A Generic Framework for Developing Exercise Assistants

Johan Jeuring

Harrie Passier

Sylvia Stuurman

Department of Information and Computing Sciences, Utrecht University

Technical Report UU-CS-2007-017

www.cs.uu.nl

ISSN: 0924-3275

A Generic Framework for Developing Exercise Assistants

Johan Jeuring,
Utrecht University and Open University, the Netherlands
johanj@cs.uu.nl

Harrie Passier and Sylvia Stuurman
Open University, the Netherlands
{Harrie.Passier,sylvia.stuurman}@ou.nl

July 19, 2007

Abstract

We are developing several exercise assistants which give very good feedback. To develop an exercise assistant, we need three components: a domain description, rules for reasoning about the domain, and one or more strategies for solving the exercises in the domain. To support the possibility to adapt the several components, to avoid code duplication and maximize code reuse, we have developed a generic framework for developing exercise assistants. The generic framework for developing exercise assistants consists of several components: a component for determining the difference between two expressions, a component for rewriting terms, etc.

1 Introduction

At the Open University NL and Utrecht University, we are developing several exercise assistants. We have developed an exercise assistant that supports interactively solving a system of linear equations [9], and we have developed an assistant that supports calculating a disjunctive normal form (DNF) of a logical expression [7]. Extensions of the former tool can be used in linear algebra courses at universities, the second tool will be used in the discrete mathematics course in our Computer Science curriculum. A minimized screenshot of the assistant that supports calculating a DNF of a logical expression is shown in Figure 1. More exercise assistants will appear this year. The distinguishing feature of our assistants is that they give very good feedback. Although giving good feedback is generally acknowledged to be vital for learning [8], current e-learning systems that support incrementally solving exercises lack sophisticated techniques for giving feedback. We hope to improve upon this situation.

To develop an exercise assistant, we need three main components: a domain description (for example: systems of linear equations, or logical expressions), together with a concrete representation of the domain (how are the expressions presented to the student); rules for reasoning about the domain (for example: multiplication distributes over addition, de Morgan for logical expressions); and one or more strategies for solving exercises in the domain (for example: first move occurrences of $\neg$ in front of logical variables, and then distribute $\wedge$ over $\vee$ to obtain an expression in DNF). According to Bundy [2], these three components are the essential aspects when modelling any kind of intelligence.

For an e-learning tool to be successful, users have to be able to adapt the presentation of domains, the solving strategy of exercises, the kinds of exercises, and probably even the domains themselves. Furthermore, when improving our tools in a non-domain-specific way, we do not want to repeat the same improvements for all the tools we have built. To support the possibility to adapt the several components, to avoid code duplication and maximize code reuse, we have developed a generic framework for developing exercise assistants.

