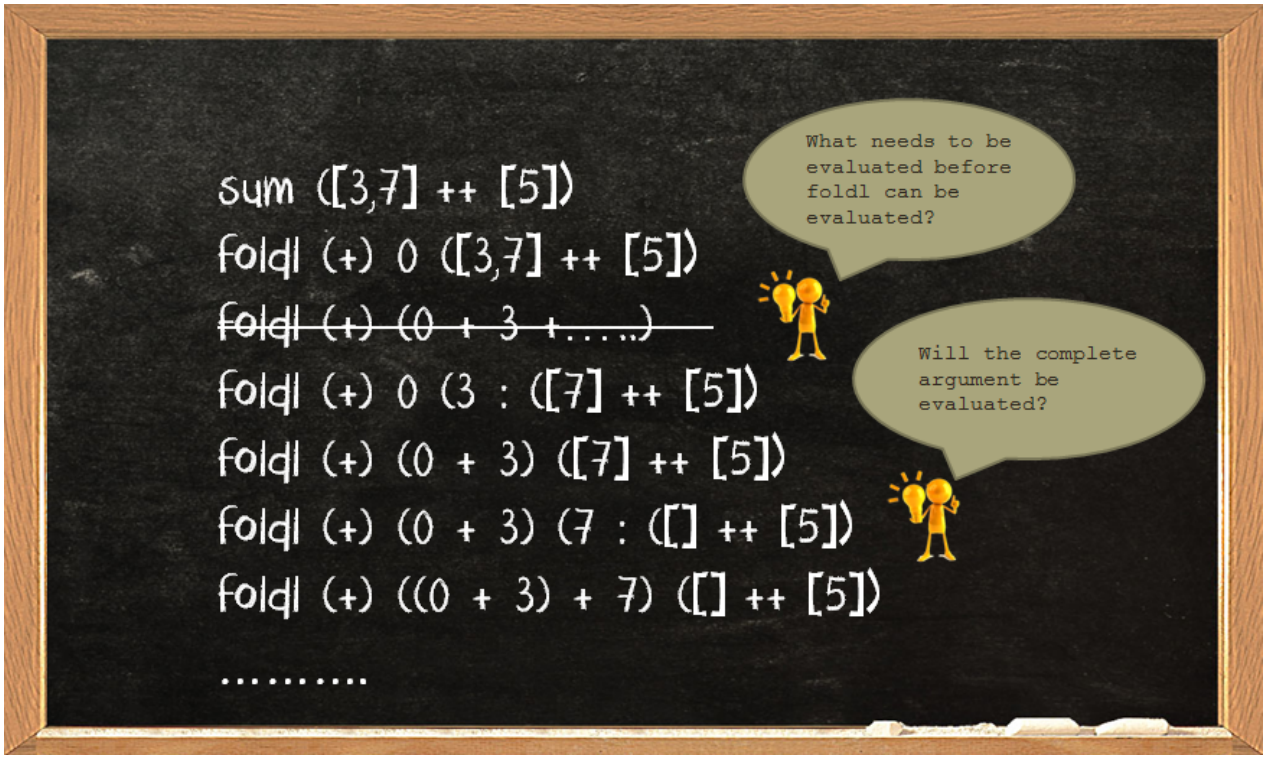
EVALUATION OF HASKELL EXPRESSIONS IN A TUTORING ENVIRONMENT

MASTER THESIS



Author:	Tim Olmer
Student number:	850530187
Date of defence:	17 June 2014

EVALUATION OF HASKELL EXPRESSIONS IN A TUTORING ENVIRONMENT

MASTER THESIS

Author:	Tim Olmer
Student number:	850530187
Date of defence:	17 June 2014
Graduation committee:	prof. dr. J.T. Jeuring (chairman)
	dr. B.J. Heeren (supervisor)

Open Universiteit of the Netherlands
Faculty of Management, Science & Technology

Master Software Engineering
T75317

ABSTRACT

Students that follow an introduction course in functional programming often face difficulties in understanding the evaluation steps of an expression. Inspecting and practicing with the evaluation steps of a Haskell expression can help students in better understanding these evaluation steps. In fact, the stepwise evaluation of expressions can be used to explain programming concepts such as recursion, higher-order functions, pattern matching and lazy evaluation. This thesis describes how students in a tutoring environment can be supported to understand evaluation strategies and programming concepts using the evaluation steps of Haskell expressions. A prototype has been developed of a step-wise evaluator for Haskell expressions. The prototype supports multiple evaluation strategies and besides inspecting the evaluation steps it is possible to practice with evaluation steps and to get feedback.

The prototype uses the IDEAS framework, which is a generic framework for rewriting terms, to rewrite expressions. In order to make use of the IDEAS framework a domain reasoner needs to be defined. A domain reasoner consists of a description of the domain (a data type for Haskell expressions), rewrite rules for this domain and rewrite strategies that specify the order of when a certain rewrite rule must be applied. The evaluation process does not only depend on the evaluation strategy that is used, but also on the function definitions. For the outermost evaluation strategy different arguments (depending on the function definition) must be brought into weak head normal form before a certain rewrite rule can be applied. The IDEAS framework contains feedback services that can be used to diagnose an evaluation step and to give several hints about the next evaluation step. A web front-end has been developed to depict the derivation of a certain expression and to call feedback services. The front-end made it also possible to conduct a small qualitative research (survey) among teachers and students to validate the research.

The survey supports the hypothesis that the prototype can help students to understand evaluation strategies and programming concepts. However, it is important to make the tool configurable for teachers so they can use the tool in their functional programming courses. To incorporate the prototype into their courses it should support the possibility to specify user-defined function definitions. This can be realized by defining a file with function definitions that the prototype can parse. Rewrite rules and evaluation strategies can then be generated from those function definitions. The major improvements to make the prototype more mature are to support the lazy evaluation strategy, to fully support user-defined function definitions and to make all function definitions visible in the front-end.

ACKNOWLEDGMENTS

First of all I would like to thank my supervisor Bastiaan Heeren for his guidance, inspiration, feedback on draft work and his rapid response on my questions. I would also like to thank Johan Jeuring, the chairman of my graduation committee for his critical comments on my work. I would also like to thank them for the possibility of writing a paper together about my topic for the 3rd International Workshop on Trends in Functional Programming in Education that was held on 25 May, 2014 (TFPIE 2014).

The opportunity to present our work on the TFPIE was a great inspiring experience. Therefore I would like to thank all participants of the TFPIE for giving me the opportunity to present our work to them.

The survey among teachers and students that was held as a validation for the research was helpful to support the hypothesis and to pinpoint useful future work. I would like to thank Peter Achten, Koen Claessen, Marko van Eekelen, Alex Gerdes, Jurriaan Hage, Wouter Swierstra, Manuela Witsiers and all anonymous student participants for experimenting with the prototype and providing relevant feedback on the prototype.

I would also like to thank my employer Sioux for making it possible to follow this education.

Last but not least I would like to thank my girlfriend Alicia for her patience and love. I know it was not always easy to live with someone who combines his study with a busy job.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Novice functional programmers often face difficulties in understanding the evaluation steps of an expression in a functional language, and even more experienced programmers find it hard to predict the space behavior of their programs [3]. Many problems that students have in understanding expressions is connected with the priority and associativity rules and the placing of parenthesis [21]. Another source of problems is connected with type expressions where for example students do not recognize a function with more than one argument or do not place parentheses around operator arguments that are functions. Many students that already have experience in an imperative language find the computational model confusing. The computational model in the functional paradigm is based on rewriting instead of memory states and state transitions in the imperative paradigm. Some students use the imperative model in functional programming. For example some students think it is needed to store intermediate results or they expect changes in the value of variables by function application [21]. The compact syntax that is used in Haskell makes it also sometimes hard to get an operational view of a functional program [37].

Many introductory textbooks about functional programming use calculations to illustrate the evaluation steps of expressions [7, 17, 19]. Other introductory textbooks that do not use these calculations compensate for this by giving a lot of examples with an interpreter [32, 26]. Actually understanding a computation and how a certain result was computed can be used to understand different concepts in computer science [34]. The depiction of the step-wise evaluation steps of Haskell expressions can be a useful approach to offer students a better insight into the central programming concepts of Haskell, such as pattern-matching, recursion, higher-order functions and lazy evaluation. This approach is also used in some textbooks on Haskell [19]. Students learning a functional programming language can get a feeling for what a program does by step-wise evaluating an expression on a piece of paper [8]. A disadvantage of this approach is that there is no simple way to view the evaluation steps for some Haskell expression and that there is no direct feedback from a tutor. Learning by doing is very important to learn the basic structures of a programming language. Most students feel that they learn programming best when they do programming by themselves [23].

1.1 RESEARCH QUESTION

This research is concentrated on how students can be given insight into the evaluation steps of a Haskell expression in a tutoring environment. The research question that will be answered in this thesis is:

> **How can students in a tutoring environment be supported to understand evaluation strategies and programming concepts using the evaluation steps of Haskell expressions?**

To accomplish this goal and because current tools lack support for practicing with the step-wise evaluation of Haskell expressions a prototype implementation has been developed of a step-wise evaluator for Haskell expressions. The prototype supports multiple evaluation strategies and besides inspecting the evaluation steps it is possible to practice with evaluation steps and to get feedback.

1.2 CONTRIBUTIONS AND SCOPE

The developed prototype implementation is based on the approach of rewrite and reduction rules to determine the evaluation steps of a Haskell expression. With the prototype a student can inspect the evaluation steps of various Haskell expressions and can also provide his own evaluation step as input. The input is checked against various evaluation strategies and based on this information the prototype can give feedback about input expressions. The prototype can also give some hints to a student. For example, how many steps are left in the derivation or a description of the next evaluation rule that should be applied. The steps are presented at the level of abstraction of Haskell expressions. The prototype supports a call-by-value and call-by-name strategy variant and therefore students can get insight into the implications and differences of using a certain strategy. Practicing with evaluation steps can give students also insight into how certain programming concepts such as recursion, higher-order functions, and pattern matching behave exactly. Another objective is to give students insight in why expressions do not behave as expected. This can be useful for example if a test fails with some expression and the student does not understand why this expression exactly fails. The main testing mechanisms in Haskell, to verify if code functionality operates as expected, are traditional unit testing with the HUnit library and property testing with QuickCheck [32]. Property-based testing is based on a high-level approach to testing where properties are specified for functions. These properties should be satisfy universally. The actual test data is generated for the programmer by QuickCheck. With this approach the functions under test will be tested with thousands of tests that would be infeasible to write by hand. If

a test fails QuickCheck will depict a counter example to which the property fails. By using the shrink option in QuickCheck, QuickCheck tries to reduce the faulty data to the minimum. This reduced faulty data can be used as an input expression in the prototype which enables the student to get insight into why this faulty data fails.

The prototype only supports integers, list notation, recursion, higher-order functions and pattern matching. The target audience for the prototype are students taking an introductory course on functional programming. Another limitation is that only small code fragments are considered. The assumption is made that the evaluated expressions are well-typed and do not contain compile-time errors.

Related work mainly focuses on showing evaluation steps, and does not offer the possibility for a student to enter his own evaluation step [35, 25] or presents evaluation steps at a lower level of abstraction, such as the lambda calculus [36, 1].

1.3 THESIS OVERVIEW

The thesis starts by first describing some background information about functional programming, programming concepts and evaluation strategies in chapter 2. Readers that are familiar with functional programming and the concepts that are considered to be important in functional programming can skip this chapter. Chapter 3 discusses some related work about methods to inspect evaluation steps of a Haskell expression and other intelligent tutoring systems. The research questions, research method and prototype are discussed in more detail in chapter 4. The rewrite rules and strategies that are used to inspect evaluation steps of Haskell expressions are explained in chapter 5. Chapter 6 discusses a method to add user-defined function definitions. To validate the research, a small qualitative research in the form of a survey with open questions has been executed among teachers and students that participate or participated in an introduction course on functional programming. The purpose of the survey is to determine if a tool like the developed prototype can support students in the understanding of programming concepts and evaluation strategies using the evaluation steps of Haskell expressions. The survey is discussed in chapter 7, the complete conducted survey can be found in appendix A and all results on the survey can be found in appendix B. Chapter 8 contains the conclusion where an answer is formulated on the research question. Improvements and suggested future work is also discussed in this chapter.

Notice that some parts of this thesis are re-used from the draft paper “Evaluation of Haskell expressions in a tutoring environment" that was submitted to the 3rd International Workshop on Trends in Functional Programming in Education (TFPIE 2014).

2

BACKGROUND

In this chapter the context of the research will be described. The first paragraph will give a short introduction into functional programming. The second paragraph characterizes the relevance of functional programming in education. The third paragraph discusses some programming concepts that are important in the functional programming paradigm and uses the evaluation steps of expressions to explain those concepts. The last paragraph discusses some evaluation strategies and also uses evaluation steps of expressions to explain those evaluation strategies.

2.1 FUNCTIONAL PROGRAMMING

Haskell is a purely functional programming language that has some fundamental differences with imperative programming languages like C, C++, C# and Java. Imperative languages are based on the idea that a variable is a changeable association between a name and values [29]. A typical imperative program consists of sequences of assignments which change a value of a stored variable. In that sense those languages are closely linked to the underlying hardware [19]. In a functional language a name can only be associated once with a value. A typical functional program is an expression that consists of nested function calls. Because Haskell lacks assignment a function in Haskell can only calculate something and return the result. This means that if a function is called twice with the same arguments, it is guaranteed to return the same result [26]. This property is called referential transparency and makes it possible to delay the evaluation of a sub-expression until it is required for computing the top expression. Because of the property of referential transparency, functions are state independent, can be evaluated in any order, or even parallel without any side-effects. The evaluation strategy that is used by default in Haskell is called lazy evaluation where sub-expressions are only evaluated if they are required for computing the top expression. In imperative programming languages the more familiar eager evaluation strategy is used where all sub-expressions are completely evaluated regardless of whether they are necessary for the final result [24].

2.2 FUNCTIONAL PROGRAMMING IN EDUCATION

Most universities traditionally teach students how to program by using an imperative programming language such as C, C++, C# or Java. However this might not always be the best choice for a first course on programming. An introduction computing course should have three principal aims: teach the elementary techniques of programming (practical aspect), introduce the essential concepts of computing (theoretical aspect) and foster the development of analytic thinking and problem solving skills (methodological aspect) [8].

A functional program consists of a clean reusable software style because a functional program is decomposed into independent functions that glue other functions together [33]. A function output only depends on its input. Advantages of functional programs are therefore that program construction is more modular, less error-prone and more rapid [37] and that the lightweight syntax helps to remove syntactic issues to the background which make it possible to focus on general programming concepts.

An important advantage of using the functional paradigm in teaching programming is that it therefore allows the student and the instructor to focus directly on algorithmic issues and avoiding technical overhead [33]. Most students have some prior experience in programming which results in a very different experience level between students. This makes is hard for a teacher to motivate all students [23]. An advantage in teaching functional programming is that almost no first year students have prior experience in functional programming which results in a more uniform starting level between all students. Functional programming also has a better match with the mathematical background of pre-university students that have no prior programming experience [8]. Those described advantages support the three principal aims mentioned earlier.

2.3 PROGRAMMING CONCEPTS

A **recursive function** is a function which is applied inside its own definition [26]. In imperative languages you can specify how something is computed. New values may be associated with the same name and loop construct such as 'while' and 'for' are used to execute the same code multiple times (for example to iterate through a list) [29]. Because a value in functional programming may only be associated once with a name this is not possible in functional programming and the alternative is to use recursion. Recursion can therefore be seen as the counterpart of the 'while' and 'for' loop constructs in imperative programming. Recursion is an important programming concept in functional programming because computations are defined by declaring what something is rather than specifying how something will be computed.

An example is the definition of the *drop* function that removes the first n elements from a list [19]. The *drop* function is defined in the prelude, a standard module that is imported by default into all Haskell modules. The *drop* function is called with two arguments: the first argument contains the number of elements that must be dropped and the second argument contains the list. The function definition has two base cases which are defined without recursion and one recursive case which results in a recursive nested function call.

$$
\begin{aligned}
&\mathit{drop} && :: \mathit{Int} \rightarrow [a] \rightarrow [a] \\
&\mathit{drop}\ 0\ \mathit{xs} && = \mathit{xs} \\
&\mathit{drop}\ (n+1)\ [] && = [] \\
&\mathit{drop}\ (n+1)\ (x : \mathit{xs}) && = \mathit{drop}\ n\ \mathit{xs}
\end{aligned}
$$

In the *drop* function definition **pattern matching** is used to determine which of the three lines in the function definition must be used. Pattern matching is used to specify patterns to which an expression must conform and to deconstruct the expression according to those patterns [26]. The patterns of a function definition will be checked from top to bottom and from left to right.

