Evaluating Haskell expressions in a tutoring environment

Tim Olmer

Bastiaan Heeren

Johan Jeuring

Technical Report UU-CS-2014-021
September 2014

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Evaluating Haskell expressions in a tutoring environment

Tim Olmer Bastiaan Heeren
Open University of the Netherlands
Faculty of Management, Science and Technology
Heerlen, The Netherlands
tim.olmer@gmail.com bhr@ou.nl

Johan Jeuring
Universiteit Utrecht
Department of Information and Computing Sciences
Utrecht, The Netherlands
J.T.Jeuring@uu.nl

A number of introductory textbooks for Haskell use calculations right from the start to give the reader insight into the evaluation of expressions and the behavior of functional programs. Many programming concepts that are important in the functional programming paradigm, such as recursion, higher-order functions, pattern-matching, and lazy evaluation, can be partially explained by showing a stepwise computation. A student gets a better understanding of these concepts if she performs these evaluation steps herself. Tool support for experimenting with the evaluation of Haskell expressions is currently lacking. In this paper we present a prototype implementation of a stepwise evaluator for Haskell expressions that supports multiple evaluation strategies, specifically targeted at education. Besides performing evaluation steps the tool also diagnoses steps that are submitted by a student, and provides feedback. Instructors can add or change function definitions without knowledge of the tool's internal implementation. We discuss some preliminary results of a small survey about the tool.

1 Introduction

Many textbooks that introduce the functional programming language Haskell begin with showing calculations that illustrate how expressions are evaluated, emphasizing the strong correspondence with mathematical expressions and the property of referential transparency of the language. For instance, Hutton presents calculations for the expressions *double* 3 and *double* (*double* 2) on page 1 of his textbook [12], and Bird and Wadler show the steps of reducing *square* $(3+4)$ on page 5 of their book [3]. Similar calculations can be found in the first chapter of Thompson's The Craft of Functional Programming [28] and Hudak's The Haskell School of Expression. Textbooks that do not show these calculations [23, 18] compensate for this by giving lots of examples with an interpreter.

Stepwise evaluating an expression on a piece of paper can give a student a feeling for what a program does [4]. However, there is no simple way to view intermediate evaluation steps for a Haskell expression. In this paper we present a prototype implementation of the Haskell Expression Evaluator (HEE) that can show evaluation steps, and lets students practice with evaluating expressions on their own by providing feedback and suggestions (see Figure 1).[1] The tool supports multiple evaluation strategies and can handle multiple (alternative) definitions for functions from the prelude. It is relatively easy for instructors to change the granularity of the steps, or to customize the feedback messages that are reported by the tool.

Showing calculations can be a useful approach to let a student better understand some of the central programming concepts behind the programming language Haskell, such as pattern-matching, recursion, higher-order functions, and lazy evaluation. This approach is also used in textbooks on Haskell [12, 3]. For instance, Haskell's lazy evaluation strategy is fundamentally different from other mainstream programming languages (such as C, C++, C# and Java), and the tool can highlight these differences. A novice functional programmer often faces difficulties in understanding the evaluation steps in a lazy

[1] The prototype is available via `http://ideas.cs.uu.nl/HEE/`.

Preliminary Report. Final version to appear in:
TFPIE 2014

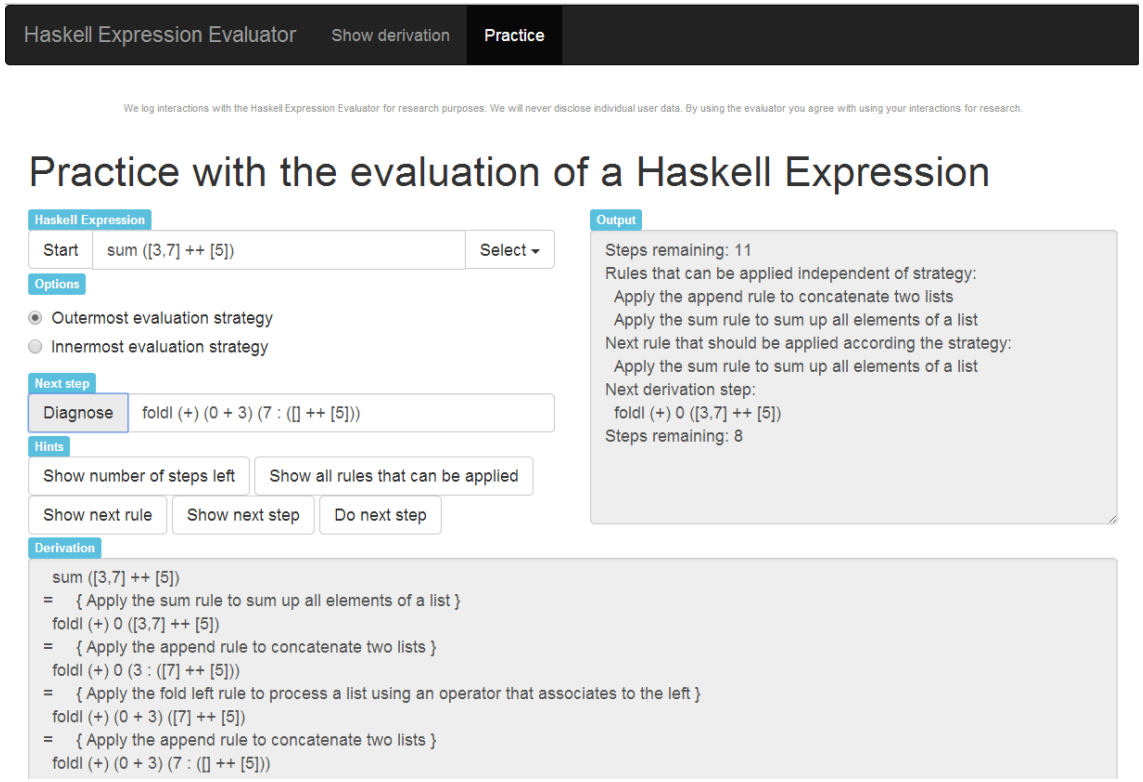

Figure 1: Prototype screen for student interaction

language, and even more experienced programmers find it hard to predict the space behavior of their programs [2]. Another stumbling block is the very compact syntax that is used in Haskell, which makes it sometimes hard to get an operational view of a functional program [27]. More generally, students often do not clearly understand operator precedence and associativity and misinterpret expressions [14, 15]. Showing evaluation steps can partly alleviate these problems.