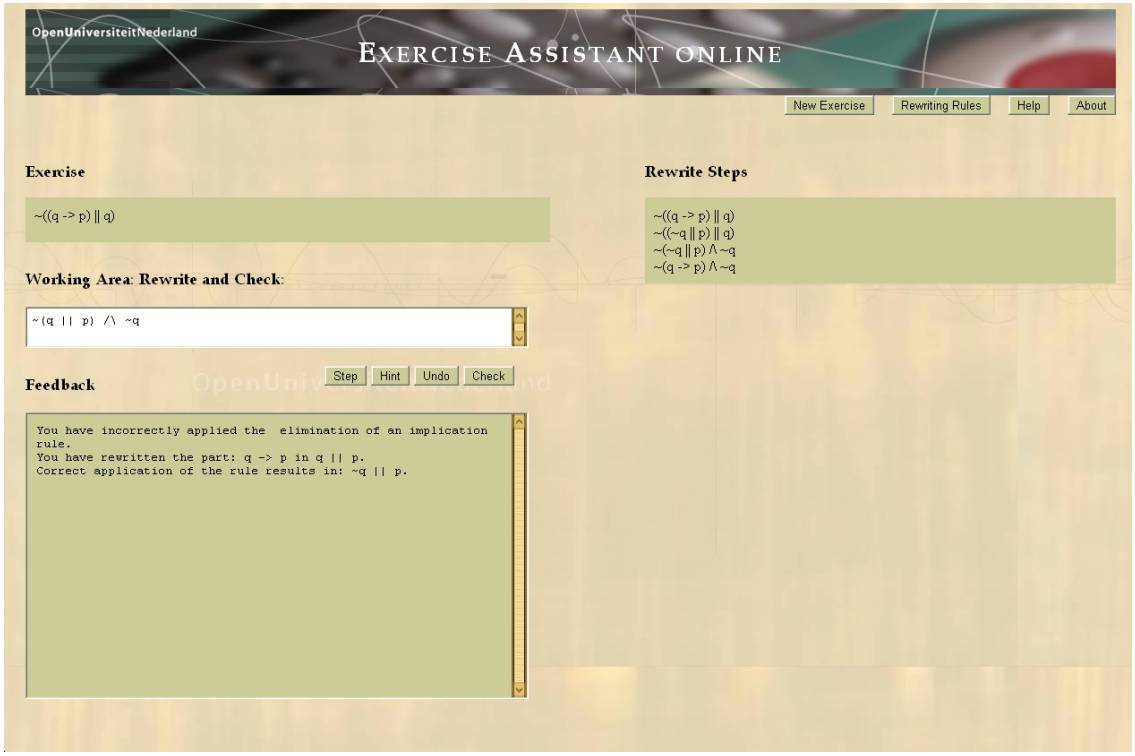

Figure 1: The Online Exercise Assistant

The generic framework for developing exercise assistants consists of several generic components: for example a component for recognizing strategies, a component for determining the difference between two expressions, a component for rewriting terms, etc. In this paper we will present the generic framework for developing exercise assistants and briefly describe the generic components. Using this framework, teachers can easily adapt the exercise assistant to their needs and wishes.

This paper is structured as follows. Section 2 introduces our exercise assistants. We describe their characteristics, the user interface, and the main components of the exercise assistants. We will use our exercise assistant that supports solving exercises about calculating a DNF of a logical expression as the running example in this paper. Section 3 discusses the functionality that is different per exercise assistant, but for which we can develop a generic framework that helps in generating components for this functionality, depending on the domain for an exercise assistant. Section 4 discusses implementation aspects of these generic components. Section 5 discusses future work and concludes.

2 The exercise assistants

This section introduces the exercise assistants we are building and using in some of our courses. Figure 1 shows the on-line exercise assistant that supports transforming a logical expression to DNF.

2.1 Characteristics

The exercise assistants we are building have the following characteristics [9]:

- An assistant is interactive, so a student solves an exercise step by step and receives semantically rich feedback after each erroneous step. Using this feedback, students can correct their

mistakes and adjust their solving strategy.

- A student enters and rewrites expressions in the working area. This approach mimics the pen-and paper situation as closely as possible; students can make syntactical as well as semantical mistakes. We envisage that we will provide buttons which a student can click to let the tool perform rewrites on (a selection of) the expression, but this is part of future work.
- Feedback is produced for a whole class of problems. For example, the exercise assistant that supports calculating the DNF of a logical expression, gives feedback about any errors made in any of the exercises. The feedback need not be specified with every exercise, but is automatically calculated based on the exercise, the available rewrite rules, possibly the known "buggy" rewrite rules [1], and the expression entered by the student. Moreover, the feedback is not hard coded, but is generated algorithmically on the level of rewrite steps.

2.2 The user interface

The user interface of the exercise assistant, shown in Figure 1, consists of four text fields and nine buttons. We have used this interface for the exercise assistant supporting solving linear equations as well as the one that supports calculating the DNF of a logical expression.

The text fields. The top-left text field shows the exercise, in this case the logical formula which a student has to transform to DNF. The current formula to be rewritten is $\neg((q \to p) \vee q)$. The second text field is the working area in which the student stepwise edits the logical formula into a DNF. The current formula in the working area is: $\neg(q \vee p) \wedge \neg q$. The third text field displays the feedback after a formula is submitted. In the figure it says that the student has incorrectly applied the elimination of an implication rule. Furthermore, it says which part of the expression is incorrectly rewritten and shows how the rule is correctly applied. The last text field, on the right, shows the derivation so far. The formula on top is the original formula of the exercise. In the figure the student has performed three additional steps: $\neg((\neg q \vee p) \vee q)$, $\neg(\neg q \vee p) \wedge \neg q$ and $\neg(q \to p) \wedge \neg q$.