In the *drop* function definition the identifier n is a variable name for the number of elements that must be dropped. The identifier *xs* represents the list variable. In Haskell, by convention, the letter s after a variable name means list. The notation $(x{:}xs)$ is used to deconstruct a value of the list data type where x is bound to the first element of a list and *xs* is bound to the tail of that list. The notation _ denotes a wildcard. The notation [] represents the empty list.

If an expression matches a certain pattern in the drop function definition then that expression will be rewritten to the expression that is given at the right-hand side of the function definition. The first pattern in the function definition results in the behaviour that if the drop function is called with the number zero and a list that the result equals the list that is given as the second argument. The second pattern in the function definition results in the behaviour that if the drop function is called with a number n that matches the pattern $(n{+}1)$ and an empty list that the result is the empty list. In other words it is not possible to drop any element from an empty list. The recursive case will result in a nested function call where the number n is decreased by one and the first element of the list is removed. This behaviour will continue until the number is zero or the list is empty. The evaluation steps of the *drop* function with the expression $\mathit{drop}\ 2\ [90, 20, 40, 75]$ is given in figure 2.1 as an example.

In Haskell, functions are first-class values, which means that functions are treated exactly the same as for example numbers. A **higher-order function** is a function that either returns a function as a return value or takes a function as a parameter [26]. The concept of higher-order functions is heavily used

$$
\begin{aligned}
& \mathit{drop}\ 2\ [90, 20, 40, 75] \\
= \quad & \{\ \text{definition } \mathit{drop}\ \} \\
& \mathit{drop}\ 1\ [20, 40, 75] \\
= \quad & \{\ \text{definition } \mathit{drop}\ \} \\
& \mathit{drop}\ 0\ [40, 75] \\
= \quad & \{\ \text{definition } \mathit{drop}\ \} \\
& [40, 75]
\end{aligned}
$$

Figure 2.1: Evaluation steps of drop

$$
\begin{aligned}
& \mathit{map}\ (3*)\ [9, 2] \\
= \quad & \{\ \text{definition } \mathit{map}\ \} \\
& (3*9) : (\mathit{map}\ (3*)\ [2]) \\
= \quad & \{\ \text{applying } *\ \} \\
& 27 : (\mathit{map}\ (3*)\ [2]) \\
= \quad & \{\ \text{definition } \mathit{map}\ \} \\
& 27 : ((3*2) : (\mathit{map}\ (3*)\ [])) \\
= \quad & \{\ \text{applying } *\ \} \\
& 27 : (6 : (\mathit{map}\ (3*)\ [])) \\
= \quad & \{\ \text{definition } \mathit{map}\ \} \\
& [27, 6]
\end{aligned}
$$

Figure 2.2: Evaluation steps of map

in Haskell. For example, in Haskell every function actually takes only one argument. A function that accepts multiple parameters is called **a curried function**. A curried function is a function that always takes exactly one parameter and when it is called with that parameter, it returns a function that takes the next parameter until all parameters are handled [26].

Another example is the *map* function that takes a function and a list and applies that function to every element in the list, producing a new list [26]. The evaluation steps of the *map* function with the expression $\mathit{map}\ (3*)\ [9, 2]$ is given in figure 2.2 as an example.

$$
\begin{aligned}
& \mathit{map} && :: (a \to b) \to [a] \to [b] \\
& \mathit{map}\ \_\ [] && = [] \\
& \mathit{map}\ f\ (x : xs) && = f\ x : \mathit{map}\ f\ xs
\end{aligned}
$$

2.4 EVALUATION STRATEGIES

Functional programming is based on the lambda calculus. It is a formal system where the notation of a computable function is defined [12]. Lambda terms can be viewed as expressions to be calculated. These calculations can be performed using conversions like alpha conversions, beta reductions and/or eta conversions. Alpha conversion allows that bound variable names may be changed. Beta reduction captures the idea of function application. Eta conversion captures the idea that a function which only passes its parameters to another functions can be replaced by this function [5].

A functional program is composed of a set of expressions. An expression that has the form of a function that is applied to one or more arguments is called a reducible expression or redex for short [19]. These expressions can be evaluated by rewriting those expressions according to some beta reduction steps. If a term cannot be reduced any further, it can be considered the result of the computation. This result is called a normal form. There are several evaluation strategies to evaluate or rewrite such an expression. There are two main strategies called **eager evaluation** and **normal-order evaluation.** The evaluation strategies will be described with an example. In this example the expression *powerOfThree* (10 - 4) will be evaluated. The function *powerOfThree* is defined below.

$$\begin{aligned} &\mathit{powerOfThree} \quad :: \mathit{Num}\, a \Rightarrow a \rightarrow a \\ &\mathit{powerOfThree}\, x = x * x * x \end{aligned}$$

Eager evaluation means that the actual parameter is evaluated once at the point of the function call [40]. This strategy is also used in imperative programming languages. The strategy is also called innermost evaluation and call-by-value evaluation. This strategy consists of always choosing the redex that does not contain an other redex. If more than one redex can be evaluated, the redex that begins at the leftmost position in the expression is selected by convention. By using the innermost strategy the arguments of a function are always fully evaluated before the function itself is applied. With this strategy arguments are passed by value. The evaluation steps according to the innermost evaluation strategy of the *powerOfThree* function example are depicted on the next page.

powerOfThree (10 - 4)
= { applying - }
powerOfThree 6
= { applying *powerOfThree* }
6 ∗ 6 ∗ 6
= { applying ∗ }
36 ∗ 6
= { applying ∗ }
216

Normal-order evaluation means that the actual parameter is only evaluated when the argument is actually needed [40]. This strategy is also called outermost evaluation and call-by-name evaluation. This strategy consists of always choosing the redex not contained in any other redex. If more than one redex can be evaluated, the redex that begins at the leftmost position in the expression is selected by convention. With this strategy arguments are passed by name. The advantage of this strategy is that an expression is only evaluated if it is really needed. A disadvantage is that an expression is evaluated multiple times if for example an argument is used more than once in a function. The evaluation steps according to the outermost evaluation strategy of the *powerOfThree* function example are depicted below.

powerOfThree (10 - 4)
= { applying *powerOfThree* }
(10 - 4) ∗ (10 - 4) ∗ (10 - 4)
= { applying - }
6 ∗ (10 - 4) ∗ (10 - 4)
= { applying - }
6 ∗ 6 ∗ (10 - 4)
= { applying ∗ }
36 ∗ (10 - 4)
= { applying - }
36 ∗ 6
= { applying ∗ }
216

The problem that an expression will be evaluated multiple times can be solved by using pointers to the evaluation result. This is called sharing. If outermost evaluation is combined with sharing it is called **Lazy evaluation**. Lazy evaluation is also the evaluation strategy that is used in Haskell. The

evaluation steps according to the lazy evaluation strategy of the *powerOfThree* function example are depicted below. Notice that lazy evaluation, because of the sharing property, never requires more evaluation steps than innermost evaluation.

```
    powerOfThree (10 - 4)
=     { applying powerOfThree }
    x * x * x                    x points to (10 - 4)
=     { applying - }
    x * x * x                    x points to 6
=     { applying * }
    36 * x                       x points to 6
=     { applying * }
    216
```

EXTENDED EXAMPLES To get a better understanding of the evaluation of Haskell expressions and the implications of using different evaluation strategies the evaluation steps of two more extended example expressions are given. The expressions are evaluated according to the outermost evaluation strategy (call-by-name) and the innermost evaluation strategy (call-by-value).

In the first example the expression *sum* $([3,7]$ ++ $[5])$ will be evaluated. In this example the function definitions *sum*, *foldl* and $(++)$ that are defined in the prelude are used [19].

The *foldl* (fold left) function processes a list using an operator that associates to the left. The first argument is the operator (binary function) that will be applied to every element of a list, the second argument is the starting value (accumulator) and the third argument is the list that must be processed. The *foldl* function reduces a list to a single value [26]. The first pattern in the function definition specifies that if the third argument is an empty list, the accumulator is used as the result. The second pattern in the function definition of *foldl* specifies that a *foldl* function will be rewritten into another *foldl* function where the second argument is the result of the operator that is applied on the first argument on the list and the accumulator and the third argument is the tail of the list.

```
foldl              :: (a → b → a) → a → [b] → a
foldl _ v []       = v
foldl f v (x : xs) = foldl f (f v x) xs
```

The *sum* function calculates the sum of a list of numbers. The *sum* function is defined in terms of the *foldl* function. The *sum* definition does not take any explicit arguments and will therefore immediately be rewritten into a *foldl*.da

In Haskell it is possible to call a function with fewer arguments than specified in the function definition. In this case a partially applied function, which is simply another function that takes as many arguments as left out, is returned [26]. For example, *foldl* (+) 0 returns a function that takes one argument (the list). This function will then be associated with the name *sum*.

$$\begin{aligned} &sum :: Num\, a \Rightarrow [a] \rightarrow a \\ &sum = foldl\ (+)\ 0 \end{aligned}$$

The (++) function appends two lists. The first pattern in the function definition specifies that if the first argument is the empty list ([]), the second argument is taken as the result. The second pattern in the function definition specifies that the first element of the list that is given as the first argument is used to construct a new list where the tail of that new list is the tail of the list that is given as first argument (*xs*) appended with the list that is given as second argument.

$$\begin{aligned} &(++) && :: [a] \rightarrow [a] \rightarrow [a] \\ &[] ++ ys && = ys \\ &(x : xs) ++ ys && = x : (xs ++ ys) \end{aligned}$$

In the derivation that is depicted at the left side of figure 2.3 the evaluation steps are given if the expression is evaluated according to the outermost (call-by-value) evaluation strategy. The outermost evaluation strategy will try to evaluate the outermost parts immediately and will delay evaluation of arguments if they are not needed for the final result. After rewriting *sum* into a *foldl* the evaluation steps of the ++ definition (which produces a list) and the *foldl* definition (which consumes a list) are interleaved. Notice that the additions are calculated at the very end.

In the derivation that is depicted at the right side of figure 2.3 the evaluation steps are given if the expression is evaluated according to the innermost (call-by-value) evaluation strategy. The innermost evaluation strategy will try to evaluate the innermost parts immediately, also if they are not needed for the final result. Observe that *sum* is immediately rewritten into a *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. Also notice that the sub-expression $[3,7]$++$[5]$ is fully evaluated before using the definition of *foldl*. Also observe that the additions are immediately calculated, this in contrast with the outermost evaluation strategy.

In the second example the expression $head(tail(take\ 3\ ([5,8]{+}{+}[80,40,120])))$ will be evaluated. The *head*, *tail* and *take* functions from the prelude will be introduced in this example [19].

```
    sum ([3,7] ++ [5])
=     { definition sum }
    foldl (+) 0 ([3,7] ++ [5])
=     { definition ++ }
    foldl (+) 0 (3 : ([7] ++ [5]))
=     { definition foldl }
    foldl (+) (0 + 3) ([7] ++ [5])
=     { definition ++ }
    foldl (+) (0 + 3) (7 : ([] ++ [5]))
=     { definition foldl }
    foldl (+) ((0 + 3) + 7) ([] ++ [5])
=     { definition ++ }
    foldl (+) ((0 + 3) + 7) [5]
=     { definition foldl }
    foldl (+) (((0 + 3) + 7) + 5) []
=     { definition foldl }
    (((0 + 3) + 7) + 5)
=     { applying + }
    ((3 + 7) + 5)
=     { applying + }
    (10 + 5)
=     { applying + }
    15
```

```
    sum ([3,7] ++ [5])
=     { definition sum }
    foldl (+) 0 ([3,7] ++ [5])
=     { definition ++ }
    foldl (+) 0 (3 : ([7] ++ [5]))
=     { definition ++ }
    foldl (+) 0 (3 : 7 : ([] ++ [5]))
=     { definition ++ }
    foldl (+) 0 [3,7,5]
=     { definition foldl }
    foldl (+) (0 + 3) [7,5]
=     { applying + }
    foldl (+) 3 [7,5]
=     { definition foldl }
    foldl (+) (3 + 7) [5]
=     { applying + }
    foldl (+) 10 [5]
=     { definition foldl }
    foldl (+) (10 + 5) []
=     { applying + }
    foldl (+) 15 []
=     { definition foldl }
    15
```

Figure 2.3: Evaluation steps of sum

$$
\begin{aligned}
& \mathit{head}\ (\mathit{tail}\ (\mathit{take}\ 3\ ([5,8] \mathbin{++} [80,40,120]))) \\
= \quad & \{\ \text{definition } ++\ \} \\
& \mathit{head}\ (\mathit{tail}\ (\mathit{take}\ 3\ (5 : ([8] \mathbin{++} [80,40,120])))) \\
= \quad & \{\ \text{definition } \mathit{take}\ \} \\
& \mathit{head}\ (\mathit{tail}\ (5 : (\mathit{take}\ 2\ ([8] \mathbin{++} [80,40,120])))) \\
= \quad & \{\ \text{definition } \mathit{tail}\ \} \\
& \mathit{head}\ (\mathit{take}\ 2\ ([8] \mathbin{++} [80,40,120])) \\
= \quad & \{\ \text{definition } ++\ \} \\
& \mathit{head}\ (\mathit{take}\ 2\ (8 : ([] \mathbin{++} [80,40,120]))) \\
= \quad & \{\ \text{definition } \mathit{take}\ \} \\
& \mathit{head}\ (8 : (\mathit{take}\ 1\ ([] \mathbin{++} [80,40,120]))) \\
= \quad & \{\ \text{definition } \mathit{head}\ \} \\
& 8
\end{aligned}
$$

Figure 2.4: Evaluation steps of head, tail and take according to the outermost evaluation strategy

The *head* function selects the first element of a non-empty list, the *tail* function removes the first element of a non-empty list and the *take* function selects the first n elements of a list.

$$
\begin{aligned}
& \mathit{head} && :: [a] \to a \\
& \mathit{head}\ (x : \_\ ) && = x \\
\\
& \mathit{tail} && :: [a] \to [a] \\
& \mathit{tail}\ (\_ : xs) && = xs \\
\\
& \mathit{take} && :: \mathit{Int} \to [a] \to [a] \\
& \mathit{take}\ 0\ \_ && = [] \\
& \mathit{take}\ (n+1)\ [] && = [] \\
& \mathit{take}\ (n+1)\ (x : xs) && = x : \mathit{take}\ n\ xs
\end{aligned}
$$

In the derivation that is depicted in figure 2.4 the evaluation steps are given for the expression when it is evaluated according to the outermost (call-by-name) evaluation strategy. Notice that as soon the first element of the list has been determined, the tail and head functions will be used.

In the derivation that is depicted in figure 2.5 the evaluation steps are given for the expression when it is evaluated according to the innermost (call-by-value) evaluation strategy. Notice that sub-expressions are fully evaluated before they are used.

$$
\begin{aligned}
& \mathit{head}\ (\mathit{tail}\ (\mathit{take}\ 3\ ([5,8] \mathbin{++} [80,40,120]))) \\
= \quad & \{\ \text{definition } ++\ \} \\
& \mathit{head}\ (\mathit{tail}\ (\mathit{take}\ 3\ (5 : ([8] \mathbin{++} [80,40,120])))) \\
= \quad & \{\ \text{definition } ++\ \} \\
& \mathit{head}\ (\mathit{tail}\ (\mathit{take}\ 3\ (5 : (8 : ([] \mathbin{++} [80,40,120]))))) \\
= \quad & \{\ \text{definition } ++\ \} \\
& \mathit{head}\ (\mathit{tail}\ (\mathit{take}\ 3\ [5,8,80,40,120])) \\
= \quad & \{\ \text{definition } \mathit{take}\ \} \\
& \mathit{head}\ (\mathit{tail}\ (5 : (\mathit{take}\ 2\ [8,80,40,120]))) \\
= \quad & \{\ \text{definition } \mathit{take}\ \} \\
& \mathit{head}\ (\mathit{tail}\ (5 : (8 : (\mathit{take}\ 1\ [80,40,120])))) \\
= \quad & \{\ \text{definition } \mathit{take}\ \} \\
& \mathit{head}\ (\mathit{tail}\ (5 : (8 : (80 : (\mathit{take}\ 0\ [40,120]))))) \\
= \quad & \{\ \text{definition } \mathit{take}\ \} \\
& \mathit{head}\ (\mathit{tail}\ [5,8,80]) \\
= \quad & \{\ \text{definition } \mathit{tail}\ \} \\
& \mathit{head}\ [8,80] \\
= \quad & \{\ \text{definition } \mathit{head}\ \} \\
& 8
\end{aligned}
$$

Figure 2.5: Evaluation steps of head, tail and take according to the innermost evaluation strategy

3 RELATED WORK

In this chapter the related work will be discussed. The first paragraph discusses related work concerning the inspection of Haskell expressions and the second paragraph discusses related work concerning intelligent tutoring systems.