Contributions and scope. This paper presents a tool that enables students to practice with evaluation steps for Haskell expressions. A student can not only inspect the evaluation steps of a program, but can also provide evaluation steps as input, which can then be checked against various evaluation strategies. Furthermore, the steps are presented at a level of abstraction typically expected in an educational setting. Practicing with evaluation steps gives a student insight into how certain programming concepts such as recursion, higher-order functions, and pattern matching work. It also gives a student insight into various evaluation strategies.

Our evaluation tool only supports integers, list notation, recursion, higher order functions, and pattern matching. The target audience for the evaluator is students taking an introductory course on functional programming. Another limitation is that only small code fragments are considered, which is sufficient for the intended audience. We assume that the expressions evaluated by the tool are well-typed and do not contain compile-time errors, although we could let the tool check for errors before evaluating.

Related work mainly focuses on showing evaluation steps [17, 25], and does not offer the possibility to let a student enter evaluation steps herself, or presents evaluation steps at a lower level of abstraction, such as the lambda-calculus [26, 1].

Roadmap. The rest of the paper is structured as follows. We start with an example that illustrates different evaluation strategies in Section 2. Next, we define rewrite rules and rewrite strategies for a simple expression language in Section 3, which are used for stepwise evaluating an expression and for calculating feedback. We also discuss how rewrite rules and rewrite strategies can be generated for arbitrary function definitions. Section 4 discusses the prototype in more detail, shows how we present feedback, and describes the results of the survey. We conclude the paper with a discussion on related work (Section 5) and present conclusions and future work (Section 6).

2 An example

We start by demonstrating some evaluation strategies that are supported by the tool. We use the expression *sum* $([3,7] +\!\!+ [5])$ as a running example in the rest of the paper. Because the tool is developed for education, we use list notation when showing expressions and we make associativity explicit. The evaluation steps in our examples are based on the following standard definitions for prelude functions:

$$
\begin{aligned}
&\mathit{sum} :: [\mathit{Int}] \to \mathit{Int}\\
&\mathit{sum} = \mathit{foldl}\ (+)\ 0
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{foldl} :: (a \to b \to a) \to a \to [b] \to a\\
&\mathit{foldl}\ f\ v\ [\,] \quad\ = v\\
&\mathit{foldl}\ f\ v\ (x : xs) = \mathit{foldl}\ f\ (f\ v\ x)\ xs
\end{aligned}
$$

$$
\begin{aligned}
&(+\!\!+) :: [a] \to [a] \to [a]\\
&[\,] \quad\ +\!\!+ ys = ys\\
&(x : xs) +\!\!+ ys = x : (xs +\!\!+ ys)
\end{aligned}
$$

Figure 2 shows two different ways to evaluate the expression *sum* $([3,7] +\!\!+ [5])$. The evaluation steps on the left-hand side of Figure 2 correspond to an outermost evaluation order (call-by-name). After rewriting *sum* into a *foldl*, it is the list pattern in *foldl*'s definition (its third argument) that drives evaluation. The evaluation steps nicely show the accumulating parameter of *foldl* for building up the result, the interleaving of steps for $+\!\!+$ (which produces a list) and *foldl* (which consumes list), and the additions that are calculated at the very end. The evaluation steps on the right-hand side of Figure 2 illustrate the left-most innermost evaluation order (call-by-value), which fully evaluates sub-expression $[3,7] +\!\!+ [5]$ before using *foldl*'s definition. Observe that in contrast to call-by-name evaluation the additions are immediately computed. Also observe that *sum* is immediately rewritten into *foldl*. This might be surprising behavior of an innermost evaluation strategy where arguments are completely evaluated before the function is evaluated. The reason for this behavior lies in the definition of *sum*. The definition of *sum* does not have an explicitly specified parameter, but it applies *foldl* partially. Therefore, the evaluator does not handle the sub-expression $[3,7] +\!\!+ [5]$ as a child of *sum* but as a neighbor of *sum*.

From an educational perspective it is interesting to allow for alternative definitions of prelude functions, e.g. *sum* defined with explicit recursion, or *sum* defined with the strict *foldl'* function. With our tool it is possible to switch between these alternative definitions and to observe the consequences for evaluation.

It is important to keep in mind that the tool is capable of doing more than only showing evaluation steps. The tool also lets students practice with evaluating expressions, and can diagnose intermediate steps, suggest reducible expressions, and provide progress information by showing the number of eval-

$$
\begin{array}{ll}
 & \mathit{sum}\ ([3,7] +\!\!+ [5]) \\
= & \quad \{\ \text{definition } \mathit{sum}\ \} \\
 & \mathit{foldl}\ (+)\ 0\ ([3,7] +\!\!+ [5]) \\
= & \quad \{\ \text{definition } +\!\!+\ \} \\
 & \mathit{foldl}\ (+)\ 0\ (3 : ([7] +\!\!+ [5])) \\
= & \quad \{\ \text{definition } \mathit{foldl}\ \} \\
 & \mathit{foldl}\ (+)\ (0+3)\ ([7] +\!\!+ [5]) \\
= & \quad \{\ \text{definition } +\!\!+\ \} \\
 & \mathit{foldl}\ (+)\ (0+3)\ (7 : ([\,] +\!\!+ [5])) \\
= & \quad \{\ \text{definition } \mathit{foldl}\ \} \\
 & \mathit{foldl}\ (+)\ ((0+3)+7)\ ([\,] +\!\!+ [5]) \\
= & \quad \{\ \text{definition } +\!\!+\ \} \\
 & \mathit{foldl}\ (+)\ ((0+3)+7)\ [5] \\
= & \quad \{\ \text{definition } \mathit{foldl}\ \} \\
 & \mathit{foldl}\ (+)\ (((0+3)+7)+5)\ [\,] \\
= & \quad \{\ \text{definition } \mathit{foldl}\ \} \\
 & ((0+3)+7)+5 \\
= & \quad \{\ \text{applying } +\ \} \\
 & (3+7)+5 \\
= & \quad \{\ \text{applying } +\ \} \\
 & 10+5 \\
= & \quad \{\ \text{applying } +\ \} \\
 & 15
\end{array}
\qquad
\begin{array}{ll}
 & \mathit{sum}\ ([3,7] +\!\!+ [5]) \\
= & \quad \{\ \text{definition } \mathit{sum}\ \} \\
 & \mathit{foldl}\ (+)\ 0\ ([3,7] +\!\!+ [5]) \\
= & \quad \{\ \text{definition } +\!\!+\ \} \\
 & \mathit{foldl}\ (+)\ 0\ (3 : ([7] +\!\!+ [5])) \\
= & \quad \{\ \text{definition } +\!\!+\ \} \\
 & \mathit{foldl}\ (+)\ 0\ (3 : (7 : ([\,] +\!\!+ [5]))) \\
= & \quad \{\ \text{definition } +\!\!+\ \} \\
 & \mathit{foldl}\ (+)\ 0\ [3,7,5] \\
= & \quad \{\ \text{definition } \mathit{foldl}\ \} \\
 & \mathit{foldl}\ (+)\ (0+3)\ [7,5] \\
= & \quad \{\ \text{applying } +\ \} \\
 & \mathit{foldl}\ (+)\ 3\ [7,5] \\
= & \quad \{\ \text{definition } \mathit{foldl}\ \} \\
 & \mathit{foldl}\ (+)\ (3+7)\ [5] \\
= & \quad \{\ \text{applying } +\ \} \\
 & \mathit{foldl}\ (+)\ 10\ [5] \\
= & \quad \{\ \text{definition } \mathit{foldl}\ \} \\
 & \mathit{foldl}\ (+)\ (10+5)\ [\,] \\
= & \quad \{\ \text{applying } +\ \} \\
 & \mathit{foldl}\ (+)\ 15\ [\,] \\
= & \quad \{\ \text{definition } \mathit{foldl}\ \} \\
 & 15
\end{array}
$$

