apply sum ((Cons (S Z)) ((Cons (S (S Z))) ((Cons (S (S (S Z)))) Empty))) -}


(S Z) + sum ((Cons (S (S Z))) ((Cons (S (S (S Z)))) Empty))

(Subterm 2/2)


Please enter a command:
{- Apply sum ((Cons (S (S Z))) ((Cons (S (S (S Z)))) Empty)) -}


(S Z) + (S (S Z)) + sum ((Cons (S (S (S Z)))) Empty)

(Subterm 3/3)


Please enter a command:
{- Apply sum ((Cons (S (S (S Z)))) Empty) -}


(S Z) + (S (S Z)) + (S (S (S Z))) + sum Empty

(Subterm 4/4)


Please enter a command:
{- Apply sum Empty -}


(S Z) + (S (S Z)) + (S (S (S Z))) + Z

(Subterm 1/3)


Please enter a command:
{- Apply (S Z) + (S (S Z)) + (S (S (S Z))) + Z -}


Z + S ((S (S Z)) + (S (S (S Z))) + Z)

(Subterm 1/3)
```

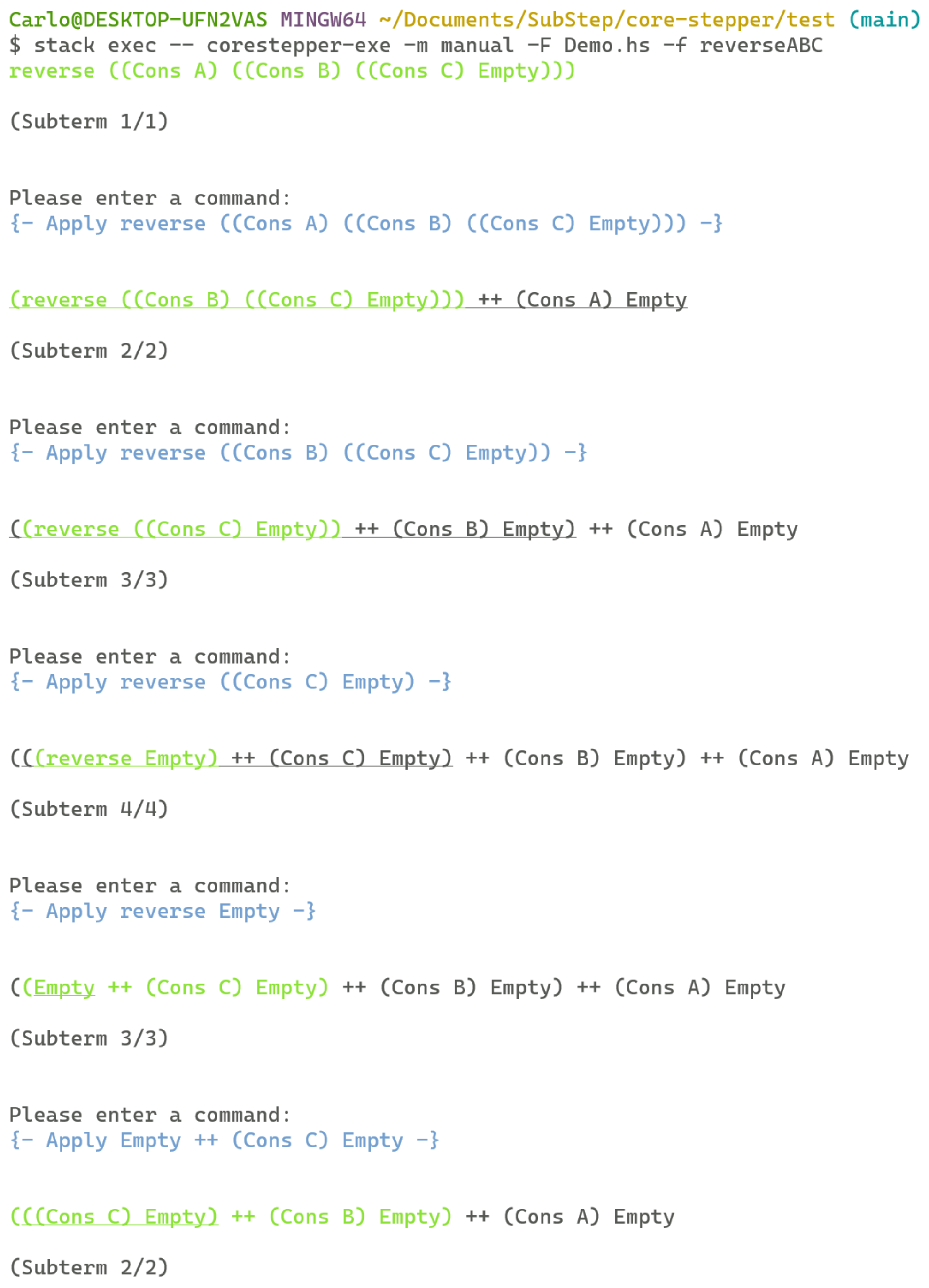
```
Please enter a command:
{- Apply Z + S ((S (S Z)) + (S (S (S Z))) + Z) -}