3.1 INSPECTION OF EVALUATION STEPS

There are roughly three approaches to inspect the evaluation steps of a Haskell expression: trace generation, observing intermediate data structures, and using rewrite rules.

TRACE GENERATION A Haskell program is divided into two separate parts: the core part that is modelled as a set of pure functions and a part that can deal with user input and user output. Pure functions are functions that take all their input as explicit arguments, and produce all their output as explicit results [19]. A common way to debug / inspect programs written in an imperative language is to add some debug statements that print a message to the standard output or a log file to depict the flow of the program. In Haskell it is not possible to add any debug statements in a pure function because of the clear separation between pure functions and functions that deal with input and output. Trace generation is therefore used to debug Haskell programs. Trace generation is based on the idea to have a look at how the compiler reduces an expression.

GLASGOW HASKELL COMPILER The main compiler for Haskell code is the Glasgow Haskell Compiler (GHC). GHC converts Haskell source code into some core syntax code by means of parsing, renaming, type checking and desugaring [20]. Several core-language transformations can be applied to improve this code. In the next step this code will be transformed to the Shared Term Graph Language. Several transformations can also be applied on this code to improve it. Finally the code generation part will convert this language to the Abstract C data type that can be printed in C syntax. The printed C syntax is offered to a C compiler which will convert the C code to machine code. A main advantage of this approach is that a C compiler is almost always available for a certain hardware platform which improves portability. Another advantage is that GHC directly benefits from improvements made in the C

compiler. Because of this approach Haskell can be viewed as a high level language on top of C.

REDEX TRAILS It is possible to view a trace of a functional program in so called redex trail format [37]. Redex trails give a user insight into what redex will be selected by the compiler to reduce an expression. The solution to achieve this is to transform the original Haskell code into Haskell code with tracing facilities. Pure Haskell functions will be transformed to Haskell functions that store the evaluation order in a data type that can be printed to the user. The research on tracing functional programs has resulted in a complete debug library called Hat [10]. Hat can be used to understand how a program works and for locating errors in a program. The tracing consists of two phases: trace generation and trace viewing. The central idea for the trace generation 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 data type that can be viewed by the trace viewing component. The main advantage of Hat is that it is completely separated from the compiler so it does not matter which compiler is used. Hat provides also a graphical trace view of a Haskell expression.

A disadvantage of the approach described above is that the instrumentation of the original code can alter the execution of the program. The Shared Term Graph of the traced program can be different from the original program without the trace instrumentation code. Another approach, that is used by WinHIPE for example, is to use traces as pure observations of the program which results in an equally Shared Term Graph for both programs [33]. The approach is not to instrument the provided Haskell code but to instrument the interpreter. The advantage of this approach is that it allows to produce traces for any program that is executed without additional work for the programmer. Another advantage is the guarantee that the Shared Term Graph will not be altered so there is a clean separation between the trace and the program being observed. A disadvantage of this approach is that the interpreter (part of the compiler) needs to be adjusted. A main feature of WinHIPE is that it can depict the evaluation of Haskell expressions graphically in a call graph view.

OTHER APPROACHES The approach to observe intermediate data structures, that is also mainly used for debugging, is used in the Hood debugger [37]. With Hood it is possible to observe for example base types (such as Int and Bool) and finite and infinite structures (such as lists and trees). It is also possible to add observational capabilities for new defined types.

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 [30].

With these approaches it is possible to inspect the evaluation steps of a Haskell expression, but it is not possible to practice with these evaluation steps.

3.2 INTELLIGENT TUTORING SYSTEMS

The biggest problem for novice programmers is not to understand basic programming concepts but to learn how to apply these programming concepts in practice [23]. 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 [39]. Most students feel that they learn programming best by studying alone and while working alone on some programming assignment [23]. A main disadvantage of learning programming by doing and especially by doing it alone is that usually no direct feedback is available. Students can get frustrated and demotivated if they do not understand what is going wrong in their program.

An intelligent tutoring system has the purpose of overcoming this disadvantage. The main advantage of an intelligent tutoring system is that the student gets feedback at any desired 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 and the main responsibility for the inner loop is to give hints and feedback on student steps.

There are three important aspects for programming tutors [14]: development process, correctness and adaptivity. Development process is an indicator to which extent a tutor supports the incremental development of programs where students can obtain feedback or hints on incomplete programs and to which extent a student can follow his preferred way to solve a programming problem. Correctness is an indicator to which extent the tutor can guarantee that a student solution is correct, to which extent it can check if the student has followed good programming practice, if it can give counterexamples for incorrect programs and if it can detect at which point of a program a property is violated. Adaptability is an indicator to which extent a tutor provides functionality for a teacher to add his own exercises and to which extent particular solutions are enforced or disallowed. Several intelligent tutoring systems have been developed to learn Haskell. Two Haskell tutors will be described, one tutor is focused on the outer loop and one tutor is focused on the inner loop.

The Web-Based Haskell Adaptive Tutor focusses 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 [27]. A student can practice with three kinds of problems: evaluating expressions, typing functions and solving programming assignments. The difficulty of the exercises will be constrained by the level that the student has reached in previous ses-

sions and a student is not allowed to practice topics that have not yet been covered in the classroom. A disadvantage of this tutor is that it does not support the stepwise development of a program.

Ask-Elle is a Haskell tutor system that focusses more on the inner loop. It aims to teach students functional programming by developing their programs incrementally, receive feedback about whether or not they are on the right track, ask for a hint when they are stuck and see how a complete program is stepwise constructed [13]. Ask-Elle is developed by the Open Universiteit of the Netherlands and Utrecht University. It aims to support students that follow an introduction course in functional programming.

4

RESEARCH DESIGN

This chapter describes the research questions and research method in more detail. The components of the developed prototype are also described in more detail.

4.1 RESEARCH QUESTIONS

The main research question in this thesis is:

> **How can students in a tutoring environment be supported to understand evaluation strategies and programming concepts using the evaluation steps of Haskell expressions?**

The research question is divided into the following sub research questions:

1. Which programming concepts are suitable to illustrate with the stepwise evaluation of Haskell expressions?
2. How can the evaluation of Haskell expressions be inspected by systematically rewriting expressions?
3. How can feedback be generated when a student must provide the evaluation steps of a Haskell expression?
4. How can multiple evaluation strategies and user defined function definitions be supported?

4.2 RESEARCH METHOD

The motivation for this research is the observation that students following a first course in functional programming facing difficulties in understanding the evaluation steps of Haskell expressions. This research aims to support these students. Multiple research methods will be used: literature study, experiment and survey.

The literature study will focus on current possibilities in a tutor environment to support students in better understanding the evaluation steps of Haskell expressions. Another part of the literature study is focused on the understanding of technical concepts of how parts of the prototype can be realized.

The experiment consists of the development of a prototype which has the purpose of supporting students in understanding the evaluation of Haskell expressions and understanding evaluation strategies and programming concepts in general.

The survey will validate the conducted research. During the survey participants will be able to use the prototype to validate if the inspection and feedback are appropriate. The participants will also answer several questions about which programming concepts may be suitable to add to the prototype, which step size (granularity of the steps) is appropriate for them and does the prototype fulfil a particular need for them.

PROTOTYPE The prototype 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 the prototype will be integrated with a functional programming tutor such as Ask-Elle [13]. The component model of the prototype is shown in figure 4.1 and consists of a front-end, a back-end, and a strategy component. The back-end component uses the external components IDEAS and Helium. The research is focussed 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 [15] is used for parsing and pretty-printing expressions. Pretty printers are libraries that produce output that is suitable either for human consumption (for example debugging) or for machine processing [32]. 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. The main reason for this is that Ask-Elle and IDEAS are also written in Haskell, which makes integration easier. The back-end operates as a glue component that connects all other components. It receives an expression string from the front-end, uses the Helium compiler to parse the string and then converts the Helium output to the defined expression data type. The back-end receives expression results from IDEAS and will convert the defined expression data type back to the Helium data type and will then use the pretty-printer of the Helium compiler to convert this data type into a string. This string is presented to the user via the front-end.

IDEAS FRAMEWORK IDEAS stands for Interactive Domain-specific Exercise Assistants and is developed by the Open Universiteit of The Netherlands and Utrecht University. It is a framework for developing domain reasoners that give intelligent feedback [16]. 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 [13]. It is possible to provide such a strategy in an embedded domain specific language (EDSL) to IDEAS. IDEAS interprets the

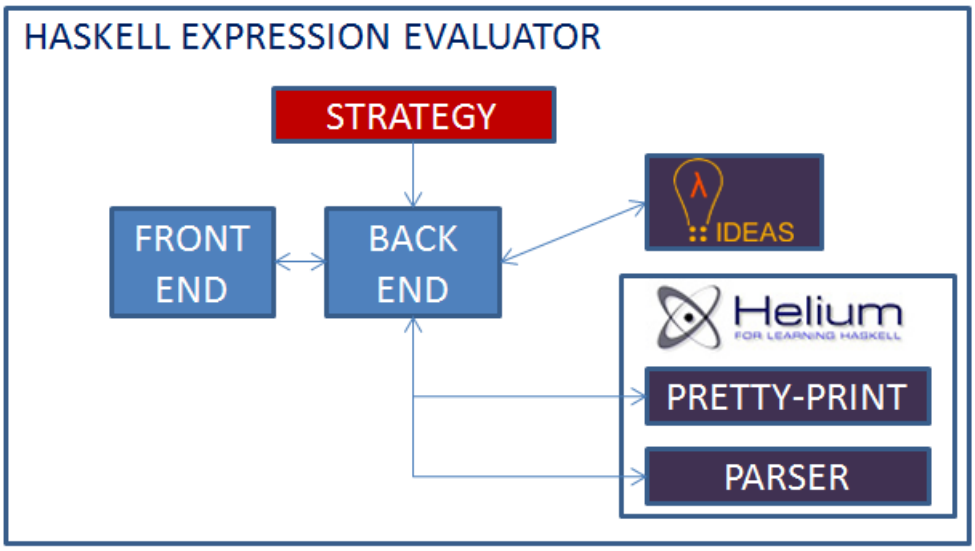


Figure 4.1: Component diagram of the prototype

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. The IDEAS framework is used for example in the functional programming tutor Ask-Elle and an intelligent tutor for rewriting logic expressions.

The back-end of the prototype can be seen as a domain reasoner for Haskell expressions. To develop a domain reasoner three parts needs to be defined: the domain itself (data type that represents Haskell expressions), rewrite rules for Haskell expressions (evaluation steps) and one or more strategies that combine rewrite rules to specify in which order sub-expressions are evaluated.

To give users feedback, an exercise and a domain reasoner need to be defined in the IDEAS framework. A domain reasoner specifies supported exercises, services and script files that are used to generate user-defined feedback. An exercise specifies the strategy, the parser, the pretty printer, a predicate to determine if an expression is fully evaluated (a literal or a list of literals), example expressions and two functions to determine if two expressions are semantically and syntactically equivalent. The prototype contains two exercises: one for the innermost evaluation strategy and one for the outermost evaluation strategy.

FRONT-END The front-end is web-based and uses the front-end framework Bootstrap. It is written in HTML and JavaScript and communicates through JSON to the back-end. The front-end provides an interface for users of the prototype 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 changed for another front-end and the purpose of this prototype front-end is to give insight into what kind of IDEAS services can be accessed.

A screenshot of the practice part of the front-end is shown in figure 4.2. A user can select an example Haskell expression by clicking on the 'Select' button or he 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 or the outermost evaluation strategy (which internally results in the behaviour of selecting a certain exercise). If the user clicks on the 'Evaluate' button the derivation of the expression is presented.

In the practice part of the prototype several standard IDEAS services are available for giving students hints about how a certain expression is evaluated. For instance, the prototype uses the services 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. The service 'do next step' makes it possible to step through the evaluation steps of a certain expression.

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'. This identifier is mapped to the string '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. Several IDEAS services are slightly modified to return this string representation instead of the rule identifier. For instance, the service that returns all applicable rules independent of the chosen strategy and the service to diagnose a student step.

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. Syn-

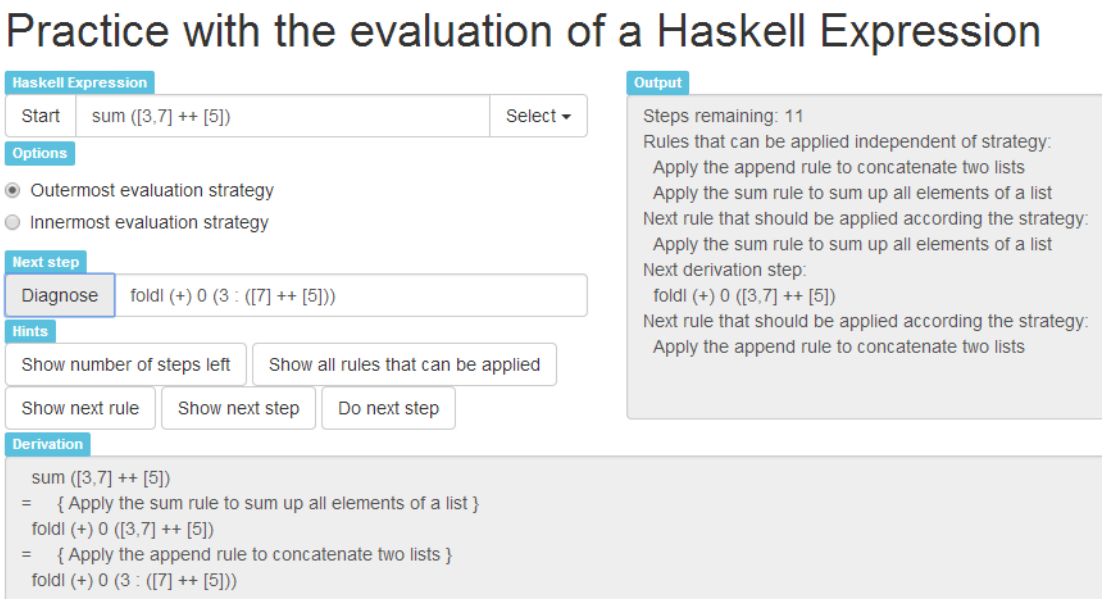


Figure 4.2: Practice with evaluation steps

tactic equivalence is obtained by deriving an instance of the Eq (equivalence) type class for the data type that represents Haskell expressions (Expr data type). 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

REWRITE RULES AND STRATEGIES

This chapter discusses the main parts to construct a domain reasoner. The first paragraph discusses the data type that represents Haskell expressions (the domain). The second paragraph discusses the rewrite rules (evaluation steps) and the third paragraph discusses the rewrite strategies that define the order in which to apply a certain evaluation step. The rewrite rules and strategies will be explained by using the example expression *sum* ([] ++ [5]). All rewrite rules for this example expression will be given in this chapter.

5.1 DEFINITION OF THE DOMAIN

An expression that a user enters in the front-end will be received by the back-end and parsed by the Helium compiler into an abstract syntax tree that is defined within Helium to represent an expression. The defined abstract syntax tree for an expression in Helium is complex and will therefore be simplified (converted) to the data type that is shown in figure 5.1. The reason for this simplification is to keep the rewrite rules and strategy readable. A disadvantage of this decision is that a translation is needed to convert the Helium expression data type to the expression data type and a translation is needed to convert the data types the other way around. However, readability is more important than not having this translation part.