Figure 2: Evaluating $\mathit{sum}\ ([3,7] +\!\!+ [5])$ using the outermost (left-hand side) or innermost (right-hand side) evaluation strategy

uation steps remaining. It is possible to train one particular evaluation strategy, or to allow any possible reduction step.

3 Rewrite rules and strategies

We use IDEAS, a framework for developing domain reasoners that give intelligent feedback [9], for rewriting expressions. Many exercises, such as solving a mathematical equation or a programming exercise, can be solved by following some kind of procedure. A procedure or strategy describes how basic steps may be combined to solve a particular problem. Such a strategy is expressed in an embedded domain-specific language. IDEAS interprets a strategy as a context-free grammar. The sentences of this grammar are sequences of rewrite steps that are used to check if a student follows the strategy. The main advantage of using IDEAS in our tool is that it is a generic framework that makes it possible to define exercises that must be solved using some kind of strategy, and that it provides feedback to a student who is solving an exercise. Feedback is added by means of labels at particular locations in the strategy.

To use the IDEAS framework, we construct three components: the domain of the exercise (an expression datatype), rules for rewriting terms in this domain (the evaluation steps), and a rewrite strategy that

```
data Expr = App Expr Expr     -- application
          | Abs String Expr   -- lambda abstraction
          | Var String        -- variable
          | Lit Int           -- integer

  -- smart constructors
appN      = foldl App         -- n-ary application
nil       = Var "[]"
cons x xs = appN (Var ":") [x,xs]
```

Figure 3: Datatype for expressions

combines these rules. Other components, such as parsing, pretty-printing, and testing expressions for equality, are omitted in this paper.

Figure 3 defines an expression datatype with application, lambda abstraction, variables, and integers, together with some helper functions for constructing expressions. The *Var* constructor is also used to represent datatype constructors (e.g., constructor : for building lists).

3.1 Rewrite rules

We introduce a rewrite rule for each function (and operator) from the prelude. The rewrite rules are based on datatype-generic rewriting technology [21], where rules are constructed using operator $\rightsquigarrow$. This operator takes expressions on the left-hand side and the right-hand side. Based on *sum*'s definition, we define the rewrite rule for *sum* as follows:

```
sumRule :: Rule Expr
sumRule = describe "Calculate the sum of a list of numbers" $
  rewriteRule "eval.sum.rule" $
    Var "sum" ⇝ appN (Var "foldl") [Var "+", Lit 0]
```

Each rule has an identifier (here `"eval.sum.rule"`) that is used for identifying the rewrite step, and optionally also a description for explaining the step. The descriptions of the prelude functions are taken from the appendix of Hutton's textbook [12]. Note that functions such as *describe*, *rewriteRule*, operator $\rightsquigarrow$ and type constructor *Rule* are provided by the IDEAS framework.

The rewrite rule for *foldl*'s definition is more involved since it uses pattern matching. The pattern variables in *foldl*'s definition are turned into meta-variables of the rewrite rule by introducing these variables in a lambda abstraction:

```
foldlRule :: Rule Expr
foldlRule =
  describe "Process a list using an operator that associates to the left" $
  rewriteRules "eval.foldl.rule"
    [λf v x xs → appN (Var "foldl") [f,v,nil] ⇝ v
    ,λf v x xs → appN (Var "foldl") [f,v,cons x xs] ⇝
                    appN (Var "foldl") [f,appN f [v,x],xs]
    ]
```

Combinator	*Description*
$s \mathbin{<\!\star\!>} t$	first s, then t
$s \mathbin{<\!\!>} t$	either s or t
$s \triangleright t$	apply s, or else t
fix f	fixed point combinator
label ℓ *s*	attach label ℓ to s
succeed	always succeeds

Combinator	*Description*
sequence xs	generalizes sequence ($<\!\star\!>$) to lists
alternatives xs	generalizes choice ($<\!\!>$) to lists
repeat s	apply s as long as possible
child n s	apply s to the n-th child
outermost s	apply s at left-most outermost position
spinebu s	apply s to the left-spine (bottom-up)
checkCurrent p	succeeds if predicate p holds

Figure 4: Strategy combinators

The rewrite rules *sumRule*, *foldlRule*, and *appendRule* (for operator ++) have an intensional representation with a left- and right-hand side, which does not only make the rules easier to define, but also lets us generate documentation for the rule, take the inverse of the rule, or alter the matching algorithm for the rule's left-hand side (e.g., to take associativity of an operator into account).

Besides the rewrite rules, we also introduce a rule for the primitive addition function (*addRule*), and a rule for beta-reduction. Adding two integers cannot be defined by a rewrite rule, and therefore we use the function *makeRule* from the IDEAS framework to turn a function of type *Expr* $\rightarrow$ *Maybe Expr* into a value of type *Rule Expr*.

```
addRule :: Rule Expr
addRule = describe "Add two integers" $ makeRule "eval.add.rule" f
  where
    f :: Expr → Maybe Expr
    f (App (App (Var "+") (Lit x)) (Lit y)) = Just $ Lit (x + y)
    f _                                     = Nothing
```

In a similar way we define *betaReduction* :: *Rule Expr* that reduces expressions of the form $(\lambda x \rightarrow e_1)\ e_2$ by using *makeRule* and implementing a capture-avoiding substitution.

