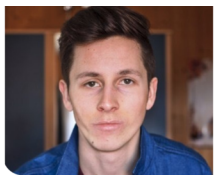
Haskell Substitution Stepper

An equational reasoning assistant for teaching and debugging

Graduate



Robin Elvedi

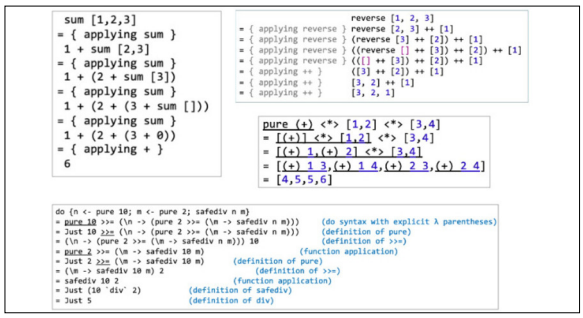
Dominik Dietler

Definition of Task: 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.The functional programming paradigm does not have the concept of a program counter or internal state. Executing a program in a functional programming language is typically viewed as evaluating an expression using repeated substitution as seen in the image.
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.
Hence the main aim of this project is to implement a substitution stepper for the functional programming language Haskell that can be used to visualise the execution of a functional program.

Approach: The first Proof of Concept was built on a simplified subset of the Haskell abstract syntaxt tree. Further research into 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 switch also enabled a closer coupling to GHC and the output of the resulting prototype proved to be more readable and user friendly.

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

Examples of Substitution Steps
Aufgabenstellung von F. Mehta



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

Haskell Core Intermediate Language
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
```

Output of Substitution Stepper
Own presentment

```
1  reverseList [1, 2, 3]
2
3  {- skipping 2 substeps -}
4
5  {- Replace with matching pattern -}
6  (reverseList [2, 3]) ++ [1]
7
8  {- skipping 20 substeps -}
9
10 {- Replace with matching pattern -}
11 3 : (([] ++ [2]) ++ [1])
12
13 {- reduction complete - reduce constructor arguments for better
   ↪  visualization -}
14 [3, 2, 1]
15
16 {- reduction completed successfully -}
```


Advisor
Prof. Dr. Farhad D. Mehta

Co-Examiner
Jasper Van der Jeugt, Zürich, ZH

Subject Area
Software Engineering - Core Systems