Evaluating Haskell expressions in a tutoring environment

Tim Olmer[*] Bastiaan Heeren
Open Universiteit
School of Computer Science
Heerlen, The Netherlands
t.olmer@studie.ou.nl 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. In fact, many programming concepts that are considered to be important by the functional programming paradigm, such as recursion, higher-order functions, pattern-matching, and lazy evaluation, can be explained by showing a step-wise computation. We think that students can get a better understanding of these concepts if they are trained to perform these evaluation steps on their own. We also think that tool support is lacking for experimenting with the evaluation of Haskell expressions. In this paper we present a prototype implementation of a step-wise evaluator for Haskell expressions that supports multiple evaluation strategies, and which is specifically targeted at education. Besides performing these evaluation steps, the tool can also diagnose steps that are submitted by students and provide feedback.

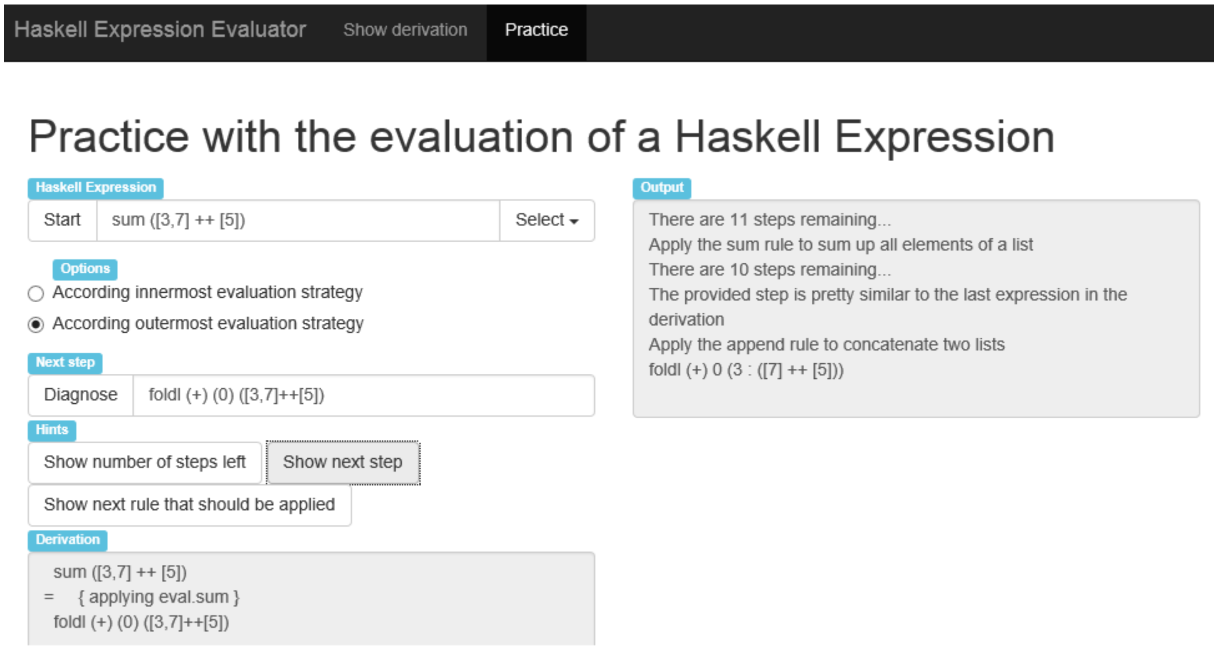
1 Introduction

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 [11], 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 [26] and Hudak's The Haskell School of Expression. Textbooks that do not show these calculations [21, 17] compensate for this by giving lots of examples with an interpreter.

Step-wise evaluating an expression on a piece of paper can give students a feeling for what a program does [4]. However, there is no simple way to view intermediate evaluation steps for some Haskell expression. In this paper we present a prototype implementation of a Haskell expression evaluator that can show evaluation steps and that 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 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 students 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 [11, 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.

[*]The work presented in this paper is part of the first author's upcoming master's thesis.
[1]The prototype is available via `http://ideas.cs.uu.nl/HEE/`.


Haskell Expression Evaluator
Show derivation
Practice

Practice with the evaluation of a Haskell Expression

Haskell Expression
Start: sum ([3,7] ++ [5])

Options
( ) According innermost evaluation strategy
(●) According outermost evaluation strategy

Next step
Diagnose: foldl (+) (0) ([3,7]++[5])

Hints
Show number of steps left
Show next step
Show next rule that should be applied

Derivation
```
  sum ([3,7] ++ [5])
=    { applying eval.sum }
  foldl (+) (0) ([3,7]++[5])
```

Output
```
There are 11 steps remaining...
Apply the sum rule to sum up all elements of a list
There are 10 steps remaining...
The provided step is pretty similar to the last expression in the
derivation
Apply the append rule to concatenate two lists
foldl (+) 0 (3 : ([7] ++ [5]))
```

Figure 1: Prototype screen for student interaction

Novice functional programmers often face difficulties in understanding the evaluation steps in a lazy 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 [25]. More generally, students often do not clearly understand operator precedence and associativity and misinterpret expressions [13, 14]. Showing evaluation steps can partly alleviate these problems.