3.2 Rewrite strategies

The embedded domain-specific language for specifying rewrite strategies in IDEAS defines several generic combinators to combine rewrite rules into a strategy [9]. We briefly introduce the combinators that are used in this paper. The combinators are summarized in Figure 4.

Rewrite rules are the basic building block for composing rewrite strategies. All strategy combinators in the IDEAS framework are overloaded and take rules or strategies as arguments. The sequence combinator ($<\!\star\!>$) specifies the sequential application of two strategies. The choice combinator ($<\!\!>$) defines that either the first operand or the second operand is applied: combinator *alternatives* generalizes the choice combinator to lists. The left-biased choice combinator ($\triangleright$) only tries the second strategy if the first strategy fails. Combinator *repeat* is used for repetition: this combinator applies its argument strategy as often as possible. The fixed point combinator *fix* is used to explicitly model recursion in the strategy. It takes as argument a function that maps a strategy to a new strategy. We can use labels at any position in the strategy to specialize the feedback that is generated.

The strategy language supports all the usual traversal combinators such as *innermost* and *oncebu* [30]. The strategy *child n s* applies strategy s to the n-th child and can be used to define other generic traversal

```
isApp :: Expr → Bool
isApp (App _ _) = True
isApp _         = False

isFun :: String → Int → Expr → Bool
isFun fn 0 (Var s)     = fn == s
isFun fn n (App f _) = isFun fn (n − 1) f
isFun _ _ _          = False
```

Figure 5: Predicates on expressions (helper functions)

combinators. Combinator *checkCurrent* takes a predicate and only succeeds if the predicate holds for the current expression.

An evaluation strategy defines in which order sub-expressions are reduced. We can use the standard left-most outermost (or innermost) rewrite strategy to turn the rewrite rules into an evaluation strategy:

```
rules :: [Rule Expr]
rules = [sumRule, foldlRule, appendRule, addRule, betaReduction]

evalOutermost :: LabeledStrategy (Context Expr)
evalOutermost = label "eval.outer" $
  outermost (alternatives (map liftToContext rules))
```

Note that we use the *Context* type from the IDEAS framework as a zipper [11] datatype for traversing expressions. A zipper maintains a sub-expression that has the focus, and is used by traversal combinators such as *outermost*. We lift the rules to the *Context* type with the function *liftToContext* :: *Rule a* → *Rule* (*Context a*) that applies the rule to the sub-expression that currently has the focus.

The attentive reader will have noticed that the *evalOutermost* strategy does not result in the evaluation that is shown on the left-hand side of Figure 2. Evaluation of a *foldl* application is driven by pattern matching on the function's third argument (the list). The expression at this position should first be evaluated to weak-head normal form (whnf), after which we can decide which case to take. We define a rewrite strategy that first checks that *foldl* is applied to exactly three arguments, then brings the third argument to weak-head normal form, and finally applies the rewrite rule for *foldl*. We need a strategy that can evaluate an expression to weak-head normal form, and pass this as an argument to the strategy definition.

```
foldlS :: Strategy (Context Expr) → LabeledStrategy (Context Expr)
foldlS whnf = label "eval.foldl" $
      checkCurrent (isFun "foldl" 3)   -- check that foldl has exactly 3 arguments
  <*> arg 3 3 whnf                     -- bring the third argument (out of 3) to whnf
  <*> liftToContext foldlRule          -- apply the rewrite rule for foldl's definition
```

The predicate *isFun* (defined in Figure 5) tests whether an expression is a function application of a specific function with an exact number of arguments. We define the strategy combinator *arg i n s*, used in the definition of *foldlS*, to apply strategy *s* to the *i*-th argument of a function application with *n* arguments. For the last argument ($i == n$), we apply *s* to the second sub-expression of an application. Otherwise, we visit the first sub-expression and call *arg* recursively. The combinator is not defined for $i > n$.

```
arg :: Int → Int → Strategy (Context a) → Strategy (Context a)
arg i n s | i == n = child 1 s
          | i < n  = child 0 (arg i (n − 1) s)
```

We also give the evaluation strategy for the definition of the append function, to emphasize its similarity with the definition of *foldlS*. Other function definitions have a similar evaluation strategy.

```
appendS :: Strategy (Context Expr) → LabeledStrategy (Context Expr)
appendS whnf = label "eval.append" $
      checkCurrent (isFun "++" 2)   -- check that append (++) has exactly 2 arguments
  <*> arg 1 2 whnf                  -- bring the first argument (out of 2) to whnf
  <*> liftToContext appendRule      -- apply the rewrite rule for append's definition
```

Evaluating an expression to weak-head normal form is a fixed-point computation over the evaluation strategies for the definitions, since each definition takes the *whnf* strategy as an argument. We combine the evaluation strategies with the rewrite rule for beta-reduction, apply it to the left-spine of an application (in a bottom-up way), and repeat this until the strategy can no longer be applied. This brings the expression in weak-head normal form.

```
prelude :: [Strategy (Context Expr) → LabeledStrategy (Context Expr)]
prelude = [sumS, foldlS, appendS, addS]

spinebu :: Strategy (Context Expr) → Strategy (Context Expr)
spinebu s = fix $ λx → (checkCurrent isApp <*> child 0 x) ▷ s

evalWhnf :: LabeledStrategy (Context Expr)
evalWhnf = label "eval.whnf" $ fix $ λwhnf →
  repeat (spinebu (liftToContext betaReduction <|> alternatives [f whnf | f ← prelude]))
```

The *repeat* combinator in the definition of *evalWhnf* applies its argument strategy zero or more times. For example, the strategy does not have to be applied for an expression that already is in weak-head normal form, and is applied twice to rewrite the expression $(id\ id)\ 3$ into $id\ 3$ and then 3 (where *id* is Haskell's identity function).

The result of applying *evalWhnf* to the expression $sum\ ([3,7] ++ [5])$ gives the calculation shown in Figure 2 on the left-hand side. To fully evaluate a value (such as a list), we have to repeat the *evalWhnf* strategy for the sub-parts of a constructor.

3.3 User-defined function definitions

From an educational viewpoint it is interesting to experiment with different function definitions for the same function and to observe the implications. For example, the function *sum* can be defined using *foldl*, *foldr*, the strict $foldl'$, or as an explicit recursive definition. This feature is also interesting for instructors who would like to incorporate the tool into their course on functional programming because they can then use their own examples to explain programming concepts to their students.

