OST
Eastern Switzerland
University of Applied Sciences

Haskell Substitution Stepper

Task Description

1. Setting

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

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

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

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

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

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

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

2. Goals

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

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

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

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

3. Deliverables

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

All deliverables may be submitted in digital form.