Haskell Substitution Stepper

An equational reasoning assistant for teaching and debugging

Student

Jan Huber

Definition of Task: In the imperative programming paradigm, a debugging tool with an appropriate visualization of the program counter and internal state is often used to learn about the program execution. However, functional programming languages like Haskell do not have the concept of a program counter or internal state. Executing a program in Haskell is typically viewed as an evaluation of an expression using repeated substitution.

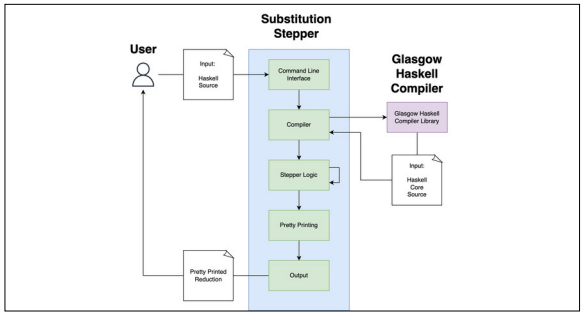
Although lists of substitutions are often used in textbooks, there is no tool to automatically generate such substitutions for the full Haskell language. Having a tool to generate such substitutions can help to learn programming and help to debug programs written in the functional style.

Hence, the main aim of this project is to implement a substitution stepper for Haskell that visualizes the execution of a program.

Approach: The first proof of concept was built upon a simplified subset of the Haskell abstract syntax tree. Further research about the Glasgow Haskell Compiler (GHC) and discussions with Haskell experts showed that it is more feasible to work with GHCs intermediate language „Core“ than with Haskell itself. This change has made it possible to support a larger part of the language.

Result: The result of this project is a command line tool which can step through most Haskell programs and produces outputs that closely look like examples given in textbooks. In comparison to similar existing tools, the Haskell Substitution Stepper supports a larger part of Haskell and is more closely coupled with the Glasgow Haskell Compiler. The output can be pretty-printed in the "original Haskell" syntax or in a "Core" syntax.

Architecture of the application
Own presentment



Haskell Core Intermediate Language Representation
Glasgow Haskell Compiler

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

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

type Arg b = Expr b

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

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

Example Substitutions
Own presentment

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

Manual Reduction (Textbook)

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

Automatic Reduction (Stepper)

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

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

Manual Reduction (Textbook)

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

Automatic Reduction (Stepper)

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


Examiner
Prof. Dr. Farhad D. Mehta

Co-Advisor
Jasper Van der Jeugt, Jasper Van der Jeugt, Zürich, ZH

Subject Area
Software Engineering - Core Systems