Multiple functions definitions can be added by giving each definition a different name (for instance, *sum*, sum', and sum'') and by manually adding rewrite rules and evaluation strategies for these definitions. However, this approach has some drawbacks. To define rewrite rules and evaluation strategies, the maintainer of the prototype (probably the instructor) needs to have knowledge of the datatype for

```
data Pat  = PCon String [Pat]      -- pattern constructor
          | PVar String            -- pattern variable
          | PLit Int               -- integer pattern

data Def = Def String [Pat] Expr   -- function definition
```

Figure 6: Datatype for patterns and function definitions

Haskell expressions and of the internal implementation to construct rules and evaluation strategies. After adding definitions, the prototype needs to be recompiled, and hence a maintainer also needs a complete build environment (Haskell compiler and libraries used). Another disadvantage is that the translation of function definitions to rewrite rules is a manual, error-prone process.

Function definitions have rewrite rules and evaluation strategies of a similar structure. An alternative approach to manually adding rewriting strategies is therefore to generate rewrite rules and evaluation strategies from function definitions. These function definitions can be defined in a Haskell source file, and using annotations [7] we can add a description to every function definition. Annotations are multi-line comments, so the file remains a valid Haskell source file. Annotations have an identifier, for instance DESC for description, which is used by the prototype to interpret the string after DESC as the rule description. With this approach there is a single file, located on the web server, that contains the function definitions, and their corresponding descriptions. No knowledge of the tool's internal implementation is required to add or change function definitions. An example of such a file is given below:

```
{-# DESC sum defined with a foldr to sum up all elements of a list. #-}
sum' = foldr (+) 0

{-# DESC sum defined recursively to sum up all elements of a list. #-}
sum'' []     = 0
sum'' (x:xs) = x + sum'' xs

{-# DESC double function to double a number. #-}
double x = x + x
```

We name this file the evaluator's *prelude* because it defines the functions that can be used in the evaluator. Notice that primitive functions (such as the operator $+$) cannot be defined in the prelude file. Primitive functions are functions that cannot be implemented directly in Haskell and are provided natively by the compiler.

The function definitions given in the example prelude file use pattern matching. The left-hand side of the equal sign consists of the function name and zero or more patterns. If the function name and the patterns match with a particular expression the expression is rewritten to the expression at the right-hand side of the equal sign (after substituting the pattern variables). Figure 6 defines a datatype for patterns and function definitions.

A function definition can have multiple function bindings. For example, the function *foldl* has a binding for the empty list and the non-empty list. In the remainder of the paper we will only consider function definitions with one binding. Definitions with more bindings can be supported by combining generated rules and strategies. An evaluation strategy is generated for each function binding, and these strategies are tried in order of appearance in the corresponding definition. For this, we use the strategy combinator $\triangleright$ from the IDEAS framework.

Generation of rewrite rules. We need a function that takes a rule identifier (that may contain a description) and a function definition and turns these into a rewrite rule for expressions.

$$genRule :: Id \to Def \to Rule\ Expr$$

Rewrite rules are constructed with operator $\rightsquigarrow$ that expects two expressions as its operands. Hence, we need a function to convert the patterns on the left-hand side of a definition into an expression.

$$\begin{aligned}
&patToExpr :: Pat \to Expr \\
&patToExpr\ (PCon\ s\ ps) = appN\ (Var\ s)\ (map\ patToExpr\ ps) \\
&patToExpr\ (PVar\ s) \quad\ = Var\ s \\
&patToExpr\ (PLit\ n) \quad\ = Lit\ n
\end{aligned}$$

Note that the pattern variables of a function definition act as meta-variables in the rewrite rule, and the corresponding variables on the right-hand side have to be substituted when the rewrite rule is applied. We omit the definition of *genRule* that uses *patToExpr*.

Generation of evaluation strategies. We can generate an evaluation strategy for a function definition used in the outermost evaluation strategy based on pattern matching. Such an evaluation strategy consists of three steps: check the name of the function, bring some arguments to weak-head normal form, and apply the definition's rewrite rule. We first focus on the second step of bringing arguments to weak-head normal form and define strategy combinator *args* for applying a list of strategies to the arguments of a function application. We reuse combinator *arg*:

$$\begin{aligned}
&args :: [Strategy\ (Context\ a)] \to Strategy\ (Context\ a) \\
&args\ xs = sequence\ [arg\ i\ (length\ xs)\ s \mid (i,s) \leftarrow zip\ [1..]\ xs]
\end{aligned}$$

The function *patS* constructs an evaluation strategy for a given pattern. For pattern variables, nothing has to be evaluated. For a pattern constructor, we bring the expression to weak-head normal form, check the name of the constructor, and then recursively deal with the constructor's sub-patterns. For a literal pattern, we bring the expression to weak-head normal form and then check whether it is the same number.

$$\begin{aligned}
&patS :: Strategy\ (Context\ Expr) \to Pat \to Strategy\ (Context\ Expr) \\
&patS\ whnf\ (PCon\ s\ ps) = whnf \mathbin{<\!\!*\!\!>} patListS\ whnf\ s\ ps \\
&patS\ whnf\ (PVar\ s) \quad\ = succeed \\
&patS\ whnf\ (PLit\ n) \quad\ = whnf \mathbin{<\!\!*\!\!>} checkCurrent\ (==Lit\ n)
\end{aligned}$$

$$\begin{aligned}
&patListS :: Strategy\ (Context\ Expr) \to String \to [Pat] \to Strategy\ (Context\ Expr) \\
&patListS\ whnf\ s\ ps \;=\; checkCurrent\ (isFun\ s\ (length\ ps)) \\
&\qquad\qquad \mathbin{<\!\!*\!\!>} args\ (map\ (patS\ whnf)\ ps)
\end{aligned}$$

Function *genEvalStrat* takes a rule identifier, a function definition and the weak-head normal form strategy and returns an evaluation strategy for that function definition. First, arguments are evaluated by *patListS*. Second, the generated rewrite rule for the definition is applied.

$$\begin{aligned}
&genEvalStrat :: Id \to Def \to Strategy\ (Context\ Expr) \to Strategy\ (Context\ Expr) \\
&genEvalStrat\ rId\ (Def\ s\ ps\ e)\ whnf \;=\; patListS\ whnf\ s\ ps \\
&\qquad\qquad \mathbin{<\!\!*\!\!>} liftToContext\ (genRule\ rId\ (Def\ s\ ps\ e))
\end{aligned}$$