The data type for an expression consists of the literal integer ('Con Int'), variables ('Var String') that are used to represent function names and to represent data type constructors, application ('App Expr Expr') and lambda abstractions ('Abs String Expr'). Lambda abstractions are anonymous functions that are typically used when a function is only needed once [26]. Some smart constructors (helper functions) are defined for constructing expressions. These smart constructors are given in figure 5.2. If a user enters an expression with the list notation (for example $[3,5]$), this list notation will be rewritten to the core syntax notation for lists ($3:5:[]$). The list notation in Haskell is actually syntactic sugar that provides an alternative way of writing code. Desugaring is the translation of syntactic sugar back to the core language [32]. The data type for expressions only supports the core syntax for lists.

Because the IDEAS framework is a generic framework it only operates on generic terms. Therefore, a function must be defined that translates the defined data type to the generic terms that are used by IDEAS. To accomplish the translation and the other way around, all constructors of the expression data type get a symbol name. IDEAS has already defined translation func-

```
data Expr = App Expr Expr     — Binary application
          | Abs String Expr   — Lambda abstraction
          | Var String        — Variable
          | Con Int           — Literal Integer
```

Figure 5.1: Data type for expressions

```
bin       :: String → Expr → Expr → Expr
bin s x y = App (App (Var s) x) y          — binary function

nil       :: Expr
nil       = Var "[]"                       — empty list

cons      :: Expr → Expr → Expr
cons      = bin ":"                        — list constructor

appN      :: Expr → [Expr] → Expr          — function with
appN      = foldl App                      — multiple parameters
```

Figure 5.2: Smart constructors for expression data type

tions for the primitive data types Int and String. For example an Int is encoded as a TNum value. To translate the data type to a generic term the function 'toTerm' is called. This function encodes the symbol name and will translate parameters recursively. The function 'fromTerm' is called to translate a generic term to the data type. In this function the original data type can easily be reconstructed based on the symbol names.

Conceptually, IDEAS contains a tree structure of an expression. To visit and modify a node in the tree data structure (tree traversal) the zipper technique [18] is used. The tree representation of the example expression *sum*([]++[5]) is depicted in figure 5.3. Notice that by calling the smart constructor *cons* with *cons* (*Con* 5) (*Var* "[]") the right part of the tree is constructed.

5.2 REWRITE RULES

IDEAS uses rewrite rules to rewrite a certain term into another term. After the expression has been entered by a user, successfully parsed, translated to the defined data type and finally translated to a generic term, IDEAS tries to apply a rewrite rule on this term. For every supported function (and operator) from the prelude a rewrite rule has been introduced.

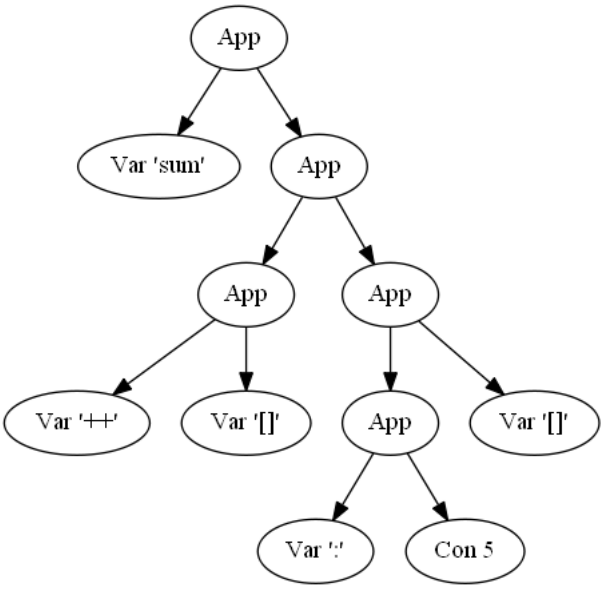


Figure 5.3: Internal structure of 'sum ([] ++ [5])'

There are two approaches to define rewrite rules: the extensional approach and the intensional approach [38]. In the extensional approach, rewrite rules are encoded as Haskell functions. Pattern matching is used to check if the argument matches the left-hand side of the rule. If the argument matches then the right-hand side of the rule is returned and if not the rule fails and nothing will be altered. An example of this approach is shown in figure 5.4 which contains the rule to add two numbers. Each rule has an identifier, and optionally also a description that is used for explaining the rewrite step. The descriptions of the prelude functions are taken from the appendix of Hutton's textbook [19].

Rules encoded as functions are straightforward but have some disadvantages [38]. The first disadvantage is that the rules cannot be encoded as one-line definitions because we have to provide a catch-all case for if the rule cannot be applied. This is needed to avoid run-time pattern matching failures. The second disadvantage is that pattern guards are needed if the left-hand side of the rewrite rule contains multiple occurrences of the same variable. The use of pattern guards makes the function less readable. The third disadvantage is that because Haskell lacks first-class pattern matching it is not easily possible to abstract common structures in the left-hand sides of the rewrite rules. Finally, rules can also not easily be analyzed since it is hard to inspect functions.

In the intensional approach, rewrite rules are encoded as values of a data type. Datatype-generic rewriting technology [38] is used to encode rules with the intensional approach. Rules are constructed with operator $\mapsto$, which

```
addRule :: Rule Expr
addRule = describe "Add two numbers" $ makeRule "add" f
  where
    f :: Expr → Maybe Expr
    f (App (App (Var "+")) (Con x)) (Con y)) =Just $ Con (x + y)
    f _ = Nothing
```

Figure 5.4: Extensional rewrite rule for (+)

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

Figure 5.5: Intensional rewrite rule for sum

takes an expression (a value of the Expr data type) at the left-hand side and the right-hand side. An example of this approach is given in figure 5.5 which contains the rewrite rule for the function definition *sum*. As soon as an expression is detected with the function name *sum* the expression can be rewritten to the equivalent *foldl* function. The datatype-generic rewriting technology contains generic rewrite functionality that attempts to apply rules to a certain expression type. If the rule cannot be applied the expression will not be altered. This intensional approach has some advantages. The first advantage is that rules can be encoded as one-line definitions. The second advantage is that it is possible to alter the matching algorithm for the rule's left-hand side to take associativity of certain operators into account. Other advantages are that it is possible to take the inverse of the rule and to generate documentation for the rule.

The rewrite rule for the definition of *foldl*, that is depicted in figure 5.6, is more involved since it uses pattern matching. The pattern variables in the definition of *foldl* are turned into meta-variables of the rewrite rule by introducing these variables in a lambda abstraction.

5.3 STRATEGIES

Rules are combined into a strategy to determine in which order certain rules must be applied. The embedded domain-specific language for specifying rewrite strategies in IDEAS defines several generic combinators to combine rewrite rules into a strategy [16]. 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. Labels can be added at any

```
foldlRule :: Rule Expr
foldlRule = describe "Process a list using an operator that associates to the left" $
    rewriteRules "foldl"
    [ \ op e x xs  →  appN (Var "foldl") [op, e, nil]  ↦  e
    , \ op e x xs  →  appN (Var "foldl") [op, e, cons x xs]
      ↦  appN (Var "foldl") [op, appN op [e, x], xs]
    ]
```

Figure 5.6: Intensional rewrite rule for foldl

```
appendRule :: Rule Expr
appendRule = describe "Append two lists" $ rewriteRules "append"
    [ \ _ _ ys  →  appN (Var "++") [nil, ys]  ↦  ys
    , \ x xs ys  →  AppN (Var "++") [cons x xs, ys]
      ↦  cons x (appN (Var "++") [xs, ys])
    ]
```

Figure 5.7: Intensional rewrite rule for (++)

position in the strategy to specialize the feedback that is generated. Table 5.1 summarizes the generic combinators that are used by the strategy.

Traversal strategies are recursive procedures that specify how and where a given transformation has to be applied [16]. There are multiple variations possible in these traversal strategies. For example, a full traversal (strategy is applied everywhere) versus a single location traversal, a top-down traversal versus a bottom-up traversal and a left-to-right traversal versus a right-to-left traversal. Navigation is needed to determine where a given transformation has to be applied. From a current location in the expression we can navigate for example to left, right, up or down. Several minor navigation rules have been defined to navigate through the data type structure. Minor rules are just ordinary rules but the difference with normal rules is that minor rules are not reported as an evaluation step. The navigation rules are used in 'apply' functions that takes a strategy and apply this strategy to for example the first child (argument) of the current node (function). These 'apply' functions are listed in figure 5.8.

An evaluation strategy defines in which order sub-expressions are reduced. Multiple evaluation strategies can be supported by encoding a different rewrite strategy per evaluation strategy. There is one rewrite strategy defined for the innermost (call-by-value) evaluation strategy and one rewrite strategy for the outermost (call-by-name) evaluation strategy.

Name	Notation	Description
Sequence	*s <*> q*	The sequence combinator specifies the sequential application of the strategies s and q.
Choice	*s <|> q*	The choice combinator defines that either the first strategy (s) or the second strategy (q) is applied.
Or else	*s |> q*	The or-else combinator defines that strategy s is applied or else strategy q is applied.
Alternatives	*alternatives*	The alternatives combinator generalizes the choice combinator to lists.
Conditional	*check p*	The check combinator takes a predicate p and only succeeds if the predicate holds.
Repetition	*repeatS s*	The repeatS combinator is used for repetition and will apply its argument strategy s as often as possible.
Recursion	*fix f*	The fixed point combinator is used to explicitly model recursion in the strategy. It takes as argument a function f that maps a strategy to a new strategy.
Succeed	*succeed*	The always succeeding strategy

Table 5.1: Generic strategy combinators

```
type Strat                =Strategy (Context Expr)

applyToFirstChild         :: IsStrategy f => f (Context Expr) → Strat
applyToFirstChild s       =goDown <*> s <*> goUp

applyToSecondChild        :: IsStrategy f => f (Context Expr) → Strat
applyToSecondChild s      =goDown <*> goRight <*> s <*> goUp
```

Figure 5.8: Navigation functions

```
innerMostStrategy :: LabeledStrategy (Context Expr)
innerMostStrategy = (repeatS . leftmostbu) ruleCombinationS
```

Figure 5.9: Innermost evaluation strategy

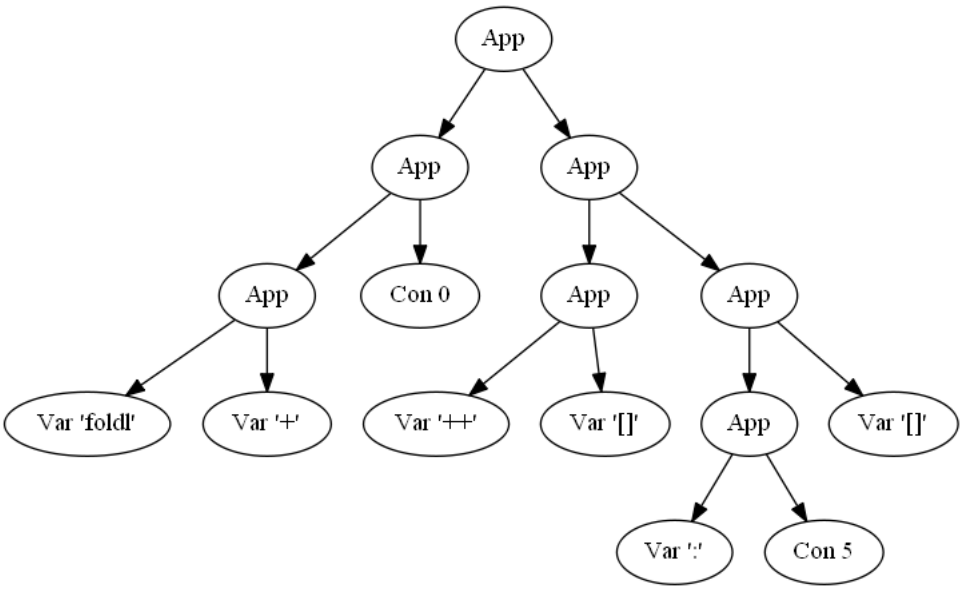


Figure 5.10: Internal structure of 'foldl (+) 0 ([] ++ [5])'

5.3.1 *Innermost strategy*

The innermost evaluation strategy is shown in figure 5.9. The innermost strategy uses *ruleCombinationS*, which is just a combination of all defined rewrite rules. It uses the traversal function *leftmostbu* that is defined within IDEAS and will try to apply this traversal function as often as possible. The *leftmostbu* traversal function traverses to the left most node position from bottom to top and tries to apply a defined rewrite rule on the term that is in focus. Note that the *Context* type is used as a zipper [18] data type for traversing expressions. Hence, we have to lift the rules to the *Context* type. From the tree representation of the expression *sum*([]++[5]) in figure 5.3 it is easy to observer that the left most node is 'Var sum'. The innermost strategy will therefore apply the sum rule immediately to this node. The result of this rewrite step can be observed in figure 5.10. The rewrite rule for the *sum* definition uses the smart constructor *appN* to construct *foldl* (+) 0.

```
outerMostStrategy :: LabeledStrategy (Context Expr)
outerMostStrategy = evalList $ fix $ \ whnf  →
    repeatS (spinebu (liftToContext betaReduction
    <|> alternatives (map ($ whnf) env)))
  where
    env =[foldlS, appendS, sumS, addS]
```

Figure 5.11: Outermost evaluation strategy

5.3.2 *Outermost strategy*

Expressions that are evaluated with the outermost evaluation strategy are not evaluated until they are required. Therefore the evaluator will try to use a rewrite rule on the outermost part of an expression. In the example expression of figure 5.10 the evaluator will try to apply a rewrite rule on the *foldl* function. From the function definition of *foldl* (see chapter 2) it can be observed that the third argument must be equal to the empty list or not the empty list before a decision can be made of which line in the function definition must be used. In other words the third argument must be evaluated to the outermost data constructor to determine which line in the function definition must be used. If the outermost part of an expression is a data constructor or a lambda abstraction it is called to be in weak head normal form [28]. Sub-expressions may or may not have been evaluated. In contrast, an expression is in normal form if it is fully evaluated and no sub-expressions could be evaluated further. Every expression that is in normal form is also in weak head normal form. For example the expression $3 : ([9,4]{+}{+}[10,20])$ is in weak head normal form because the outermost part is the data constructor :. And the expression 10+10 is not in weak head normal form because the outermost part is the function application (+). So to determine if a certain rewrite rule can be applied it is sometimes (depending on the function definition) needed to evaluate an argument to weak head normal form.

The outermost strategy that is shown in figure 5.11 can be divided into several parts: functions that inspect the value of a certain node, strategies that navigate through the tree structure, evaluation strategies for functions definitions and a strategy to evaluate data structures completely. Currently the list is the only supported data structure in the prototype. The list *env* in figure 5.11 contains all evaluation strategies for function definitions. For reasons of space only the evaluation strategies of *foldlS*, *appendS*, *sumS* and *addS* are shown. Notice that these functions get the lambda abstraction *whnf* as an argument (through the map function that applies a function to all elements of a list).

NODE INSPECTION In this section the helper functions that are responsible for inspecting a node value are explained. There are two helper functions for inspecting a node value: *isFun* and *isApp*. These helper functions are shown in figure 5.12.

The function *isFun* will verify if the current node in focus equals the function name that is given as the first argument and the number of parameters that is given as second argument. The function *currentInContext* is defined within the IDEAS framework and will return the current node in focus. It will return a *Maybe* type to indicate the possibility that there isn't any node in focus. The *Maybe* type is used in Haskell to represent the possibility of an error [32]. If the operation has failed the *Nothing* constructor is returned otherwise the expression (*Expr*), wrapped in the *Just* constructor is returned. The *maybe* function from the prelude takes a default value, a function and a *Maybe* type. If *currentInContext* returns a *Nothing* constructor the default value is returned (in this case the Boolean value False) otherwise the result of the function *rec* is returned. The *rec* function will return the Boolean value True if the *isFun* function is called with 0 as the number of arguments and the node in focus corresponds to a 'Var' node that has the same function name as the function name that was provided as the first argument. The other case that *rec* will return the Boolean value True is the case that the *isFun* function is called with x as the number of arguments, the node in focus corresponds to an 'App' node and unwrapping that node x times will result in the reach of a 'Var' node that has the same function name as the function name that was provided as the first argument. For example the expression *isFun* "*foldl*" 3 will return True in figure 5.10 if the node in focus is the root node.