S ((S (S Z)) + (S (S (S Z))) + Z)

(Subterm 1/2)


Please enter a command:
{- Apply (S (S Z)) + (S (S (S Z))) + Z -}


S ((S Z) + S ((S (S (S Z))) + Z))

(Subterm 1/2)


Please enter a command:
{- Apply (S Z) + S ((S (S (S Z))) + Z) -}


S (Z + S (S ((S (S (S Z))) + Z)))

(Subterm 1/2)


Please enter a command:
{- Apply Z + S (S ((S (S (S Z))) + Z)) -}


S (S (S ((S (S (S Z))) + Z)))

(Subterm 1/1)


Please enter a command:
{- Apply (S (S (S Z))) + Z -}


S (S (S ((S (S Z)) + S Z)))

(Subterm 1/1)


Please enter a command:
{- Apply (S (S Z)) + S Z -}


S (S (S ((S Z) + S (S Z))))

(Subterm 1/1)
```

```
Please enter a command:
{- Apply Z + S ((S (S Z)) + (S (S (S Z))) + Z) -}


S ((S (S Z)) + (S (S (S Z))) + Z)

(Subterm 1/2)


Please enter a command:
{- Apply (S (S Z)) + (S (S (S Z))) + Z -}


S ((S Z) + S ((S (S (S Z))) + Z))

(Subterm 1/2)


Please enter a command:
{- Apply (S Z) + S ((S (S (S Z))) + Z) -}


S (Z + S (S ((S (S (S Z))) + Z)))

(Subterm 1/2)


Please enter a command:
{- Apply Z + S (S ((S (S (S Z))) + Z)) -}


S (S (S ((S (S (S Z))) + Z)))

(Subterm 1/1)


Please enter a command:
{- Apply (S (S (S Z))) + Z -}


S (S (S ((S (S Z)) + S Z)))

(Subterm 1/1)


Please enter a command:
{- Apply (S (S Z)) + S Z -}


S (S (S ((S Z) + S (S Z))))

(Subterm 1/1)
```

Figure 13.1: The first example, summing 1, 2, and 3 in a list to obtain the value 6.

13.2 Example 2

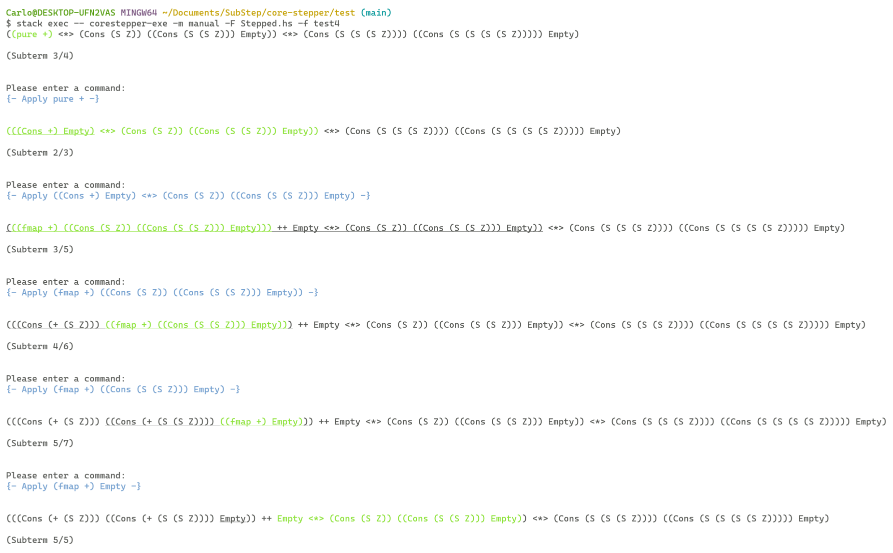
The second example reverses a list containing 3 elements. Again, this example had to be adjusted, this time it uses letters as they are more easily readable than the Nat datatype.