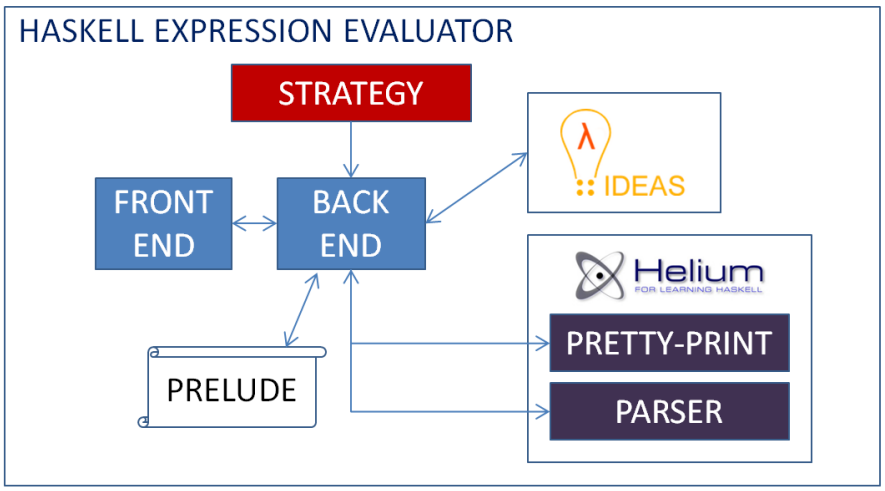

Figure 7: Component diagram of the prototype

The generated evaluation strategy defined above does not behave correctly for multiple or nested patterns. The strategy $s \mathbin{<\!\star\!>} t$ only succeeds if both s and t succeed. In the context of an evaluation strategy this is undesirable because the effect of partly evaluating expressions with strategy s is undone (and cannot be observed) if strategy t fails. The solution is to replace all occurrences of $<\!\star\!>$ (and derived combinators such as *sequence*) in the generated strategy by a new combinator $<\!\star$, which we define as:

$$(<\!\star) :: \mathit{Strategy}\ a \to \mathit{Strategy}\ a \to \mathit{Strategy}$$
$$s <\!\star\ t = s \mathbin{<\!\star\!>} (t \triangleright \mathit{succeed})$$

4 Prototype

The prototype for practicing with evaluation steps is divided into a number of separate components. Figure 7 shows the component model of the prototype: it consists of a front-end, a back-end, and a strategy component. The strategy component contains all rewrite rules and rewrite strategies for a certain evaluation strategy. The back-end component uses the external components IDEAS and Helium, and reads the prelude file with function definitions that resides on the server. This paper focuses on the back-end and strategy components.

In the future we hope to integrate the evaluator with the ASK-ELLE programming tutor [7] for learning Haskell. This tutor supports students in solving introductory programming exercises by providing hints on how to continue, and by diagnosing intermediate programs that are submitted by a student. Having a stepwise evaluator in the ASK-ELLE environment is promising because it allows students to see their programs being evaluated, and to pinpoint mistakes in their definitions (e.g. by evaluating a counter-example). ASK-ELLE is also build on top of the IDEAS framework and the Helium compiler. Decomposing the evaluator into components, some of which are shared with ASK-ELLE, should make future integration easier.

Front-end. The front-end is web-based and written in HTML and JavaScript. It uses JSON to communicate with the back-end. It provides an interface to inspect the evaluation of a Haskell expression or to

practice with the evaluation of a Haskell expression. The prototype front-end can easily be replaced by another front-end and the purpose of this prototype front-end is to show what kind of IDEAS services [8] can be used.

A user can select an example Haskell expression by clicking on the 'Select' button (see Figure 1) or she can enter a Haskell expression. The prototype currently only supports a subset of the Haskell syntax, so it is possible that this operation fails. After typing or selecting a Haskell expression the user can choose between the innermost evaluation strategy and the outermost evaluation strategy. The user can now call several standard IDEAS services such as the service for calculating the number of steps left, getting information about the next rule that should be applied, or finding out what the result is of applying the rule. The user can fill in the next evaluation step, possibly with the help of the services, and click on the button 'Diagnose' to see if the provided next step is the correct step according to the strategy.

The string representation of a rule is used by the feedback service that gives information about the next rule that should be applied. This string representation can be modified in a script file [8] where rule identifiers are mapped to textual representations. Every rewrite rule has an identifier. For example, the identifier of the *foldl* rewrite rule is `"eval.foldl.rule"` (Section 3.1). This identifier is mapped to 'Apply the fold left rule to process a list using an operator that associates to the left' in the script file. This script file can be changed without recompiling the evaluator, which makes it possible to easily adapt the information, for example to support feedback in another language.

Back-end. The back-end is developed in Haskell, and uses the Helium compiler [10] for parsing and pretty-printing expressions. The back-end operates as a glue component that connects all other components. It receives a string from the front-end, uses the Helium compiler to parse the string, and converts the abstract syntax tree produced by Helium to the expression datatype. The back-end receives expression results from IDEAS, converts values from the expression datatype back to Helium's expression datatype, and uses the Helium compiler to convert this value to a string. This string is presented to the user using the front-end.

To determine if a provided step follows the evaluation strategy, the IDEAS framework is instantiated with functionality for determining if the provided expression is equal to the expected expression. The evaluator therefore needs to implement two functions that are required by the framework: one function to determine if two expressions are semantically equivalent, and one function to determine if two expressions are syntactically equivalent. Syntactic equivalence is obtained by deriving an instance of the *Eq* type class for the *Expr* datatype. Semantic equivalence is more subtle because a student may submit an expression that is syntactically different from the expression we expect to get from the step, but semantically the same. Semantic equivalence is defined by using a function that applies the rewrite strategy until it cannot be applied any further and returns *True* if both results are the same. With these two functions we can also spot mistakes in rewrite steps, even when an incorrect expression happens to produce the same result.

Survey. We carried out a small survey to find out if the tool has potential in an educational setting. We invited instructors of functional programming courses at several universities, primarily but not only in the Netherlands, and students that follow or completed the functional programming course at the Open Universiteit to experiment with the prototype, and to answer open-ended and closed-ended questions about the tool. All participants were approached via email on April 30, 2014. By May 12, 7 instructors (out of 8) and 9 students (out of 29) completed the survey. All questions and answers are included in the first author's master's thesis [22]. We discuss the most important observations.