The buttons. A student clicks the Check button to submit an edited formula, and the Undo button to undo the last step (or any amount of steps). If a student wants help, he or she clicks the Hint or Step button. Clicking the Hint button gives a suggestion about how to proceed, whereas clicking the Step button gives a detailed next step. If for example the current formula is $\neg q \vee \neg(t \wedge t)$, clicking the Hint button gives the message: *"You can apply the De Morgan rule"*, and clicking the Step button gives the message *"Apply the De Morgan rule on:* $\neg(t \wedge t)$*, giving:* $\neg t \wedge \neg t$*"*. The Finished button only appears if the expression in the work area is syntactically and semantically correct. A student clicks the Finished button if he or she thinks the formula is in DNF. If the formula is not yet in DNF, the exercise assistant gives the massage: *"You have not yet reached a DNF yet"*, and otherwise *"Your logical formula is in DNF"*. If a student has solved an exercise, but does not click the Finished button, the exercise assistant gives the feedback *"You have already reached a DNF"*. If the student clicks the New exercise button a new exercise is generated and presented in the left top text field and the working area. Clicking the button Rewrite rules gives a list of all rewrite rules which can be used during the solving process. Clicking the Help button gives an overview of how the exercise assistant can be used, which symbols are used, etc. The About button gives high-level information about the exercise assistant.

Feedback messages. A student can make different kinds of mistakes when solving an exercise in our exercise assistants:

- syntactical mistakes, for example when a student writes $\neg(q \wedge (t\wedge))$ instead of $\neg(q \wedge (t \wedge t))$ (the and operator $\wedge$ needs two arguments) or a formula with a missing parenthesis;

- semantical mistakes, such as forgetting to change a disjunction into a conjunction when applying the De Morgans rule. For example $\neg(t \wedge t)$ is rewritten to $\neg t \wedge \neg t$ instead of $\neg t \vee \neg t$.

2.3 The main components

To model intelligence in a computer program, Bundy [2] identifies three important, basic needs:

- The need to have knowledge about the domain.
- The need to reason with that knowledge.
- The need for knowledge about how to direct or guide that reasoning.

Our exercise assistants take instantiations of these components as input. Each exercise assistant needs a domain description, rules that tell how expressions from the domain may be rewritten, and strategies that guide a student toward a solution.

This subsection briefly illustrates these three components for the domain of logical expressions. Passier and Jeuring [9] describe the components for the domain of systems of linear equations. We will illustrate the components with small, illustrative pieces of code from the programming language in which we implemented our tool for this domain.

To implement our tools, we need functionality for parsing, pretty-printing, symbolic evaluation, several analyses, etc. This functionality builds, traverses or folds abstract syntax trees. Furthermore, the exercise-assistant tools for the different domains (logical expressions, linear equations) are very similar, and we want to minimize code duplication. The lazy higher-order functional programming language Haskell [10] is particularly good at manipulating abstract syntax trees, and the high level of abstraction supported by Haskell minimizes code duplication, so we have implemented our tools in Haskell.

Domain description. The domain of the exercise assistant that supports calculating a DNF of a logical expression is the domain of propositional formulae, which is usually defined as follows in logic textbooks :

- Each of the two constants TRUE and FALSE is a propositional formula.
- Logical variables, such as $p, q, r, ...$ are propositional formulae.
- If α and β are propositional formulae, then so are $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $(\alpha \rightarrow \beta)$, $(\alpha \leftrightarrow \beta)$ and $\neg\alpha$.
- Nothing else is a propositional formula.