```
Carlo@DESKTOP-UFN2VAS MINGW64 ~/Documents/SubStep/core-stepper/test (main)
$ stack exec -- corestepper-exe -m manual -F Demo.hs -f reverseABC
reverse ((Cons A) ((Cons B) ((Cons C) Empty)))

(Subterm 1/1)


Please enter a command:
{- Apply reverse ((Cons A) ((Cons B) ((Cons C) Empty))) -}


(reverse ((Cons B) ((Cons C) Empty))) ++ (Cons A) Empty

(Subterm 2/2)


Please enter a command:
{- Apply reverse ((Cons B) ((Cons C) Empty)) -}


((reverse ((Cons C) Empty)) ++ (Cons B) Empty) ++ (Cons A) Empty

(Subterm 3/3)


Please enter a command:
{- Apply reverse ((Cons C) Empty) -}


(((reverse Empty) ++ (Cons C) Empty) ++ (Cons B) Empty) ++ (Cons A) Empty

(Subterm 4/4)


Please enter a command:
{- Apply reverse Empty -}


((Empty ++ (Cons C) Empty) ++ (Cons B) Empty) ++ (Cons A) Empty

(Subterm 3/3)


Please enter a command:
{- Apply Empty ++ (Cons C) Empty -}


(((Cons C) Empty) ++ (Cons B) Empty) ++ (Cons A) Empty

(Subterm 2/2)
```

```
Please enter a command:
{- Apply ((Cons C) Empty) ++ (Cons B) Empty -}


((Cons C) (Empty ++ (Cons B) Empty)) ++ (Cons A) Empty

(Subterm 2/2)


Please enter a command:
{- Apply Empty ++ (Cons B) Empty -}


((Cons C) ((Cons B) Empty)) ++ (Cons A) Empty

(Subterm 1/1)


Please enter a command:
{- Apply ((Cons C) ((Cons B) Empty)) ++ (Cons A) Empty -}


(Cons C) (((Cons B) Empty) ++ (Cons A) Empty)

(Subterm 1/1)


Please enter a command:
{- Apply ((Cons B) Empty) ++ (Cons A) Empty -}


(Cons C) ((Cons B) (Empty ++ (Cons A) Empty))

(Subterm 1/1)


Please enter a command:


(Cons C) ((Cons B) ((Cons A) Empty))

Derivation Finished.
```

Figure 13.2: The second example, reversing the list [A,B,C].



```
Carlo@DESKTOP-UFN2VAS MINGW64 ~/Documents/SubStep/core-stepper/test (main)
$ stack exec -- corestepper-exe -m manual -F Stepped.hs -f test4
((pure +) <*> (Cons (S Z)) ((Cons (S (S Z))) Empty)) <*> (Cons (S (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty)

(Subterm 3/4)


Please enter a command:
{- Apply pure + -}


(((Cons +) Empty) <*> (Cons (S Z)) ((Cons (S (S Z))) Empty)) <*> (Cons (S (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty)

(Subterm 2/3)


Please enter a command:
{- Apply ((Cons +) Empty) <*> (Cons (S Z)) ((Cons (S (S Z))) Empty) -}


(((fmap +) ((Cons (S Z)) ((Cons (S (S Z))) Empty))) ++ Empty <*> (Cons (S Z)) ((Cons (S (S Z))) Empty)) <*> (Cons (S (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty)