All participants agree that it is useful for students following an introductory course in functional programming to inspect how an expression is evaluated. Eleven participants agree that it is also useful to inspect multiple evaluation strategies. One student argued that the examples in Hutton's textbook [12] are sufficient to get a good understanding of the evaluation steps, and two participants argued that inspecting evaluation steps according to multiple evaluation strategies may be confusing for those who start programming for the first time. There is less consensus about the question if more evaluation strategies should be supported. Four instructors argue that lazy evaluation should be supported because it is very subtle, but two other instructors argue that this is too much for the basic understanding of programming concepts. Another instructor thinks it would be nice to add lazy evaluation, but is not sure if this is worth the effort. Students find the addition of lazy evaluation useful.

Practicing with the evaluation steps was well received by all participants. Three participants argue that it is only useful at the very start, and that students can quickly become bored by entering all evaluation steps. Another remark from an instructor is that studying evaluation steps is not the only way to reason about a program. To ensure that students do not get bored quickly it might be useful to skip certain evaluation steps. However, participants disagree about whether or not this feature should be supported. Five instructors and one student argue that it should not be possible to skip steps to illustrate that each step is necessary. The other participants would like to see this feature added to the prototype.

Nine participants find it possibly useful to add user-defined function definitions and to adjust function definitions to be able to inspect the effect of these changes on the evaluation of expressions. Three participants do not find this useful and think support for prelude functions is sufficient. Some participants suggest to also show the function definitions. This could help students in understanding the evaluation steps. Another suggestion is to show a derivation according to multiple evaluation strategies side by side, so that the student can easily observe the differences. Some participants also suggest to detect if an expression cannot be evaluated according to the innermost strategy and notify the user accordingly.

It is clear that instructors not always agree upon desirability of features. Therefore, it is important that features are configurable for instructors, and that an instructor can adapt the tool to her own course.

5 Related work

There are roughly three approaches to inspect the evaluation steps of a Haskell expression: trace generation, observing intermediate data structures, and using rewrite rules. The central idea of the trace generation approach, which is mainly used for debugging, is that every expression is transformed into an expression that is supplemented with a description in the trace. The trace information is saved in a datatype that can be viewed by a trace viewing component. There are two methods for trace generation. The first method is to instrument Haskell source code. Pure Haskell functions are transformed to Haskell functions that store the evaluation order in a certain datatype that can be printed to the user. This approach is used to show complete traces in so called redex trail format [27], and this is also the approach used in the Hat debug library [5]. An advantage of this method is that it is completely separated from the compiler, so it does not matter which compiler is used. A disadvantage of this method is that the instrumentation of the original code can alter the execution of the program. The second method, which is used for example by WinHIPE [24], is to instrument the interpreter. The advantage of this method is that the execution of the program is exactly the same, but a disadvantage is that the interpreter (part of the compiler) needs to be adjusted. The approach to observe intermediate data structures, which is also mainly used for debugging, is used in the Hood debugger [27]. The approach to specify rewrite rules to inspect the evaluation of expressions is used for example in the stepeval project where a subset

of Haskell expressions can be inspected [20]. With the above approaches it is possible to inspect the evaluation steps of a Haskell expression, but it is not possible to practice with these evaluation steps.

Several intelligent tutoring systems have been developed that support students with learning a functional programming language. One of the main problems for novice programmers is to apply programming concepts in practice [16]. To keep students motivated to learn programming it is therefore important to teach it incrementally, to practice with practical exercises, and to give them early and direct feedback on their work [29]. The main advantage of an intelligent tutoring system is that a student can get feedback at any moment. An intelligent tutoring system consists of an inner loop and an outer loop. The main responsibility of the outer loop is to select an appropriate task for the student; the main responsibility of the inner loop is to give hints and feedback on student steps. The Web-Based Haskell Adaptive Tutor (WHAT) focuses more on the outer loop. It classifies each student into a group of students that share some attributes and will behave differently based on the group of the student [19]. With WHAT, a student can practice with three kinds of problems: evaluating expressions, typing functions, and solving programming assignments. A disadvantage of this tutor is that it does not support the stepwise development of a program. ASK-ELLE [7] is a Haskell tutor system that focuses primarily on the inner loop. Its goal is to help students learn functional programming by developing programs incrementally. Students receive feedback about whether or not they are on the right track, can ask for a hint when they are stuck, and see how a complete program is constructed stepwise [13].

6 Conclusions and future work

In this paper we have presented a prototype tool that can be used to show the evaluation steps of a Haskell expression according to different evaluation strategies. We support the left-most innermost evaluation strategy and an outermost evaluation strategy based on pattern matching. The tool can also be used to practice with evaluation steps. This prototype may help students to better understand important programming concepts such as pattern matching, higher-order functions, and recursion. How effective the tool is in practice needs to be further investigated by collecting more empirical data.

The evaluation process is driven by the definition of the rewrite rules and the evaluation strategy that is used. The rewrite rules and evaluation strategies for the definitions can be generated by the prototype from a set of function definitions. The extension of adding user-defined function definitions makes it possible for users to easily get an understanding of how their function will be evaluated. Another advantage is that alternative definitions for prelude functions can be tried, and that the results can be inspected. For example, a student can use a strict version of *foldl* for *sum*, or define *sum* recursively. At the moment, adding user-defined functions requires an instructor to change the content of the prelude file on the server. Preferably, some support for this feature is added to the front-end.

The expression language in the current tool is limited. In the future, we hope to support other language constructs such as guards, conditional expressions, list comprehensions, algebraic datatypes, etc. Other future work is to offer the possibility to change the step size of a function, and to add sharing to an evaluation strategy:

- In the examples, we use a fixed step size of one. The step size is the number of steps that the evaluator uses to rewrite a certain expression. For example, the expression $3+(4+7)$ is evaluated to 14 in two steps, although most students will typically combine these steps. More research must be carried out to automatically derive or configure a certain step size that suits most students.
- The lazy evaluation strategy used by Haskell combines the outermost evaluation strategy with sharing. Currently, sharing is not supported in the prototype. To help students learn about which

computations are shared, we plan to extend the prototype along the lines of Launchbury's natural semantics for lazy evaluation [17], and by making the heap explicit.

The long-term goal of our work is to integrate the functionality of the prototype in the ASK-ELLE programming tutor, which then results in a complete tutoring platform to help students learn programming. We are also considering to combine the evaluator with QuickCheck properties [6]: when QuickCheck finds a minimal counter-example that falsifies a function definition (e.g. for a simple programming exercise), then we can use the evaluator to explain more precisely why the result was not as expected, or use the evaluator as a debugging tool.