The function *isApp* will return the Boolean value True if the node in focus corresponds with an 'App' node, otherwise the Boolean value False is returned.

NODE NAVIGATION STRATEGIES In this section sub-strategies are explained that are responsible for changing the focus in the tree. These sub-strategies uses the earlier mentioned navigation functions in figure 5.8. There are two node navigation strategies: *arg* and *spinebu*. These strategies are shown in figure 5.13.

The *arg* strategy is responsible for setting the correct focus for sub-expressions that must be brought into weak head normal form. This function is called with three arguments. The first argument specifies the argument number that must be brought into weak head normal form, the second argument specifies the total amount of arguments and the third argument contains the strategy that is responsible for actually bringing the sub-expression into weak head normal form. The *foldr* function from the prelude takes a function, a starting value and a list. The function is applied to all list arguments from right to left. The enumeration notation $[1 .. n\text{-}i]$ is used to construct a list [32]. The length

```
isFun          :: String → Int → Context Expr → Bool
isFun fn args = maybe False (rec args) . currentInContext
  where
    rec 0 (Var s)     = fn == s
    rec 0 _           = False
    rec n (App f _ )  = rec (n-1) f
    rec _ _           = False

isApp :: Context Expr → Bool
isApp = maybe False f . currentInContext
  where
    f (App _ _ ) = True
    f _          = False
```

Figure 5.12: Node inspection

```
arg       :: Int → Int → Strategy (Context Expr) → Strategy (Context Expr)
arg i n
    |i < n       =foldr (\ _ f → applyToFirstChild . f)
                    applyToSecondChild [1 .. n-i]
    |otherwise =error "invalid use of arg"

spinebu   :: Strategy (Context Expr) → Strategy (Context Expr)
spinebu s = fix $ \ x → (check isApp <*> applyToFirstChild x) |> s
```

Figure 5.13: Node navigation

of this list is the number of times that the focus must be moved to the first child node. After this focus change the strategy that was given as the third argument is applied to the second child of the current node in focus.

The *spinebu* strategy specifies a recursive strategy (fixpoint). If the node in focus (verified with the *isApp* function) is an 'App' node the focus is changed to the first child node. This process is a recursive process that will stop if the node in focus is not an 'App' node. On this node the strategy that was received as the first argument is applied.

EVALUATE DATA STRUCTURES The first step in the outermost evaluation strategy is the *evalList* strategy that is shown in figure 5.14. Notice that the argument of this function is the complete outermost evaluation strategy. The strategy will first execute the argument strategy and will then verify if the

```
evalList :: Strategy (Context Expr) → Strategy (Context Expr)
evalList s = fix $ \ x → s <*> try (check (isFun ":" 2)
    <*> arg 1 2 x <*> arg 2 2 x)
```

Figure 5.14: Evaluate data structures

```
sumS :: Strategy (Context Expr)
sumS _ = check (isFun "sum" 0)
    <*> liftToContext sumRule

appendS :: Strategy (Context Expr)
appendS whnf = check (isFun "++" 2)
    <*> arg 1 2 whnf
    <*> liftToContext appendRule

foldlS :: Strategy (Context Expr)
foldS whnf = check (isFun "foldl" 3)
    <*> arg 3 3 whnf
    <*> liftToContext foldlRule
```

Figure 5.15: Evaluation strategies for the definitions

node in focus is the data constructor : with two arguments (verified with the *isFun* function). If this is the case it will bring the first argument and then the second argument into weak head normal form (with the help of the *arg* function).

EVALUATION STRATEGIES For every supported function definition a rewrite rule and an evaluation strategy has been constructed. The evaluation strategies of *foldlS*, *appendS*, *sumS* and *addS* are shown in figure 5.15. The evaluation strategy functions will first check if the evaluation strategy must be applied on the node in focus (verified with the *isFun* function). If the evaluation strategy must be applied on the node in focus it will bring the arguments that need to be brought into weak head normal form into weak head normal form (with the help of the *arg* function). If all relevant arguments (depending on the function definition) are brought into weak head normal form the rewrite rule is applied on the node in focus. Notice that the strategy to bring an argument into weak head normal form is received by the *outerMostStrategy* function.

CONNECTING EVERYTHING TOGETHER The *outerMostStrategy* strategy that is shown in figure 5.11 connects all above functions together. The beta-reduction

rewrite rule will rewrite lambda abstractions 'App (Abs x e) a', where the variable x is substituted by a in an expression e.

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. These strategies are combined with the rewrite rule for beta-reduction. This strategy is applied to the left-spine of an application (in a bottom-up way) and will be repeated until the strategy can no longer be applied. To evaluate data structures completely (such as a list), the strategy must be repeated for the sub-parts of a constructor.

EXAMPLE The operation of the outermost evaluation strategy can be explained by inspecting the tree structure of the example expression *sum*([]++[5]). The tree structure of that expression is given in figure 5.3. The strategy will first use the *spinebu* function. This function navigates to the first child node using the function *applyToFirstChild* until the expression in focus does not contain a function application ('App'). If a node has been found that is not equal to an 'App' node, the *spinebu* function will apply strategy s, which is a fixed-point computation over the evaluation strategies. In the example, *applyToFirstChild* is only called once because only the top node contains a function application and the first child contains the 'Var sum' node. Now the evaluation strategies for the definitions are used. These strategies use the function *isFun* to verify the function name and to verify if it is used with the correct amount of arguments. The evaluation strategy for the *sum* definition is a match and therefore the sum definition will be used to rewrite the expression into a *foldl* on the first child node (seen from the root node) as depicted in 5.10.

Now the process is started over again, the strategy uses the *spinebu* strategy to navigate to the node 'Var foldl'. The evaluation strategies for the definitions will use the function *isFun* to determine if the expression in focus is a *foldl* function with exactly three arguments. This function will fail for all defined evaluation strategies for the definitions because there isn't a function called *foldl* with zero arguments. Because the strategy fails the focus will be changed (by *spinebu*) to the node directly above the node in focus. All evaluation strategies are tried again on that node but this will also fail. When the focus is at the root node there is a match for the *foldl* function with three arguments. The evaluation strategy for the definitions will now bring the third argument in weak head normal form by first changing the focus to the second child (third argument) and then calling the complete *whnf* strategy recursively. So on the second child (seen from the root node) the complete strategy is repeated again. It will spine to the node 'Var ++' and will apply the append rewrite rule on the second child node (seen from the root node).

6

USER DEFINED FUNCTION DEFINITIONS

From the conducted survey it became clear that there is a need to make the function definitions visible in the front-end. If the function definitions are visible, students can observe those definitions and can verify the evaluation steps. This can help students in the understanding of a derivation. The survey also showed that some teachers would like to add their own function definitions in order to incorporate the prototype into their course on functional programming. From an educational viewpoint it is interesting to experiment with different function definitions for the same function to observe the implications. For example, the *sum* function can be defined with a *foldl* function, *foldr* function or recursively.

It is possible to add multiple functions definitions by giving each definition an own name (for instance *sum* , sum' and sum'') and to manually add rewrite rules and evaluation strategies for the definitions. However, this approach has some drawbacks. To define rewrite rules and evaluation strategies the maintainer of the prototype (likely a teacher) needs to have knowledge of the defined data type for Haskell expressions and of the internal implementation to construct those rules and evaluation strategies. It is also necessary to recompile the prototype and therefore a maintainer would also need the complete build environment (e.g. Haskell compiler and used libraries). Another disadvantage is that the translation of function definitions to rewrite rules is a manual process where mistakes can be made.

When the strategies are observed closely, some common code can be noticed. Function definitions that have the same patterns have similar rewrite rules and evaluation strategies for the definitions. Another approach is therefore to generate these rewrite rules and evaluation strategies for the definitions from the function definitions. These function definitions can be defined in a Haskell source file (just a text file with valid Haskell syntax). With annotations [13] it is possible to add a description to every function definition. Annotations are multi-line comments with an identifier so the Haskell file remains a valid Haskell source file. This identifier, for instance DESC for description, can be used by the prototype to interpret the string after DESC as the rule description that is used in the front-end. The advantage of adding a description to a function definition is that the script file that contains a mapping of rule identifiers to rule descriptions as discussed in chapter 4 is not needed anymore. With the introduction of a function definition file rule identifiers are only used internally within the tool. With this approach there is one file, located on the web server, which can contain function definitions

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

Figure 6.1: Function definition file

that are supported by the prototype. An example of a function definition file is shown in figure 6.1. Notice that primitive functions (such as the operator +) cannot be defined in the function definition file. Primitive functions are functions that cannot be implemented directly in Haskell and are provided natively by the compiler.

There are various language constructs for defining functions in Haskell such as conditional expressions, guarded equations, pattern matching, lambda expressions and sections [19]. The function definitions that are depicted in figure 6.1 use pattern matching and this is currently the only supported way for defining functions. The left-hand side of the equal sign consists of a pattern and if that pattern matches with a particular expression the expression is rewritten to the expression at the right-hand side of the equal sign. The right-hand side of the function definition is an expression. The functionality for parsing an expression with the Helium compiler and converting this to the data type for Haskell expressions (see figure 5.1) can therefore be re-used. The pattern at the left-hand side of the function definition can also be parsed with the help of the Helium compiler. After the parsing process has been executed successfully a Helium data type for patterns is available. Because this data type is huge and complex it is simplified (in a similar way as the data type for expressions) to a data type for patterns that is shown in figure 6.2. In fact the complete function definition can be parsed by Helium. A *FDef* data type is introduced to represent a function definition that combines the pattern and expression parts. The *FDef* data type consists of two constructors: pattern binding (*PBinding*) and function binding (*FBinding*).

A pattern binding binds variables to values. Function definitions without any explicit parameters (such as *sum'*) are pattern bindings. If the function name of these functions are detected they can immediately be rewritten for the right-hand side of the definition. The *PBinding* constructor consists of a tuple of the pattern (left-hand side) and corresponding expression (right-hand side). For example the *sum'* definition is encoded as a *PBinding* with pattern variable PVar "*sum'*" as pattern.

```
data Pat = PCon Pat [Pat]    — Pattern constructor
         | PVar String       — Pattern variable
         | PLit Int          — Pattern literal

data FDef = PBinding (Pat, Expr)                  — Pattern Binding
          | FBinding (String, [([Pat], Expr)])    — Function Binding
```

Figure 6.2: Pattern data type

A function binding binds a variable to a function value. Function definitions with explicit parameters (such as *double*) are function bindings. The *FBinding* constructor consists of a tuple of the function name and a list of tuples (every line in the function definition corresponds to an element in this list). The tuple consists of a list of patterns (every parameter corresponds to an element in this list) and the corresponding expression. For example the *double* definition is encoded as a *FBinding* with pattern variable PVar "x" and function name *double*.

The *FDef* data type is combined with a String in a tuple. This string represents the rule identifier (and in future releases the rule description) that must be used. This description (DESC) can be used in the front-end to describe the generated rewrite rule and will be depicted if a user requests the next applicable rule.

GENERATE RULES AND STRATEGIES Take a look at the function definition of sum' in figure 6.1. The left-hand side of this function definition is just a simple pattern that consists of the variable name. All function definitions with only one variable name as pattern have the same rewrite rule structure. The function to generate rewrite rules for this kind of patterns is shown in figure 6.3. If there is a match with a pattern binding (*PBinding*) that contains one pattern variable a rewrite rule is constructed.

The generation of rewrite rules for function bindings with one pattern variable is also shown in figure 6.3. An example is the function definition $double\ x\ =\ x + x$. Remember from chapter 5 that pattern variables are turned into meta-variables of the rewrite rule by introducing these variables in a lambda abstraction. For every detected pattern variable (here x) a meta-variable must be introduced. This meta-variable must be substituted in the expression at the right-hand side of the function definition otherwise the variables are taken verbatim. This substitution is done with the function *subst* that substitutes variable x by expression a in expression e.

```
type MRule = Maybe (Rule Expr)

genRule                                        :: (String, FDef) → MRule
genRule (rId, PBinding (PVar f, e))            =
    Just $ rewriteRule rId $ Var f ↦ e
genRule (rId, FBinding (f, [([PVar x], e)])) =
    Just $ rewriteRule rId $ \ a → App (Var f) a ↦ subst x a e
genRule _                                      = Nothing
```

Figure 6.3: Generate rewrite rules

```
type Strat       = Strategy (Context Expr)

evalStrat        :: String → Int → Rule Expr → [Strat] → Strat
evalStrat f n r s = check (isFun f n)
    <*> foldl (<*>) succeed s
    <*> liftToContext r
```

Figure 6.4: Abstraction of evaluation strategies for the definitions

The evaluation strategies for the function definitions that are used in the outermost evaluation strategy can be generalized. The function *evalStrat* that is shown in figure 6.4 is an abstraction of these strategies. This function takes the function name, number of arguments, the rewrite rule and a list of strategies that contain optional steps to bring certain arguments into weak head normal form. The *evalStrat* function will first check with the *isFun* function if the function name and the number of arguments correspond with the current node in focus. If there is a match all strategy steps to bring arguments into weak head normal form will be applied. As a final step the rewrite rule will be applied.

The functions to generate evaluation strategies are shown in figure 6.5. The function *genEvalStrat* generates evaluation strategies. This function calls the function *genRule* to generate a rewrite rule and then calls the function *conEvalStrat* to construct an evaluation strategy. The function *conEvalStrat* uses the function *evalStrat* to construct the strategy. The empty list is given as the third argument because for these patterns there are no arguments that must be brought into weak head normal form. Notice that these functions currently only support pattern bindings and simple function bindings with one pattern variable. Another limitation is that function definitions that have multiple patterns are not supported.

```
type MStrat              = Maybe (Strategy (Context Expr))

genEvalStrat             :: (String, FDef) → Strat → Strat
genEvalStrat fdef whnf = fromJust $
    conEvalStrat fdef (fromJust $ genRule fdef) whnf


conEvalStrat :: (String, FDef) → Rule Expr → Strat → MStrat
conEvalStrat ( _ , PBinding (PVar f, e)) r whnf =
    Just $ evalStrat f 0 r []
conEvalStrat ( _ , FBinding (f, [([PVar x], e)])) r whnf =
    Just $ evalStrat f 1 r []
conEvalStrat _ _ _ = Nothing
```

Figure 6.5: Generate evaluation strategies

MORE COMPLEX PATTERNS The defined generate functions can be extended in the future to support more complex patterns. An extension is to determine the number of arguments from a function definition. The number of arguments is used in the function *isFun* to determine if the evaluation strategy can be applied. This can be accomplished by counting the number of elements in the pattern list.

It is also possible to determine which argument(s) must be brought into weak head normal form by taking multiple patterns into account. For example, from the recursive definition of sum'' it can be determined that the first argument (the list) must be brought into weak head normal form. The list must be in weak head normal form, otherwise it is not known which line in the function definition must be used.

Notice that the complete support of user defined functions is future work and therefore the presented data type and generate functions could be changed due to new insights.

7

VALIDATION

To validate the research a small qualitative research study has been executed. With this qualitative research approach it is possible to get a better understanding if a tool such as the developed prototype can support students in the understanding of programming concepts and evaluation strategies. Qualitative research is mainly exploratory and preliminary in nature and its results are only an indication of a possible generic conclusion [22]. In qualitative research it is important to precisely depict all answers on questions. Usual qualitative research methods are the group discussion and the open interview.

DESIGN The decision was made to hold a survey with closed and open questions among teachers and students that participate (or participated) in a functional programming course. The purpose of the open questions is to transform the survey more to an open interview than an ordinary survey. A survey is actually a quantitative method because of the fixed question layout and the inflexibility to ask and respond to questions. The reason to choose for a survey and not for multiple personal open interviews is to keep the participation accessible and to reduce the time to carry out the research. Also notice that the approached teachers work on different geographic locations and the approached students study at the Open Universiteit of the Netherlands (which is a distance learning university) which means that students can be located anywhere in the Netherlands and Belgium. The disadvantage of using a survey instead of an open interview is that the question layout is fixed and it is not possible to anticipate on the response of a question. It is also not possible to observe the non-verbal communication of a participant.