This form of the formulae is shown to users of the exercise assistant. It is the concrete representation of our domain. Internally in our tool we use an abstract syntax. Using Haskell data types we represent the domain of logical expressions as follows:

```
data Logic  =
     T                -- true
  |  F                -- false
  |  Var String       -- variable
  |  Logic :&&:  Logic -- and/conjunction
  |  Logic :||:  Logic -- or/disjunction
  |  Logic :->:  Logic -- implication
  |  Logic :<->: Logic -- equivalence
  |  Not Logic         -- not
```

A propositional formula is represented using the recursive data type `Logic`. For example $\neg(t \wedge t)$ is represented as `Not(Var "t" :&&: Var "t")`.

Rewrite rules. Each domain has a set of rewrite rules with which terms of the domain can be manipulated. A rewrite rule consists of a pair (*lhs*, *rhs*) of terms, denoted by $lhs \Longrightarrow rhs$, in which variable terms may appear. Rewriting a term by means of such a rewrite rule is done by matching the term with the left-hand side of the rule, and using the resulting bindings obtained by matching to obtain the resulting term from the right-hand side term.

As an example, some of the rewrite rules for logical expressions are the De Morgan rules: $\neg(\alpha \vee \beta) \Longrightarrow \neg\alpha \wedge \neg\beta$, and $\neg(\alpha \wedge \beta) \Longrightarrow \neg\alpha \vee \neg\beta$. When we apply the latter rule to the logical formula $\neg(t \wedge t)$, we bind both α and β to t, and obtain the following rewritten expression: $\neg t \vee \neg t$. Here are some more examples of rewrite rules for logical formulae.

$$
\begin{array}{lrcl}
\textsc{TrueUnitAnd}: & \textsc{True} \wedge x & = & x \\
\textsc{FalseZeroAnd}: & \textsc{False} \wedge x & = & \textsc{False} \\
\textsc{AndComm}: & x \wedge y & = & y \wedge x \\
\textsc{AndAssoc}: & x \wedge (y \wedge z) & = & (x \wedge y) \wedge z
\end{array}
$$

Logical expressions form a boolean algebra, and hence there exist a number of rules for logical expressions, such as True is the unit of $\wedge$, False is the zero of $\wedge$, and $\wedge$ is commutative and associative. Each rule is given a name.

The complete set of rewrite rules for logical formulae consists of around 15 rules. Internally, rewrite rules are implemented using pattern matching.

```
deMorgan :: Formula -> Maybe Formula
deMorgan (Not (a :&&: b)) =
         Just (Not a :||: Not b)
deMorgan (Not (a :||: b)) =
         Just (Not a :&&: Not b)
deMorgan _                = Nothing
```

Function `deMorganRule` takes a formula as argument and produces the rewritten formula as output. If for example function `deMorganRule` is applied to `Not(Var "t" :&&: Var "t")`, the argument matches the first pattern `(Not (a :&&: b))`, and the function returns the formula `Just(Not (Var "t") :||: Not (Var "t"))`. If the argument supplied doesn't match the first two patterns, it matches with the wild-card _, and the function returns `Nothing`.

Strategies. For our exercise assistant that supports transforming a logical expression to DNF, we need a *strategy*. Informally, the DNF of a logical expression is of the form $(\ldots \wedge \ldots \wedge \ldots) \vee \ldots \vee (\ldots \wedge \ldots \wedge \ldots)$, where negations only appear in front of propositional variables. To rewrite a logical expression to DNF, the basic rules for logical expressions have to be combined to describe how a logical expression is rewritten to DNF. One possible strategy for rewriting logical formulas to DNF is to

- first eliminate all $\neg$'s that are not in front of an expression variable by means of any of the rules for $\neg$.
- Then bottom-up eliminate all $\vee$'s that appear below top level, using the rule that says that $\wedge$ distributes over $\vee$.

Both of these two parts have to be applied until they cannot be applied anymore. This is an informal description of the strategy for transforming a logical formula to DNF, in the exercise assistant we need an explicit representation of this strategy. It is beyond the scope of this paper to give this formal definition [6].