Acknowledgements The authors would like to thank the students and instructors that participated in the survey for their feedback and suggestions. We also thank the anonymous reviewers for their detailed comments.

References

[1] Martin Abadi, Luca Cardelli, Pierre-Louis Curien & Jean-Jacques Lvy (1990): *Explicit Substitutions*. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, ACM, New York, NY, USA, pp. 31–46.

[2] Adam Bakewell & Colin Runciman (2000): *The Space Usage Problem: An Evaluation Kit for Graph Reduction Semantics*. In: *Selected Papers from the 2nd Scottish Functional Programming Workshop (SFP00)*, Intellect Books, Exeter, UK, pp. 115–128.

[3] Richard S. Bird & Philip. Wadler (1998): *Introduction to functional programming using Haskell*. Prentice-Hall.

[4] Manuel M.T. Chakravarty & Gabriele Keller (2004): *The risks and benefits of teaching purely functional programming in first year*. *Journal of Functional Programming* 14(1), pp. 113–123.

[5] Olaf Chitil, Colin Runciman & Malcolm Wallace (2003): *Transforming Haskell for Tracing*. In Ricardo Peña & Thomas Arts, editors: *Implementation of Functional Languages*, *Lecture Notes in Computer Science* 2670, Springer Berlin Heidelberg, pp. 165–181.

[6] Koen Claessen & John Hughes (2000): *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, ACM, New York, NY, USA, pp. 268–279.

[7] Alex Gerdes (2012): *Ask-Elle: a Haskell Tutor*. PhD thesis, Open Universiteit Nederland.

[8] Bastiaan Heeren & Johan Jeuring (2014): *Feedback services for stepwise exercises*. *Science of Computer Programming* 88(0), pp. 110 – 129.

[9] Bastiaan Heeren, Johan Jeuring & Alex Gerdes (2010): *Specifying Rewrite Strategies for Interactive Exercises*. *Mathematics in Computer Science* 3(3), pp. 349–370.

[10] Bastiaan Heeren, Daan Leijen & Arjan van IJzendoorn (2003): *Helium, for learning Haskell*. In: *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, Haskell '03, ACM, New York, NY, USA, pp. 62–71.

[11] Gérard Huet (1997): *The Zipper*. *Journal of Functional Programming* 7(5), pp. 549–554.

[12] Graham Hutton (2007): *Programming in Haskell*. Cambridge University Press.

[13] Johan Jeuring, Alex Gerdes & Bastiaan Heeren (2012): *A Programming Tutor for Haskell*. In Viktória Zsók, Zoltán Horváth & Rinus Plasmeijer, editors: *Central European Functional Programming School*, *Lecture Notes in Computer Science* 7241, Springer Berlin Heidelberg, pp. 1–45.

[14] Aravind K. Krishna & Amruth N. Kumar (2001): *A Problem Generator to Learn Expression: Evaluation in CSI, and Its Effectiveness*. *J. Comput. Sci. Coll.* 16(4), pp. 34–43.

[15] Amruth N. Kumar (2005): *Results from the Evaluation of the Effectiveness of an Online Tutor on Expression Evaluation*. In: *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '05, ACM, New York, NY, USA, pp. 216–220.

[16] Essi Lahtinen, Kirsti Ala-Mutka & Hannu-Matti Järvinen (2005): *A Study of the Difficulties of Novice Programmers*. In: *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '05, ACM, New York, NY, USA, pp. 14–18.

[17] John Launchbury (1993): *A Natural Semantics for Lazy Evaluation*. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, ACM, New York, NY, USA, pp. 144–154.

[18] Miran Lipovaca (2011): *Learn You a Haskell for Great Good!: A Beginner's Guide*, 1st edition. No Starch Press, San Francisco, CA, USA.

[19] Natalia López, Manuel Núñez, Ismael Rodríguez & Fernando Rubio (2002): *What: Web-Based Haskell Adaptive Tutor*. In Donia Scott, editor: *Artificial Intelligence: Methodology, Systems, and Applications*, *Lecture Notes in Computer Science* 2443, Springer Berlin Heidelberg, pp. 71–80.

[20] Ben Millwood (2011): *stepeval library: Evaluating a Haskell expression step-by-step*. Available at `https://github.com/bmillwood/stepeval`.

[21] Thomas van Noort, Alexey Rodriguez Yakushev, Stefan Holdermans, Johan Jeuring, Bastiaan Heeren & José Pedro Magalhães (2010): *A lightweight approach to datatype-generic rewriting*. *Journal of Functional Programming* 20, pp. 375–413.

[22] Tim Olmer (2014): *Evaluation of Haskell expressions in a tutoring environment*. Master's thesis, Open Universiteit Nederland. Available at `http://hdl.handle.net/1820/5389`.

[23] Bryan O'Sullivan, John Goerzen & Don Stewart (2008): *Real World Haskell*, 1st edition. O'Reilly Media, Inc.

[24] Cristóbal Pareja-Flores, Jamie Urquiza-Fuentes & J. Ángel Velázquez-Iturbide (2007): *WinHIPE: An IDE for Functional Programming Based on Rewriting and Visualization*. *SIGPLAN Not.* 42(3), pp. 14–23.

[25] Jan Rochel (2010): *The Very Lazy λ-calculus and the STEC Machine*. In: *Proceedings of the 21st International Conference on Implementation and Application of Functional Languages*, IFL'09, Springer-Verlag, Berlin, Heidelberg, pp. 198–217.

[26] Peter Sestoft (2002): *Demonstrating lambda calculus reduction*. In: *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones, number 2566 in Lecture Notes in Computer Science*, Springer-Verlag, pp. 420–435.

[27] Jan Sparud & Colin Runciman (1997): *Tracing lazy functional computations using redex trails*. In Hugh Glaser, Pieter Hartel & Herbert Kuchen, editors: *Programming Languages: Implementations, Logics, and Programs*, *Lecture Notes in Computer Science* 1292, Springer Berlin Heidelberg, pp. 291–308.

[28] Simon Thompson (2011): *Haskell: The Craft of Functional Programming*, 3rd edition. Addison-Wesley Longman Publishing Co., Inc.

[29] Arto Vihavainen, Matti Paksula & Matti Luukkainen (2011): *Extreme Apprenticeship Method in Teaching Programming for Beginners*. In: *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, ACM, New York, NY, USA, pp. 93–98.

[30] Eelco Visser, Zine-el-Abidine Benaissa & Andrew Tolmach (1998): *Building Program Optimizers with Rewriting Strategies*. In: *ICFP 1998: International Conference on Functional Programming*, pp. 13–26.