The complete survey, that can be found in appendix A, and the prototype was made available online. Participants could follow the survey steps, explore the prototype and enter their answers online. Eight teachers and twenty nine students have been approached via e-mail to ask if they would join the survey. The teachers are involved in functional programming at the Open Universiteit of the Netherlands, Utrecht University, Radboud University Nijmegen or Chalmers University of Technology. The students are currently enrolled in, or have just completed an introduction course in functional programming at the Open Universiteit of the Netherlands. All participants were approached on April 30, 2014 and all surveys that were filled in by May 12, 2014 have been taken into account. An overview of all the answers on the survey can be found in appendix B.

RESULTS Seven teachers in functional programming and nine students (total of sixteen people) have participated in the survey. The participation rate under teachers is 88% and the participation rate under students is 31%. Three student participants have not filled in the open questions so the effective participation rate under students is 21%. Almost every participant (fifteen of the sixteen participants) has practical experience with an imperative programming language and even eight participants are experts in an imperative programming language. All teacher participants are experts in some functional language. Two student participants have practical experience in a functional language, five student participants have completed an introduction course in functional programming and two student participants are currently enrolled in an introduction course in functional programming. The three student participants that have not filled in the open questions have various experience in functional programming: one has practical experience, one has completed an introduction course and one is currently enrolled in an introduction course. These three student participants are further not taken into account.

All participants agree that it is useful for students following an introduction course in functional programming to inspect how an expression is evaluated exactly. However, one student participant argued that the examples in Hutton's textbook [19] are sufficient to get a good understanding of the evaluation steps and another student participant argued that inspecting evaluation steps can also be confusing for those who just start programming for the first time. Eleven participants agree that it is also useful to inspect multiple evaluation strategies, but two participants think it might be confusing for absolute beginners. Also one participant would like to see the multiple evaluation strategies side-by-side so the differences can easily be inspected. To support this functionality, it must be possible to detect if an expression can be evaluated according to a certain strategy so that the user can be notified accordingly.

There is less consensus about the question if more evaluation strategies should be supported. Four teacher participants convincingly argued that lazy evaluation must be supported because the evaluation is very subtle, but two other teacher participants argued that this is too much for the basic understanding of programming concepts. There is also a teacher participant who says it would be nice to add lazy evaluation but is not sure if this is worth the effort. One student participant does not know enough of the subject to answer this question properly and all other student participants find the addition of lazy evaluation useful.

Practicing with the evaluation steps was received well by all participants. However, three participants argued that it is only useful in the very beginning and students can quickly become boring by entering all evaluation steps. Another remark is made by a teacher saying that the studying of evaluation

steps is not the only way to reason about a program. A tool similar to the prototype must therefore be seen as an additional tool for students.

There was division among participants whether or not it is useful to skip certain evaluation steps. Five teacher participants and one student participant argued that it should not be possible to skip steps to keep students aware that each step is necessary. But the other participants would like to see this feature into the prototype because entering in all steps can be boring especially with larger exercises. This problem can be solved by using a student model in the tool as has been done in the Web-Based Haskell Adaptive Tutor [27]. With this feature students must log into the tool to get a proper exercise based on their own progress. For example, in the very beginning students must fill in all evaluation steps and later on several evaluation steps may be skipped. Another advantage of having a student model is that students facing some difficulties with a certain concept (for example higher-order functions) can get more exercises based on that topic.

All participants agree that a tool similar to the prototype is useful for the understanding of programming concepts that are important in functional languages. One participant remarked that the concepts of guards and conditional expressions can also be explained by using a tool similar to the prototype. Another remark that is made is that a tool similar to the prototype is only suitable for beginners.

Twelve participants would use or consider to use the tool if they were learning a functional language for the first time. One participant remarked that he had never had problems with understanding the concepts, but other participants would use the tool or would advise the tool to their students.

Four participants find it useful to add their own function definitions and to adjust function definitions to inspect the behavior of their change on the evaluation of the expression. Five participants find this functionality possibly useful. Three participant find this functionality not useful and find the support of prelude functions sufficient.

Some participants suggested that it would be a nice improvement to see the actual function definitions. This could help students in understanding the evaluation steps.

CONCLUSIONS The conclusion of the survey is that most participants appreciate the prototype. They find it useful for students participating in an introduction course on functional programming and think it can help those students with understanding some programming concepts. However, some students face more difficulties with the understanding of those concepts than others and therefore some students will benefit more from a tool similar the prototype than others.

The participants also suggested some advisable improvements. Examples of these improvements are to detect if an expression cannot be evaluated

according to the innermost strategy and notify the user accordingly, add support for the lazy evaluation strategy, make supported function definitions visible and add user-defined function definitions.

Another conclusion is that there is not always consensus among teachers about features. One such example is the feature for skipping certain evaluation steps. Therefore it is important that those features are configurable for the teacher so the teacher can adapt the tool to its course.

8

CONCLUSIONS AND FUTURE WORK

In this chapter an answer to the research questions is given in the 'conclusion' section. The section 'maturity' describes which Haskell language features are currently supported and which language features are useful to add in the future to the prototype. The section 'lazy evaluation' describes several approaches for expressing sharing. Suggested future work is discussed in the 'future work' section.

CONCLUSIONS The inspection of the stepwise evaluation steps of Haskell expressions can be used by students to get a better understanding of important programming concepts such as recursion, higher-order functions, pattern matching and lazy evaluation. Chapter 2 describes how the stepwise evaluation of expressions can be used to explain those concepts. Practicing with these evaluation steps can motivate students to think thoroughly about how these concepts work exactly. Practicing and inspecting evaluation steps is especially useful for students that follow an introduction course in functional programming without any prior experience in functional programming.

The presented prototype uses rewrite rules and rewrite strategies to show the evaluations steps of a Haskell expression according to a certain evaluation strategy. A domain reasoner in the IDEAS framework has been developed for rewriting expressions. The evaluation process does not only depend on the evaluation strategy that is used, but also on the function definitions. For the outermost evaluation strategy different arguments (depending on the function definition) must be brought into weak head normal form before a certain rewrite rule can be applied.

The IDEAS framework contains standard services that can be used to diagnose an evaluation step and to give several hints about the next evaluation step. For instance, one service returns all rules that can be applied without considering the strategy and another service gives the number of steps left in the derivation. A script file, that contains a mapping of rewrite rule identifiers to rewrite rule strings, can be used to modify the description of rewrite rules and therefore the specific feedback messages for a student. A web front-end has been developed to depict the derivation of a certain expression. The front-end calls feedback services and made it possible to conduct a small qualitative research among teachers and students involved in an introduction course in functional programming. The qualitative research (survey) supports the hypothesis that the prototype can help students in a better understanding

of programming concepts such as recursion, higher-order functions, pattern matching and lazy evaluation.

Multiple evaluation strategies can be supported to define a rewrite strategy for every evaluation strategy. Currently, the innermost and outermost evaluation strategies are supported by the prototype. User defined function definitions can be defined by introducing a file with function definitions that the prototype can parse. Rewrite rules and evaluation strategies can then be generated from those function definitions.

The overall conclusion is that the prototype can support students to understand evaluation strategies and programming concepts. However, it is important to make the tool configurable for teachers so they can adapt the tool to their courses. Also further experimental research is necessary to investigate the extent to which the prototype will support students in the understanding of these concepts.

MATURITY The prototype can be used to support an introduction course in functional programming. Several introductory concepts from for example Hutton's textbook [19] are currently supported:

- Pattern matching that is described in chapter 4
- Recursive functions that is described in chapter 6
- Higher-order functions that is described in chapter 7
- Evaluation strategies that are described in chapter 12

To make the prototype more mature it is useful to also support lazy evaluation (chapter 12), conditional expressions, guarded equations (chapter 4) and list comprehensions (chapter 5).

LAZY EVALUATION The lazy evaluation strategy that is used by Haskell combines the outermost evaluation strategy with sharing. If an expression is evaluated according to the lazy evaluation strategy an argument is only reduced if the value of the corresponding formal parameter is needed for the evaluation of the function body. After reducing the argument, the resulting value will be saved for future references to that formal parameter [2]. Currently, sharing is not supported in the prototype and therefore does not give the student any insight in which terms are used for sharing. There are different approaches for expressing sharing [4].

One approach is to use graphs to make sharing visible by drawing one physical variable representation with pointers to the locations of where physical parameters are accessed. To automatically construct such graphs, a structure to represent binding or variable renaming must be defined.

Another approach is called extraction [4]. This approach uses a transformation that extracts all free variables from a function and everything that remains of the function is replaced by a symbol called the supercombinator. To handle the new symbols, new reduction rules are added.

It is also possible to formalize the operational semantics. There are two styles: by using natural semantics (also called big-step semantics) or by using reduction semantics (also called small-step semantics) [31]. In natural semantics a term is directly related to its result and the behavior of a term is expressed in terms of the behavior of its sub terms [9]. In reduction semantics a subterm is reduced step by step and these transitions are reflected at the top level. Natural semantics is used by Launchbury where let expressions are introduced to name the arguments and then put these arguments in a heap [25]. If the content of a variable is needed, the corresponding expression in the heap is evaluated, and the heap is updated with the obtained result. The heap can be seen as an unordered set of pairs of variables and expressions that binds variable names to expressions. Reduction semantics is used by Ariola e.a. where let expressions are introduced as closures [2]. A closure is a function or reference to a function together with a referencing environment. This referencing environment contains a table that stores a reference to each of the non-local (free) variables of that function. Another example of using reduction semantics is to express sharing by using explicit substitutions [6].

Another principle to express sharing is to use sharing via labelling [4]. In this approach every symbol in a term gets a label. Two sub terms with the same label are supposed to be equal.

At first sight extending the prototype along the lines of Launchbury's natural semantics for lazy evaluation, and by making the heap explicit seems the best option. This corresponds with the explanation of lazy evaluation in Hutton's textbook [19]. To indicate sharing, arguments can get a name that refers to a formal parameter. A separate column in the front-end can then depict the heap with the current values of that formal parameter.

FUTURE WORK The purpose of the prototype is to investigate if a tool such as the prototype can support students in the understanding of programming concepts. Several improvements are suggested to make the tool more mature. An important improvement is to make all function definitions that can be used in the tool visible for the user. This can be done by parsing a function definition file (which contains all supported function definitions) and present the function definitions in the front-end. For example, the front-end can have a pop-up window so that the user can place the function definitions next to a expression derivation.

Another improvement is to fully support the possibility to use user-defined function definitions. This is especially useful if all kind of example derivations must be supported. For example, Hutton [19] redefines the prelude functions

product and length on page 50 to explain recursive functions. To fully support this functionality, the prototype needs to be extended to generate rules and generate evaluation strategies for all kind of function definitions. Currently, the prototype only supports single line function definitions and function definitions may contain only one pattern variable. A first step could be to support all patterns that are used in the function definitions that are currently supported. All supported functions, except primitive functions (such as the operator +), can then be defined in the function definition file. This file can be used for the visualization of these function definitions in the front-end.

Because entering every evaluation step multiple times can be boring after some time, it is also desirable to configure the granularity (step size) of a certain step. Currently the prototype only supports a step size of one. Step size is the number of steps that the evaluator will use to rewrite a certain expression. For example, the expression $3 + (4 + 7)$ will be evaluated in two steps, although most students will typically combine these steps. The step size of the evaluation steps can be altered by extending certain rewrite rules to accept different step sizes. The description of the function definitions and the step size of a function definition can be configured, for example with annotations in the function definition file. More research must be carried out to automatically derive or configure a certain step size that suits most students.

Another improvement is to add more Haskell language constructs such as conditional expressions and guarded equations and to add support for the lazy evaluation strategy. Lazy evaluation can be supported by introducing let expressions to label the arguments, to put these arguments in a heap and to make the heap explicit in the front-end.

The long-term goal is to integrate the functionality of the prototype in the Ask-Elle programming tutor, which then results in a complete tutor platform to help students learn programming. This tutor platform also supports distance-learning. Another long-term goal is to combine the prototype with QuickCheck properties [11]: when QuickCheck finds a minimal counter-example that falsifies a function definition (for example for a simple programming exercise), then the prototype can be used to explain more precisely why the result was not as expected.

A

SURVEY

1. Are you a teacher or student?
   a) Student
   b) Teacher
2. Do you have any experience with an imperative programming language? (For example C, C++ or Java)
   a) No experience
   b) Participating in an introduction course
   c) Completed an introduction course
   d) Practical experience
   e) Expert
3. Do you have any experience with an functional programming language? (For example Haskell, Clean or ML)
   a) No experience
   b) Participating in an introduction course
   c) Completed an introduction course
   d) Practical experience
   e) Expert
4. At the time that you were learning a programming language; did you ever face difficulties with understanding how an expression was evaluated?
   a) Never
   b) Only with my first programming language
   c) Only with a language in a new paradigm (imperative/functional/-logic)
   d) With every new language
   e) Even with languages I already know
5. Do you know that various programming languages use a different strategy to calculate expressions like the innermost (call-by-value) evaluation strategy and the outermost (call-by-name) evaluation strategy?

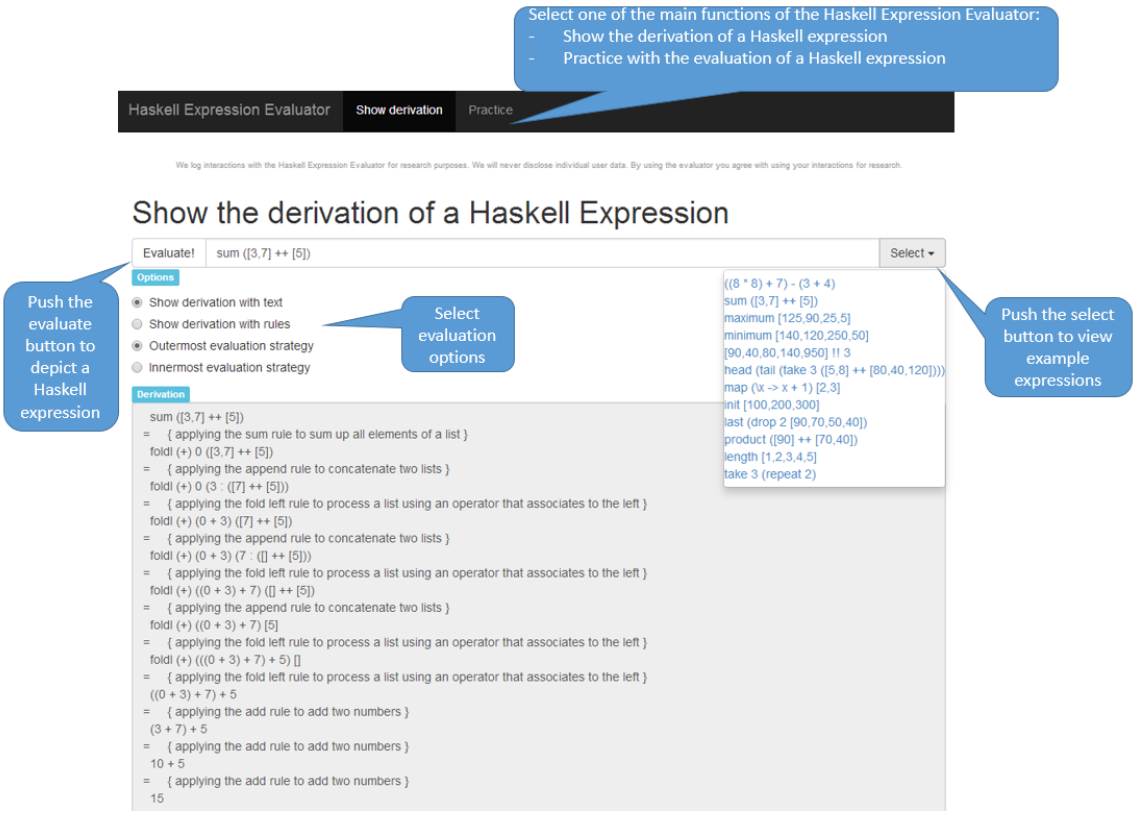
a) Yes

b) Yes, but I do not know the evaluation strategies mentioned

c) No

A prototype has been developed to depict the evaluation steps of a Haskell expression. The purpose is to give students insight in how expressions are evaluated exactly.