(Subterm 3/5)


Please enter a command:
{- Apply (fmap +) ((Cons (S Z)) ((Cons (S (S Z))) Empty)) -}


(((Cons (+ (S Z))) ((fmap +) ((Cons (S (S Z))) Empty))) ++ Empty <*> (Cons (S Z)) ((Cons (S (S Z))) Empty)) <*> (Cons (S (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty)

(Subterm 4/6)


Please enter a command:
{- Apply (fmap +) ((Cons (S (S Z))) Empty) -}


(((Cons (+ (S Z))) ((Cons (+ (S (S Z)))) ((fmap +) Empty))) ++ Empty <*> (Cons (S Z)) ((Cons (S (S Z))) Empty)) <*> (Cons (S (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty)

(Subterm 5/7)


Please enter a command:
{- Apply (fmap +) Empty -}


(((Cons (+ (S Z))) ((Cons (+ (S (S Z)))) Empty)) ++ Empty <*> (Cons (S Z)) ((Cons (S (S Z))) Empty)) <*> (Cons (S (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty)

(Subterm 5/5)
```

13.3 Example 3

The third example uses the Nat datatype again. It corresponds to `pure (+) <*> [1,2] <*> [3,4]`

```
Please enter a command:
{- Apply Empty <*> (Cons (S Z)) ((Cons (S (S Z))) Empty) -}


(((Cons (+ (S Z))) ((Cons (+ (S (S Z)))) Empty)) ++ Empty) <*> (Cons (S (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty)

(Subterm 2/4)


Please enter a command:
{- Apply ((Cons (+ (S Z))) ((Cons (+ (S (S Z)))) Empty)) ++ Empty -}


((Cons (+ (S Z))) (((Cons (+ (S (S Z)))) Empty) ++ Empty)) <*> (Cons (S (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty)

(Subterm 3/4)


Please enter a command:
{- Apply ((Cons (+ (S (S Z)))) Empty) ++ Empty -}


((Cons (+ (S Z))) ((Cons (+ (S (S Z)))) (Empty ++ Empty))) <*> (Cons (S (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty)

(Subterm 4/4)


Please enter a command:
{- Apply Empty ++ Empty -}


((Cons (+ (S Z))) ((Cons (+ (S (S Z)))) Empty)) <*> (Cons (S (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty)

(Subterm 1/3)


Please enter a command:
{- Apply ((Cons (+ (S Z))) ((Cons (+ (S (S Z)))) Empty)) <*> (Cons (S (S (S Z))))
((Cons (S (S (S (S Z))))) Empty) -}


((fmap (+ (S Z))) ((Cons (S (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty))) ++ ((Cons (+ (S (S Z)))) Empty) <*> (Cons (S (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty)

(Subterm 2/5)


Please enter a command:
{- Apply (fmap (+ (S Z))) ((Cons (S (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty)) -}


((Cons ((S Z) + S (S (S Z)))) ((fmap (+ (S Z))) ((Cons (S (S (S (S Z))))) Empty))) ++ ((Cons (+ (S (S Z)))) Empty) <*> (Cons (S (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty)

(Subterm 3/6)
```

```
Please enter a command:
{- Apply (fmap (+ (S Z))) ((Cons (S (S (S (S Z))))) Empty) -}


((Cons ((S Z) + S (S (S Z)))) ((Cons ((S Z) + S (S (S (S Z))))) ((fmap (+ (S Z))) Empty))) ++ ((Cons (+ (S (S Z)))) Empty) <*> (Cons (S (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty)

(Subterm 4/7)


Please enter a command:
{- Apply (fmap (+ (S Z))) Empty -}


((Cons ((S Z) + S (S (S Z)))) ((Cons ((S Z) + S (S (S (S Z))))) Empty)) ++ ((Cons (+ (S (S Z)))) Empty) <*> (Cons (S (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty)

(Subterm 2/5)


Please enter a command:
{- Apply (S Z) + S (S (S Z)) -}


((Cons (Z + S (S (S (S Z))))) ((Cons ((S Z) + S (S (S (S Z))))) Empty)) ++ ((Cons (+ (S (S Z)))) Empty) <*> (Cons (S (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty)