Contributions and scope. This paper presents an approach based on rewriting that enables students to practice with evaluation steps for Haskell expressions. A student cannot only inspect the evaluation steps of a program, but 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 students insight into how certain programming concepts such as recursion, higher-order functions, and pattern matching work. It also gives students insight in 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 suitable for the intended audience. We assume that the evaluated expressions are well-typed and do not contain compile-time errors.

Related work mainly focuses on showing evaluation steps [16, 23], and does not offer the possibility to let a student input evaluation steps herself, or presents evaluation steps at a lower level of abstraction, such as the lambda-calculus [24, 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 step-wise evaluating an expression and for calculating feedback. Section 4 discusses the prototype in more detail and how we present feedback. 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:

$$\begin{array}{ll}
sum = foldl\ (+)\ 0 & \\
foldl\ \_\ v\ [\,] & = v \\
foldl\ f\ \ v\ (x:xs) & = foldl\ f\ (f\ v\ x)\ xs \\
[\,] \qquad +\!\!+\, ys & = ys \\
(x:xs) +\!\!+\, ys & = x:(xs +\!\!+ ys)
\end{array}$$

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 the additions are immediately computed, this in contrast to call-by-name evaluation. Also observe that *sum* is immediately rewritten into *foldl*. You might not expect this behavior with 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* that is used 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 evaluation steps remaining. It is possible to train one particular evaluation strategy, or to allow any possible reduction step.

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

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

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

3 Rewrite rules and strategies

For rewriting expressions we use IDEAS, a framework for developing domain reasoners that give intelligent feedback [8]. A mathematical equation or a programming exercise is mostly solved by following some kind of procedure. A procedure or strategy describes how basic steps may be combined to solve a particular problem [12]. It is possible to provide such a strategy in an embedded domain-specific language to IDEAS. IDEAS interprets the provided strategy as a context-free grammar. The sentences of this grammar are sequences of rewrite steps that will be used to check if a student follows the strategy. The main advantage of using IDEAS 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 doing the exercise. The feedback is implemented by adding label information to certain locations in the strategy.

To use the IDEAS framework, we need to define three parts: a definition of the domain of the exercise (an expression datatype), rules for rewriting terms in this domain (the evaluation steps), and a rewrite strategy that combines these rules. Other parts, 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,

```
data Expr = App Expr Expr     -- application
          | Abs String Expr   -- lambda abstraction
          | Var String        -- variable
          | Con Int           -- integer
  -- smart constructors
appN      = foldl App         -- n-ary application
nil       = Var "[]"
cons x xs = appN (Var ":") [x, xs]
```

Figure 3: Datatype for expressions

together with some helper-functions for constructing expressions. The variables are 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 [20], where rules are constructed with operator ⇝. 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 "+", Con 0]
```

Each rule has an identifier (here `"eval.sum.rule"`) that is used for explaining the rewrite step, and optionally also a description. The descriptions of the prelude functions are taken from the appendix of Hutton's textbook [11]. 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]
    ]
```

The rewrite rules have an intensional representation with a left- and right-hand side, which does not only make the rules easier to define, but which 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 certain operators into account).

Besides the rewrite rules, we also introduce a rule for the primitive addition function (*addRule*), and a rule for beta-reduction.

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 [8]. We will briefly introduce the combinators that are used in this paper.

Rewrite rules are the basic building block for composing rewrite strategies. All combinators are lifted by means of overloading and take rules or strategies as arguments. The sequence combinator (<*>) 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. Combinator *check* takes a predicate and only succeeds if the predicate holds. Combinator *repeatS* is used for repetition: this combinator will apply 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 also supports all the usual traversal combinators such as *innermost* and *oncebu* [28].

An evaluation strategy defines in which order sub-expression 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 as a zipper [10] datatype for traversing expressions. Hence, we have to lift the rules to the *Context* type.

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" $
        check (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
```

We omit the definition of *arg* and the strategies for the other function definitions (all with the same type as *foldlS*) for reasons of space.

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 these 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.