A screenshot that explains the various buttons is depicted below. Notice that the prototype currently only supports a limited set of Haskell expressions and evaluation rules. Currently two evaluation strategies are supported: the outermost (call-by-name) evaluation strategy and the innermost (call-by-value) evaluation strategy.



Follow the following steps to evaluate a Haskell expression:

- Open the prototype: http://ideas.cs.uu.nl/HEE/
- Push the 'Select' button to view some example Haskell expressions that are supported by the evaluator.
- Select the expression $sum([3,7]{+}{+}[5])$
- Click on the 'Evaluate' button to view the evaluation steps.

- Modify the evaluation strategy by clicking on 'Innermost evaluation strategy' (call-by-value).
- Click on the evaluate button to view the evaluation steps of the same Haskell expression according to a different strategy. Notice the differences between the outermost (call-by name) evaluation strategy where the steps for ++ and foldl are interleaved and the additions are calculated at the end and the innermost (call-by-value) evaluation strategy which fully evaluates the sub-expression [3,7]++[5] before using foldl's definition.

6. Do you think it is useful for students following an introduction course in functional programming to view how the expression was evaluated exactly?
   Why do you think this is useful or not useful?

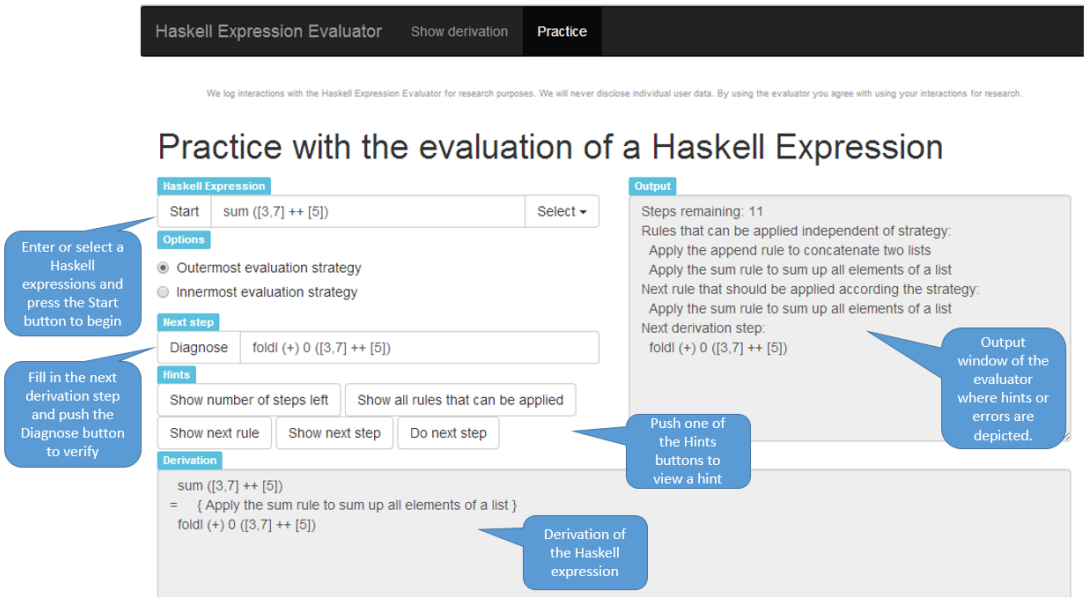
Another nice example where the differences between the different evaluation strategies is made clear is the Haskell expression 'head (tail (take 3 ([5,8] ++ [80,40,120])))' (available via the 'Select' button).

7. Do you think it is useful for students following an introduction course in functional programming to view how expressions are evaluated according to different evaluation strategies?
   Why do you think this is useful or not useful?

8. Do you find it useful to add more evaluation strategies? For example combining the outermost evaluation strategy with sharing (lazy evaluation strategy)?
   Are there any other evaluation strategies that you find useful to add to this prototype?

It is also possible to practice with the evaluation steps of a Haskell expression. Below a screenshot is depicted of the practice part of the evaluator. Follow the following steps to practice with a Haskell expression:

- Open the prototype: http://ideas.cs.uu.nl/HEE/
- On the navigation bar (top, black bar) click on 'Practice'.
- Click on the 'Select' button to select the expression 'sum([3,7]++[5])'
- Click on the 'Start' button
- You can fill in the next evaluation step under section 'Next step' and push the 'Diagnose' button to verify the input step.
- If you like to view a hint you can click on a button under the section 'Hints'. You can view for example how many steps there are left, the next rule that should be applied and the next derivation step.



9. Do you think it is useful, for students following an introduction course in functional programming, to practice with the evaluation steps of a Haskell expression?
   Why do you think this is useful or not useful?

10. Currently it is not possible to add evaluation rules to the prototype by yourself. Do you find it useful to add your own evaluation rules?
    a) No, just support the prelude
    b) Maybe

c) Useful

d) Very useful

11. Currently the prototype depicts every evaluation step and in the practice part every evaluation step must be provided by the student. For example the expression '(3+4)+7' is evaluated to '7+7'. Most students might write the final answer (14) immediately.

    Do you think it is useful to skip certain evaluation steps? And to configure this for example for some evaluation rules?

12. A tool like the prototype can also help with the understanding of certain programming concepts. Like recursion (for example ++), higher-order functions (for example foldl) and pattern matching.

    Do you think it is useful, for students following an introduction course in functional programming, to practice with the evaluation steps of an expression and to view the derivation of an expression to understand these concepts?
    Do you know other programming concepts where a tool like this could help students in understanding this concept?

13. Do you think that if you had a tool like this available when you learned a functional language that it helped you with the understanding of how expressions are calculated?
    And would you use a tool like this?

14. Remarks

B

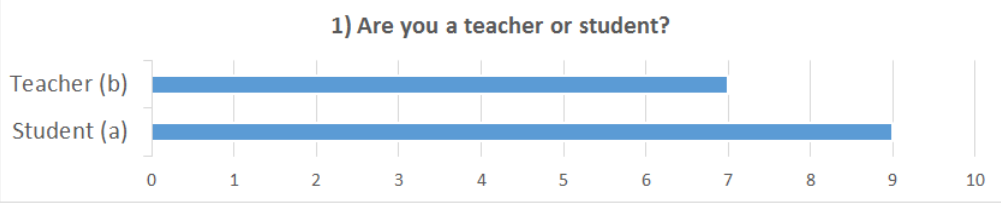
SURVEY RESULTS

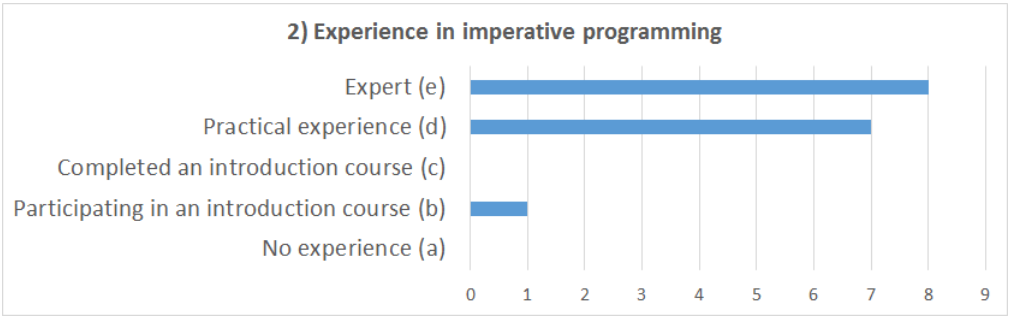
MATRIX OF CLOSED QUESTIONS

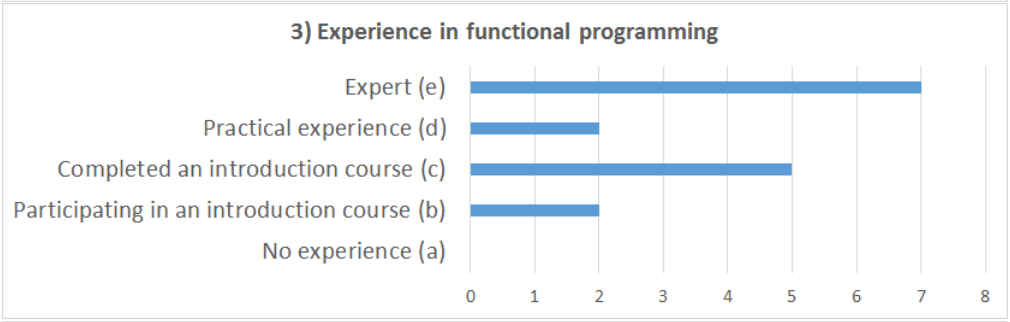
Sixteen people have filled in the survey. The questions with a fixed answer are encoded with numbers (see appendix A) and presented in a matrix that is depicted below.

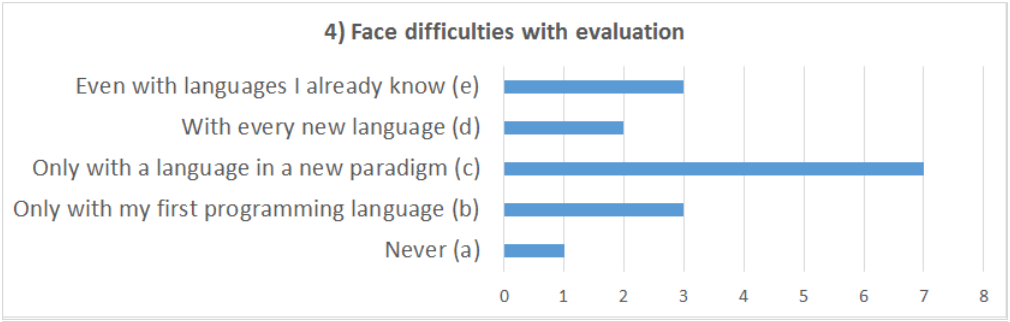
Question	1	2	3	4	5	10
Respondent 1	b	e	e	e	a	d
Respondent 2	a	d	d	e	a	-
Respondent 3	a	e	c	a	a	a
Respondent 4	a	e	b	c	a	-
Respondent 5	a	d	c	b	a	-
Respondent 6	a	e	c	c	a	b
Respondent 7	a	d	d	d	b	b
Respondent 8	b	e	e	c	a	b
Respondent 9	a	d	c	b	a	c
Respondent 10	b	e	e	d	a	c
Respondent 11	b	d	e	e	a	b
Respondent 12	b	e	e	c	a	a
Respondent 13	a	e	c	c	a	a
Respondent 14	b	d	e	c	a	-
Respondent 15	b	d	e	c	a	b
Respondent 16	a	b	b	b	a	c

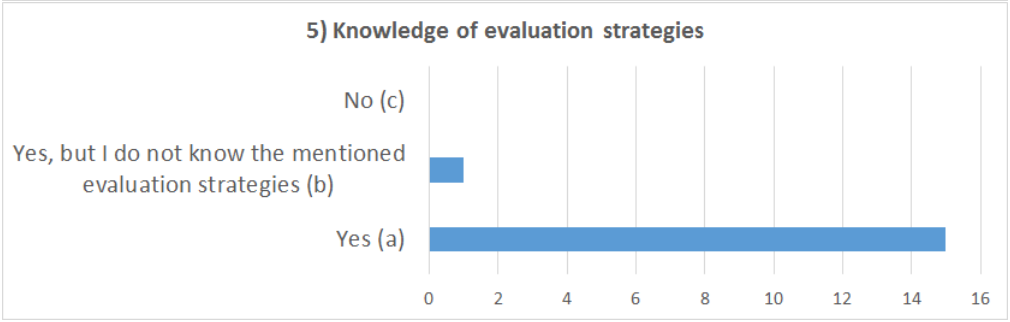
GRAPHICAL VIEW OF CLOSED QUESTIONS

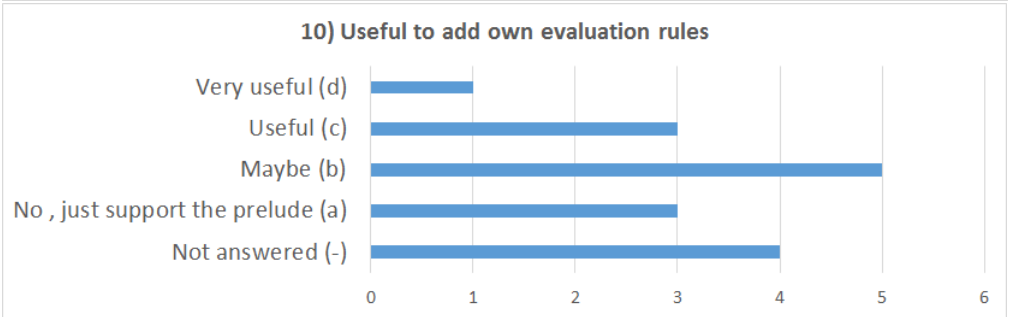



2) Experience in imperative programming
Expert (e)
Practical experience (d)
Completed an introduction course (c)
Participating in an introduction course (b)
No experience (a)
0 1 2 3 4 5 6 7 8 9


3) Experience in functional programming
Expert (e)
Practical experience (d)
Completed an introduction course (c)
Participating in an introduction course (b)
No experience (a)
0 1 2 3 4 5 6 7 8


4) Face difficulties with evaluation
Even with languages I already know (e)
With every new language (d)
Only with a language in a new paradigm (c)
Only with my first programming language (b)
Never (a)
0 1 2 3 4 5 6 7 8


5) Knowledge of evaluation strategies
No (c)
Yes, but I do not know the mentioned evaluation strategies (b)
Yes (a)
0 2 4 6 8 10 12 14 16


10) Useful to add own evaluation rules
Very useful (d)
Useful (c)
Maybe (b)
No , just support the prelude (a)
Not answered (-)
0 1 2 3 4 5 6

OPEN QUESTIONS

In this section all answers from the open questions are depicted. Notice that not all respondents have answered all open questions.

- **Question 6:** Do you think it is useful for students following an introduction course in functional programming to view how the expression was evaluated exactly?

Respondent	Answer
1	This is certainly useful. It can help to understand evaluation mechanisms. However, using this tool may give the student the idea that studying evaluation orders is the *only* way to reason about a program. Applying induction and substitution are also very good ways to understand a program.
3	Seeing a step by step evaluation of the expression makes it easier to understand
6	It is useful, in the sense that more is better. Hutton and the workbook also contain lots (in my opinion enough) of samples. I don't think that it would have made a difference in my understanding of Haskell. It might have saved me some time though, because I could have skipped some of the tedious work of writing down the evaluations of the Haskell expressions.
7	Yes, it provides more insight.
8	Yes, I think so. High-level programming languages tend to be more abstract, so any concreteness is appreciated. However, looking through a student evaluation of my FP course, the students never complain that this information is provided.
9	Yes, very useful. Breaking down in steps explains a lot.
10	This is useful for two reasons: (1) seeing different evaluation strategies at work helps students to get rid of the 'imperative' view that programs specify how something must be computed instead of what must be computed (the declarative view) (2) practicing with performing evaluation steps themselves increases their accuracy at equational reasoning, which is necessary for doing (induction) proofs
11	I think it's useful to illustrate the difference between non-strict and strict languages (see question 7).
12	Yes it is useful because students see how it works and will remember it easier. They can also try with own expressions
13	This shows what is really happening, which allows better understanding of the code.
14	I certainly think it is useful.
15	Yes, it helps understanding how functions 'work. It is nice to see a higher order function in action.
16	It's important to understand the underlying evaluation techniques used by the two approaches. Although it can be confusing. First of all functional programming is different from imperative/OO programming. The latter are mostly the first languages learned. So I think the tool is not suitable for beginners in programming. Secondly it would be more clear what the differences between the two approaches are, it they where presented next to each other on the same page.

- **Question 7:** Do you think it is useful for students following an introduction course in functional programming to view how expressions are evaluated according to different evaluation strategies?