(Subterm 2/5)


Please enter a command:
{- Apply Z + S (S (S (S Z))) -}


((Cons (S (S (S (S Z))))) ((Cons ((S Z) + S (S (S (S Z))))) Empty)) ++ ((Cons (+ (S (S Z)))) Empty) <*> (Cons (S (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty)

(Subterm 2/4)


Please enter a command:
{- Apply (S Z) + S (S (S (S Z))) -}


((Cons (S (S (S (S Z))))) ((Cons (Z + S (S (S (S (S Z)))))) Empty)) ++ ((Cons (+ (S (S Z)))) Empty) <*> (Cons (S (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty)

(Subterm 2/4)


Please enter a command:
{- Apply Z + S (S (S (S (S Z)))) -}


((Cons (S (S (S (S Z))))) ((Cons (S (S (S (S (S Z)))))) Empty)) ++ ((Cons (+ (S (S Z)))) Empty) <*> (Cons (S (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty)

(Subterm 2/3)
```

```
Please enter a command:
{- Apply ((Cons (+ (S (S Z)))) Empty) <*> (Cons (S (S (S Z)))) ((Cons (S (S (S (S Z)))))
Empty) -}


((Cons (S (S (S (S Z))))) ((Cons (S (S (S (S (S Z)))))) Empty)) ++ ((fmap (+ (S (S Z)))) ((Cons (S (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty))) ++ Empty <*> (Cons (S (S (S Z)))) ((Cons (S (S (S (S Z)))))
Empty)

(Subterm 3/5)


Please enter a command:
{- Apply (fmap (+ (S (S Z)))) ((Cons (S (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty)) -}


((Cons (S (S (S (S Z))))) ((Cons (S (S (S (S (S Z)))))) Empty)) ++ ((Cons ((S (S Z)) + S (S (S Z)))) ((fmap (+ (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty))) ++ Empty <*> (Cons (S (S (S Z))))
((Cons (S (S (S (S Z))))) Empty)

(Subterm 4/6)


Please enter a command:
{- Apply (fmap (+ (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty) -}


((Cons (S (S (S (S Z))))) ((Cons (S (S (S (S (S Z)))))) Empty)) ++ ((Cons ((S (S Z)) + S (S (S Z)))) ((Cons ((S (S Z)) + S (S (S (S Z))))) ((fmap (+ (S (S Z)))) Empty))) ++ Empty <*> (Cons (S (S (S Z))))
((Cons (S (S (S (S Z))))) Empty)

(Subterm 5/7)


Please enter a command:
{- Apply (fmap (+ (S (S Z)))) Empty -}


((Cons (S (S (S (S Z))))) ((Cons (S (S (S (S (S Z)))))) Empty)) ++ ((Cons ((S (S Z)) + S (S (S Z)))) ((Cons ((S (S Z)) + S (S (S (S Z))))) Empty)) ++ Empty <*> (Cons (S (S (S Z)))) ((Cons (S (S (S (S Z)))))
Empty)

(Subterm 5/5)


Please enter a command:
{- Apply Empty <*> (Cons (S (S (S Z)))) ((Cons (S (S (S (S Z))))) Empty) -}


((Cons (S (S (S (S Z))))) ((Cons (S (S (S (S (S Z)))))) Empty)) ++ ((Cons ((S (S Z)) + S (S (S Z)))) ((Cons ((S (S Z)) + S (S (S (S Z))))) Empty)) ++ Empty

(Subterm 3/4)
```

```
Please enter a command:
{- Apply (S (S Z)) + S (S (S Z)) -}


((Cons (S (S (S (S Z))))) ((Cons (S (S (S (S (S Z)))))) Empty)) ++ ((Cons ((S Z) + S (S (S (S Z))))) ((Cons ((S (S Z)) + S (S (S (S Z))))) Empty)) ++ Empty

(Subterm 3/4)