```
prelude = [sumS, foldlS, appendS, addS]
```

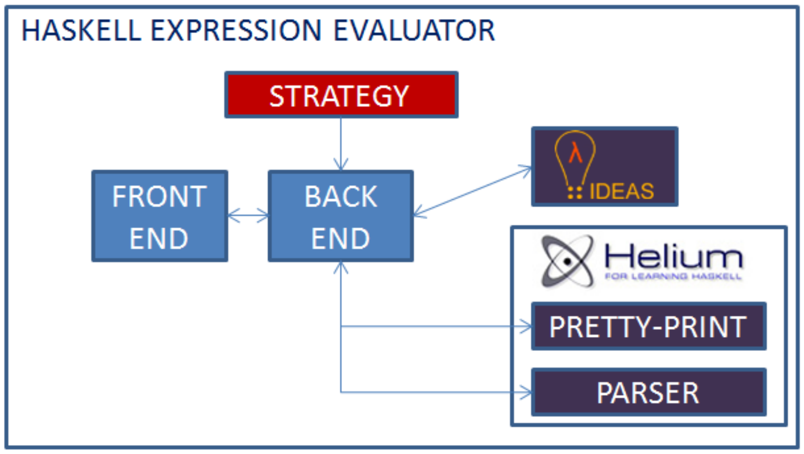


Figure 4: Component diagram of the prototype

$$\begin{array}{l} \mathit{evalWhnf} :: \mathit{LabeledStrategy}\ (\mathit{Context}\ \mathit{Expr}) \\ \mathit{evalWhnf} = \mathit{label}\ \texttt{"eval.whnf"}\ \$\ \mathit{fix}\ \$\ \lambda \mathit{whnf} \rightarrow \\ \quad \mathit{repeatS}\ (\mathit{spinebu}\ (\mathit{liftToContext}\ \mathit{betaReduction} \mathrel{\lhd\!\rhd} \mathit{alternatives}\ [f\ \mathit{whnf} \mid f \leftarrow \mathit{prelude}])) \end{array}$$

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 this strategy for the sub-parts of a constructor.

4 Prototype

The prototype for practicing with evaluation steps we have developed is divided into separate components so they can easily be changed for other components in future releases. This might be useful in the future if we integrate the prototype with a functional programming tutor such as ASK-ELLE [7]. The component model of the prototype is shown in Figure 4 and consists of a front-end, a back-end, and a strategy component. The back-end component uses the external components IDEAS and Helium. This paper focuses on the back-end and strategy components.

The strategy component contains all rewrite rules and rewrite strategies for a certain evaluation strategy. The Helium compiler [9] is used for parsing and pretty-printing expressions. The advantage of choosing the Helium compiler is that this component is already used by the ASK-ELLE tutor, which makes future integration easier. The back-end is developed in Haskell. Since ASK-ELLE and IDEAS are also written in Haskell, this will make integration easier. 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 Helium output to the expression datatype. The back-end receives expression results from IDEAS and converts values from the expression datatype back to the Helium datatype and uses the Helium compiler to convert this value to a string. This string is presented to the user using the front-end.

Student feedback. 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 can be used. A user can select an example Haskell expression by clicking on the 'Select' button (see Figure 1) or she can input 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 or 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 expression looks like after applying this 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.

In the feedback service that gives information about the next rule that should be applied, the string representation of a certain rule is used. The string representation of a rule can be modified in a script file where rule identifiers are mapped to a textual representation. All rewrite rules have an identifier, for example, the identifier of the *foldl* rewrite rule is `"eval.foldl.rule"`. 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 another language.

To determine if a provided step follows the evaluation strategy, the IDEAS framework needs to determine if the provided expression is equal to the expected expression. The evaluator therefore needs to implement two functions: 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 provide a step that is syntactically different from the expected step, but semantically the same. It is defined by using a function that calculates a final answer for the two expressions and returns *True* if both results are the same.

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 will be transformed into an expression that is supplemented with a description in the trace. The trace information will be 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 [25], 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 [22], 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, that is also mainly used for debugging, is used in the Hood debugger [25]. 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 [19]. 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 [15]. 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 [27]. 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 for 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 [18]. 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 learn students 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 [12].

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. The tool can also be used to practice with these evaluation steps. This prototype may help students with better understanding important programming concepts such as lazy evaluation, pattern matching, higher-order functions, and recursion. In the near future (full paper) we have planned to validate the prototype with a short survey that will be distributed under teachers and students that follow an introductory course on functional programming.

The evaluation process is driven by the definition of the rewrite rules and the evaluation strategy that is used. In the near future (full paper) we will extend our prototype to include a way to add user-defined function definitions. We will describe how rewrite rules and rewrite strategies can be derived automatically from such a user-defined function definition. This extension makes it possible for users to easily get an understanding of how their function will be evaluated. Another benefit is that alternative definitions for prelude functions can be tried, and the results can be inspected. For example, a student can use a strict version of *foldl*, or can define *sum* recursively.

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 that is 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 [16], 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.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien & J.-J. Lvy (1996): *Explicit substitutions*.

[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, UK, pp. 115–128.

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

[4] M.M.T. Chakravarty & G. Keller (2004): *The risks and benefits of teaching purely functional programming in first year*. *Journal of Functional Programming* 14(01), 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 & Alex Gerdes (2010): *Specifying Rewrite Strategies for Interactive Exercises*. *Mathematics in Computer Science* 3(3), pp. 349–370.

[9] 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.

[10] Gérard Huet (1997): *The Zipper*. *J. Funct. Program.* 7(5), pp. 549–554.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

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

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.