Respondent	Answer
1	This is certainly useful. It can help to understand evaluation mechanisms. However, using this tool may give the student the idea that studying evaluation orders is the *only* way to reason about a program. Applying induction and substitution are also very good ways to understand a program.
3	I think it is useful to follow an introduction course in functional programming, but evaluation expression is not the biggest difference.
6	The understanding of innermost and outermost evaluations is quite natural. Hutton is enough to grasp the concepts.
7	Maybe, it depends on why you would want to teach this. It might be confusing to absolute beginners.
8	again, I think so, particularly if one of these explains how it works in a situation that they are used to (by value), and the other how it is when you use Haskell (lazy).
9	Yes, very useful. Breaking down in steps explains a lot
10	This is useful because this gives the insight that there is always a 'safe' strategy, but it might not always be the most efficient strategy.
11	Yes it is useful. But only on small examples to illustrate how non-strict evaluation works.
12	For very long evaluations it is difficult to keep concentrated. You could add a button for evaluation step by step
13	Same reason as before
14	It is good to know there are different evaluation strategies.
15	Yes.
16	See previous answer. Particularly the presentation of the evaluation.

- **Question 8:** Do you find it useful to add more evaluation strategies?

Respondent	Answer
1	If you really want to understand effects of evaluation orders, this may be useful. For the basic programming understanding this may be too much.
3	Lazy evaluation is important, because this normally doesn't exist in imperative languages
6	More is better
7	Yes.
8	Well, lazy evaluation is important, because I teach students Haskell. So yes, I certainly would like that then. Otherwise, it makes little sense to use the tool in the setting of my course. No others are necessary as far as I can tell. However, it would be really nice to see the influence of something like seq in evaluating expressions. I tried using seq but it only evaluated seq's arguments, not seq itself.
9	For me showing different type of strategies will explain a lot
10	Although sharing is fundamental to obtain efficient evaluation, the idea is easy to get across. Not certain whether it is worth the effort to add to the prototype. The influence of strictness annotations (Haskell seq or Clean !) could be interesting.
11	Yes. Lazyness! Reasoning about lazy evaluation is super subtle. Visualizing this nicely would be very useful indeed.
12	For a beginning student it is not useful. After trying they know that there are different strategy and they only have to remember the strategy for their programming language.
13	I don't know enough of the subject to answer this question.
14	Lazy evaluation would be interesting.
15	Lazy would be really nice.
16	Sure they could be useful, but presentation is important. Put them lazy evaluation, next to eager evaluation on the same page when showing the evaluation strategies. I think it applies to all know strategies: *https://en.wikipedia.org/wiki/Evaluation_strategy*

- **Question 9:** Do you think it is useful, for students following an introduction course in functional programming, to practice with the evaluation steps of a Haskell expression?

Respondent	Answer
1	This is certainly useful. It can help to understand evaluation mechanisms. However, using this tool may give the student the idea that studying evaluation orders is the *only* way to reason about a program. Applying induction and substitution are also very good ways to understand a program. However, I could not find a way to define new functions myself nor could I find which functions were available in the tool.
3	Yes, hands-on practice is always useful
6	It has added value, but pen and paper and thinking hard tends to stick more than pressing a button on the screen I think
7	Yes, it forces a student to think about what happens when applying certain functions.
8	I am not sure at this point whether that is helpful. I'd like them to understand, and providing input by them would help them understand it more thoroughly. But I am not sure it is worth the effort.
9	Useful, it gives a better insight.
10	This is useful because the tool can be used for self-study to give feedback whether or not the student is applying the correct rules.
11	A little. Perhaps on very small examples.
12	Yes, this is very useful. Student have time to think about the next step and keep concentrated.
13	Yes, to give a deeper understanding.
14	Yes, I can image a student needs to predict the next step in the evaluation.
15	Yes, I think it is useful. They learn a bit to reason with programs.
16	It will be useful in the very beginning. I think very quickly the step based approach will be boring, because it takes too long to step though all the steps. Furthermore the tool is intended to give an understanding of the steps performed by the language "underwater". Writing the steps down is no goal.

- **Question 11:** Do you think it is useful to skip certain evaluation steps? And to configure this for example for some evaluation rules?

Respondent	Answer
1	Since the goal is to understand the evaluation, you need to show the full evaluation, all the steps so that you understand that the answer 14 is not reached in one step.
3	Yes, the user should be able to skip steps, when (3+4)+7 is expected as answer, (3+4)+7, 7+7 and 14 should all be accepted as correct.
6	No,a detailed evaluation adds to the understanding of the language
7	Yes.
8	I think that will be necessary for larger programs. But does it make sense to do this for larger programs? I am not so sure. The concepts should be clear from smaller ones as well.
9	Yes
10	It don't think evaluation steps should be skipped. Keep the examples small. Students must be aware that each step is necessary and need to think in what order these occur.
11	If you want to force them to be explicit about every step, do so. Otherwise students may end up skipping steps, without really understanding what's going on.
12	It seems better not to skip some parts because students will take too many steps at a time.
13	When starting to study the language it will be nice to see all the details, but later it should be possible to skip this.
15	That is a good feature to have, especially when the exercises grow larger.
16	Yes, see previous answers

- **Question 12:** Do you think it is useful, for students following an introduction course in functional programming, to practice with the evaluation steps of an expression and to view the derivation of an expression to understand these concepts? Do you know other programming concepts where a tool like this could help students in understanding this concept?

Respondent	Answer
1	In particular for recursion, pattern matching and higher order aspects, evaluation order can surprise you. Try e.g. the following 5 expressions and study their evaluation behaviour: twice inc 0 ; twice twice inc 0 ; twice twice twice inc 0 ; twice twice twice twice inc 0 ; twice twice twice twice twice inc 0
3	True/false evaluations in conditions
6	I think it adds value for every programming concept or algoritm, especially when the course lacks good examples and exercises
7	Yes. I think for absolute beginners the understanding of the programming concepts might even be more important than knowing different evaluation strategies.
8	Yes, I think it is useful. Students in my class have a hard time understanding monads so if you can contribute to that... The rest they seem to be quite okay with.
9	working with datatypes
10	It is useful to get a feeling for the operational behavior of these concepts. A risk is that you lose the level of abstraction: solving problems with recursion and/or higher-order functions is not the same as attempting to do that with the operational behavior of these concepts. In other words: concentrating too much on the operational behavior may trap students back into the 'imperative' mind set.
11	Perhaps if you're learning about folds for the first time, for example.
12	Yes, it is useful, for students following an introduction course in functional programming, to practice with the evaluation steps of an expression and to view the derivation of an expression to understand these concepts. Conditional expressions is also a concept that students should practice.
15	Yes, like I mentioned before.
16	As outlined before. I think the tool is only useful for very beginners to get an understanding of what is going on.

- **Question 13:** Do you think that if you had a tool like this available when you learned a functional language that it helped you with the understanding of how expressions are calculated? And would you use a tool like this?

Respondent	Answer
1	Yes, I do. As an experienced user, I do not think that I would still use it a lot.
3	Yes, and I certainly would have used the tool
6	I would have taken a look at it, but because Hutton and the workbook have plenty examples and excercises I don't think that it would have make a difference.
7	Probably.
8	I do not think so, but then again, I never had a problem understanding it. I would not use it myself now, but I think I would like to offer it to students to improve their understanding. I guess I would also use it in explaining the concept of laziness (if that was supported) during the lecture. Since making slides to show this in animated form of some kind is way too much work.
9	I have used the tool
10	Yes, it would have helped. I wouldn't use it myself, but my students can certainly benefit from it.
11	Maybe. It's hard to say. I've been programming with functional languages for too long now. I wouldn't use it myself, but I can imagine it being useful to students trying to understand very basic concepts like folds/maps/filters etc.
12	Yes, it would be nice.
13	I would probably use it just to experiment a bit, not to support the study. Unless the usage of the tool would be encouraged in the study materials.
15	Yes.
16	See answer question 12

- **Question 14:** Remarks?

Respondent	Answer
1	Good work!
8	In all, I am hoping to see lazy evaluation in your system, in some way. If that happens for before, say September, when my course starts again, I'd certainly consider employing it to illustrate the different between by-value and lazy evaluation. I was not sure about what to answer under 10. since I do not really know what I would gain from having new rules, and how much work it would be to come up with them. Hence the "maybe". For take 3 (repeat 2) it would be nice if you could somehow signal explicitly that innermost evaluation fails. Also, having the code on hand for the expression that are evaluated would be a great help to students.
10	Nice tool! In the tool it would be useful if the actual function definitions can be visualized as well. This makes it easier to see which options are available and how the definition is applied to the term that is rewritten. Bug report: at some point I could no longer select the expression 'take 3 (repeat 2)'; instead it reacted with the first expression '((8 * 8) + 7) - (3 + 4)'. Also, the innermost evaluation of 'take 3 (repeat 2)' simply doesn't react because it doesn't terminate. It is more instructive to show the steps that lead to this non-terminating behavior.
12	Good looking tool!
13	Nice tool. Nice that it also works on mobile!
15	Nice tool!

BIBLIOGRAPHY

[1] Martin Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 31–46, New York, NY, USA, 1990. ACM. (Cited on page 3.)

[2] Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. A call-by-need lambda calculus. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 233–246, New York, NY, USA, 1995. ACM. (Cited on pages 50 and 51.)

[3] Adam Bakewell and Colin Runciman. The space usage problem: An evaluation kit for graph reduction semantics. In *Selected papers from the 2nd Scottish Functional Programming Workshop (SFPoo)*, pages 115–128, Exeter, UK, 2000. Intellect Books. (Cited on page 1.)

[4] Thibaut Balabonski. A unified approach to fully lazy sharing. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 469–480, New York, NY, USA, 2012. ACM. (Cited on pages 50 and 51.)

[5] Henk Barendregt and Erik Barendsen. Introduction to lambda calculus. `ftp://ftp.cs.ru.nl/pub/CompMath.Found/lambda.pdf`, 2000. [Online; accessed 19-October-2013]. (Cited on page 9.)

[6] Zine-El-Abidine Benaissa, Pierre Lescanne, and Kristoffer Høgsbro Rose. Modeling sharing and recursion for weak reduction strategies using explicit substitution. In *Proceedings of the 8th International Symposium on Programming Languages: Implementations, Logics, and Programs*, PLILP '96, pages 393–407, London, UK, 1996. Springer-Verlag. (Cited on page 51.)

[7] Richard S. Bird and P. Wadler. *Introduction to functional programming using Haskell*. Prentice-Hall, 1998. (Cited on page 1.)

[8] Manuel M.T. Chakravarty and Gabriele Keller. The risks and benefits of teaching purely functional programming in first year. *Journal of Functional Programming*, 14(1):113–123, 2004. (Cited on pages 1 and 6.)

[9] Arthur Charguéraud. Pretty-big-step semantics. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 41–60, Berlin, Heidelberg, 2013. Springer-Verlag. (Cited on page 51.)

[10] Olaf Chitil, Colin Runciman, and Malcolm Wallace. Transforming Haskell for Tracing. In *Proceedings of the 14th International Conference on Implementation of Functional Languages*, IFL'02, pages 165–181, Berlin, Heidelberg, 2003. Springer-Verlag. (Cited on page 18.)

[11] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM. (Cited on page 52.)

[12] Zoltán Csörnyei and Gergely Dévai. Central European Functional Programming School. chapter An Introduction to the Lambda Calculus, pages 87–111. Springer-Verlag, Berlin, Heidelberg, 2008. (Cited on page 9.)

[13] Alex Gerdes. *Ask-Elle: a Haskell Tutor*. PhD thesis, Open Universiteit Nederland, 2012. (Cited on pages 20, 22, and 39.)

[14] Alex Gerdes, Bastiaan Heeren, and Johan Jeuring. Teachers and students in charge: using annotated model solutions in a functional programming tutor. In *Proceedings of the 7th European conference on Technology Enhanced Learning*, EC-TEL'12, pages 383–388, Berlin, Heidelberg, 2012. Springer-Verlag. (Cited on page 19.)

[15] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning Haskell. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, Haskell '03, pages 62–71, New York, NY, USA, 2003. ACM. (Cited on page 22.)

[16] Bastiaan Heeren, Johan Jeuring, and Alex Gerdes. Specifying rewrite strategies for interactive exercises. *Mathematics in Computer Science*, 3(3): 349–370, 2010. (Cited on pages 22, 30, and 31.)

[17] Paul Hudak. *The Haskell school of expression: learning functional programming through multimedia*. Cambridge University Press, 2000. (Cited on page 1.)

[18] Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997. (Cited on pages 28 and 33.)

[19] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007. (Cited on pages 1, 5, 7, 9, 11, 12, 17, 29, 40, 46, 50, and 51.)

[20] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Jones Cordy, Hall Kevin, Will Partain, and Phil Wadler. The Glasgow Haskell compiler: a technical overview. `http://www.research.microsoft.com/Users/simonpj/Papers/grasp-jfit.ps.Z`, 1992. [Online; accessed 19-October-2013]. (Cited on page 17.)

[21] Stef Joosten, Klaas Van Den Berg, and Gerrit Van Der Hoeven. Teaching functional programming to first-year students. *Journal of Functional Programming*, 3:49–65, January 1993. (Cited on page 1.)

[22] Roelof Kooiker. *Marktonderzoek*. Wolters-Noordhoff, 1997. (Cited on page 45.)

[23] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 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, pages 14–18, New York, NY, USA, 2005. ACM. (Cited on pages 1, 6, and 19.)

[24] Konstantin Laufer and George K. Thiruvathukal. Scientific programming: The promises of typed, pure, and lazy functional programming: Part ii. *Computing in Science and Engineering*, 11(5):68–75, September 2009. (Cited on page 5.)

[25] John Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 144–154, New York, NY, USA, 1993. ACM. (Cited on pages 3 and 51.)

[26] Miran Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, 1st edition, 2011. (Cited on pages 1, 5, 6, 7, 8, 11, 12, and 27.)

[27] Natalia López, Manuel Núñez, Ismael Rodríguez, and Fernando Rubio. WHAT: Web-based Haskell adaptive tutor. *Lecture Notes in Computer Science*, 2443:71–80, 2002. (Cited on pages 19 and 47.)

[28] Simon Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilly Media, Inc., 2013. (Cited on page 34.)

[29] Greg Michaelson. *An introduction to Functional Programming Through Lambda Calculus*. Dover Publications, Inc., 1st edition, 2011. (Cited on pages 5 and 6.)

[30] Ben Millwood. stepeval library: Evaluating a haskell expression step-by-step. `https://github.com/bmillwood/stepeval`, 2011. [Online; accessed 25-January-2014]. (Cited on page 18.)

[31] Keiko Nakata and Masahito Hasegawa. Small-step and big-step semantics for call-by-need. *Journal of Functional Programming*, 19(6):699–722, November 2009. (Cited on page 51.)

[32] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008. (Cited on pages 1, 2, 22, 27, and 35.)

[33] Cristóbal Pareja-Flores, Jamie Urquiza-Fuentes, and J. Ángel Velázquez-Iturbide. WinHIPE: an IDE for functional programming based on rewriting and visualization. *SIGPLAN Not.*, 42(3):14–23, March 2007. (Cited on pages 6 and 18.)

[34] Roly Perera, Umut A. Acar, James Cheney, and Paul Blain Levy. Functional programs that explain their work. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 365–376, New York, NY, USA, 2012. ACM. (Cited on page 1.)

[35] Jan Rochel. 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, pages 198–217, Berlin, Heidelberg, 2010. Springer-Verlag. (Cited on page 3.)

[36] Peter Sestoft. The essence of computation. chapter Demonstrating Lambda Calculus Reduction, pages 420–435. Springer-Verlag New York, Inc., New York, NY, USA, 2002. (Cited on page 3.)

[37] Jan Sparud and Colin Runciman. Tracing lazy functional computations using redex trails. In *Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs: Including a Special Trach on Declarative Programming Languages in Education*, PLILP '97, pages 291–308, London, UK, 1997. Springer-Verlag. (Cited on pages 1, 6, and 18.)

[38] Thomas van Noort, Alexey Rodriguez Yakushev, Stefan Holdermans, Johan Jeuring, Bastiaan Heeren, and José Pedro Magalhães. A lightweight approach to datatype-generic rewriting. *Journal of Functional Programming*, 20:375–413, 7 2010. (Cited on page 29.)

[39] Arto Vihavainen, Matti Paksula, and Matti Luukkainen. Extreme apprenticeship method in teaching programming for beginners. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 93–98, New York, NY, USA, 2011. ACM. (Cited on page 19.)

[40] David A. Watt. *Programming Language Design Concepts*. John Wiley & Sons, 2004. (Cited on pages 9 and 10.)