Please enter a command:
{- Apply (S Z) + S (S (S (S Z))) -}


((Cons (S (S (S (S Z))))) ((Cons (S (S (S (S (S Z)))))) Empty)) ++ ((Cons (Z + S (S (S (S (S Z)))))) ((Cons ((S (S Z)) + S (S (S (S Z))))) Empty)) ++ Empty

(Subterm 3/4)


Please enter a command:
{- Apply Z + S (S (S (S (S Z)))) -}


((Cons (S (S (S (S Z))))) ((Cons (S (S (S (S (S Z)))))) Empty)) ++ ((Cons (S (S (S (S (S Z)))))) ((Cons ((S (S Z)) + S (S (S (S Z))))) Empty)) ++ Empty

(Subterm 3/3)


Please enter a command:
{- Apply (S (S Z)) + S (S (S (S Z))) -}


((Cons (S (S (S (S Z))))) ((Cons (S (S (S (S (S Z)))))) Empty)) ++ ((Cons (S (S (S (S (S Z)))))) ((Cons ((S Z) + S (S (S (S (S Z)))))) Empty)) ++ Empty

(Subterm 3/3)


Please enter a command:
{- Apply (S Z) + S (S (S (S (S Z)))) -}


((Cons (S (S (S (S Z))))) ((Cons (S (S (S (S (S Z)))))) Empty)) ++ ((Cons (S (S (S (S (S Z)))))) ((Cons (Z + S (S (S (S (S (S Z))))))) Empty)) ++ Empty

(Subterm 3/3)


Please enter a command:
{- Apply Z + S (S (S (S (S (S Z))))) -}


((Cons (S (S (S (S Z))))) ((Cons (S (S (S (S (S Z)))))) Empty)) ++ ((Cons (S (S (S (S (S Z)))))) ((Cons (S (S (S (S (S (S Z))))))) Empty)) ++ Empty

(Subterm 2/2)
```

```
Please enter a command:
{- Apply ((Cons (S (S (S (S (S Z)))))) ((Cons (S (S (S (S (S (S Z))))))) Empty)) ++ Empty -}


((Cons (S (S (S (S Z))))) ((Cons (S (S (S (S (S Z)))))) Empty)) ++ (Cons (S (S (S (S (S Z)))))) (((Cons (S (S (S (S (S (S Z))))))) Empty) ++ Empty)

(Subterm 2/2)


Please enter a command:
{- Apply ((Cons (S (S (S (S (S (S Z))))))) Empty) ++ Empty -}


((Cons (S (S (S (S Z))))) ((Cons (S (S (S (S (S Z)))))) Empty)) ++ (Cons (S (S (S (S (S Z)))))) ((Cons (S (S (S (S (S (S Z))))))) (Empty ++ Empty))

(Subterm 2/2)


Please enter a command:
{- Apply Empty ++ Empty -}


((Cons (S (S (S (S Z))))) ((Cons (S (S (S (S (S Z)))))) Empty)) ++ (Cons (S (S (S (S (S Z)))))) ((Cons (S (S (S (S (S (S Z))))))) Empty)

(Subterm 1/1)


Please enter a command:
{- Apply ((Cons (S (S (S (S Z))))) ((Cons (S (S (S (S (S Z)))))) Empty))
++ (Cons (S (S (S (S (S Z)))))) ((Cons (S (S (S (S (S (S Z))))))) Empty) -}


(Cons (S (S (S (S Z))))) (((Cons (S (S (S (S (S Z)))))) Empty) ++ (Cons (S (S (S (S (S Z)))))) ((Cons (S (S (S (S (S (S Z))))))) Empty))

(Subterm 1/1)


Please enter a command:
{- Apply ((Cons (S (S (S (S (S Z)))))) Empty) ++ (Cons (S (S (S (S (S Z))))))
((Cons (S (S (S (S (S (S Z))))))) Empty) -}


(Cons (S (S (S (S Z))))) ((Cons (S (S (S (S (S Z)))))) (Empty ++ (Cons (S (S (S (S (S Z)))))) ((Cons (S (S (S (S (S (S Z))))))) Empty)))

(Subterm 1/1)


Please enter a command:


(Cons (S (S (S (S Z))))) ((Cons (S (S (S (S (S Z)))))) ((Cons (S (S (S (S (S Z)))))) ((Cons (S (S (S (S (S (S Z))))))) Empty)))

Derivation Finished.
```

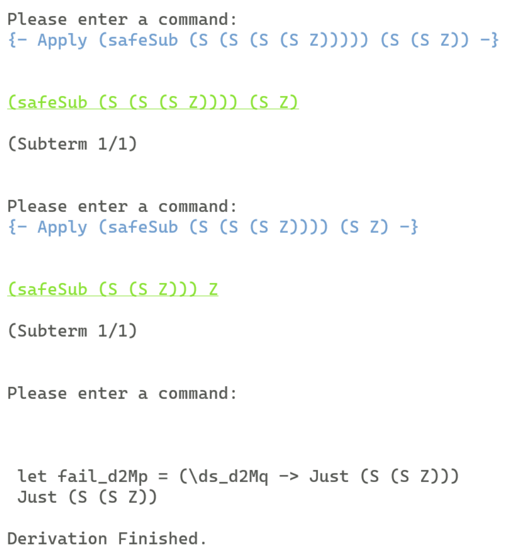
Figure 13.3: The third example, equivalent to `pure (+) <*> [1,2] <*> [3,4]`

13.4 Example 4

The fourth example needed to make a slightly bigger change again, since it is using the Nat datatype as well. Since division is harder to implement on the Nat datatype, I have chosen to perform a subtraction instead. Similar to the division for the Int datatype, the subtraction for the Nat datatype can cause an error as well, if a bigger number is subtracted from a smaller number (negative number error).

Because of this, the fourth example here is pretty much equivalent to the example in the task description, even though different operations and datatypes are used.

```
Carlo@DESKTOP-UFN2VAS MINGW64 ~/Documents/SubStep/core-stepper/test (main)
$ stack exec -- corestepper-exe -m manual -F Stepped.hs -f test5
(pure (S (S (S (S (S Z)))))) >>= (\n_aAH -> (pure (S (S (S Z)))) >>= (\m_aAI -> (safeSub n_aAH) m_aAI))

(Subterm 2/5)


Please enter a command:
{- Apply pure (S (S (S (S (S Z))))) -}


(Just (S (S (S (S (S Z)))))) >>= (\n_aAH -> (pure (S (S (S Z)))) >>= (\m_aAI -> (safeSub n_aAH) m_aAI))

(Subterm 1/4)


Please enter a command:
{- Apply (Just (S (S (S (S (S Z)))))) >>= (\n_aAH -> (pure (S (S (S Z))))
>>= (\m_aAI -> (safeSub n_aAH) m_aAI)) -}


(pure (S (S (S Z)))) >>= (\m_aAI -> (safeSub (S (S (S (S (S Z)))))) m_aAI)

(Subterm 2/3)


Please enter a command:
{- Apply pure (S (S (S Z))) -}


(Just (S (S (S Z)))) >>= (\m_aAI -> (safeSub (S (S (S (S (S Z)))))) m_aAI)

(Subterm 1/2)


Please enter a command:
{- Apply (Just (S (S (S Z)))) >>= (\m_aAI -> (safeSub (S (S (S (S (S Z)))))) m_aAI) -}


(safeSub (S (S (S (S (S Z)))))) (S (S (S Z)))

(Subterm 1/1)


Please enter a command:
{- Apply (safeSub (S (S (S (S (S Z)))))) (S (S (S Z))) -}


(safeSub (S (S (S (S Z))))) (S (S Z))

(Subterm 1/1)
```



```
Please enter a command:
{- Apply (safeSub (S (S (S (S Z))))) (S (S Z)) -}


(safeSub (S (S (S Z)))) (S Z)

(Subterm 1/1)


Please enter a command:
{- Apply (safeSub (S (S (S Z)))) (S Z) -}


(safeSub (S (S Z))) Z

(Subterm 1/1)


Please enter a command:



 let fail_d2Mp = (\ds_d2Mq -> Just (S (S Z)))
 Just (S (S Z))

Derivation Finished.
```

Figure 13.4: The fourth example, equivalent to {`do n <- 5; m <- 3; safeSub n m`}