3 The framework

As described in the previous section, the main components of our exercise assistants are the domain, rewrite rules on the domain, and strategies on the domain. We do not want to implement a

separate exercise assistant for each domain, since there are many of these domains (in mathematics, logic, computer science, physics, biology, statistics, etc). Furthermore, the implementation of the different exercise assistants is very similar: the user interface, the web application, the communication between the different components, etc, are all exactly the same. The difference between the different assistants only depends on the description and structure of the domain. We want to *generate* an exercise assistant given the description of the main components, in particular the domain. This section describes the generic framework we use for building exercise assistants. The generic framework

- makes it much easier to build exercise assistants, by minimizing programming effort and maximizing code reuse,
- forces developers to be explicit about the essential components of an exercise assistant, and
- makes it possible to produce high quality tools, by focussing implementation efforts on the essential components.

We will focus the description on the components that differ per exercise assistant. These components deal with:

- rewriting expressions,
- determining the distance between two expressions,
- traversing expressions,
- selecting within expressions.

We will introduce each of these components by means of some examples. In the next section we will discuss implementation issues for these components.

Rewriting. After a student has rewritten and submitted a logical formula, the exercise assistant analyses this rewriting. It tries to determine if the formula has been correctly rewritten, and if so, which rewrite rule has been applied. The result of this analysis is a feedback message. For example, a student may rewrite (in the working area) the formula $\neg q \vee \neg(t \wedge t)$ into $\neg q \vee (\neg t \vee \neg t))$ using the De Morgan rule. And the feedback would then be something like *"This rewrite step is a correct application of the De Morgan rule"*. If the rewrite step is incorrect, we try to determine the most likely mistake, and give that as feedback. For example, if a student rewrites the formula $\neg q \vee \neg(t \wedge t)$ into $\neg q \vee (\neg t \wedge \neg t)$, we give as feedback that the student probably tried to apply the De Morgan rule, but has applied it incorrectly.

At the moment we assume a student applies only one rewrite rule per submitted formula. In practice, this will not always be the case. We are working on recognizing the application of multiple rewrite rules.

The rewrite analysis consists of a number of steps. First we determine whether or not the rewritten expression is correct, that is, whether its solution is the same as the solution of the exercise. If it is correct we try to determine the rewrite rule that has been applied, by calculating the subexpressions that have been deleted from the old expression, and the expressions that have been inserted in the new expression. We try to find a rewrite rule that rewrites the deleted subexpressions into the inserted subexpressions, by trying all rewrite rules on the deleted expressions, and comparing the results of these rewrites with the inserted expressions. If there is a match, we report the rule, if there is no match, we give as feedback that the current expression is correct, but that we cannot find the rule that has been applied. If the rewritten expression is incorrect, we try to find a likely rewrite rule the student tried to apply. We do this by rewriting the old expression in all possible ways, and by calculating the edit distance between the resulting terms and the new expression submitted by the student. We report the rule that returns an expression with minimum edit distance to the new expression. The next paragraph explains the minimum edit distance.

Continuing our example in which $\neg q \vee \neg(t \wedge t)$ is rewritten into $\neg q \vee (\neg t \vee \neg t)$, the subexpression deleted is $\neg(t \wedge t)$ and the subexpression introduced is $\neg t \vee \neg t$. Only the De Morgan rewrite rule succeeds on the argument $\neg(t \wedge t)$, and applying it gives $\neg t \vee \neg t$. All other rewrite rule applications return `Nothing`. This rule explains the introduced subexpressions from the deleted subexpressions.

Minimum edit distance. When a student makes a semantical mistake, the exercise assistant takes the deleted subexpressions from the old expression and tries to rewrite these using all rewrite rules, giving a list of successful rewritings. The exercise assistant determines an expression in the list of successful rewritings that is *closest* to the introduced subexpressions in the new expression from the student. The exercise assistant gives feedback about the rule that leads to this expression. A closest rewriting is determined by using a technique called *minimal-edit-distance*, which we will briefly explain.

The difference between two values can be expressed as a sequence of edit steps that transforms one value into another. For example, the difference between the two lists [1,2,3] and [1,2,4] can be expressed as "delete the third value in the input and insert 4 at the same position". An edit step inserts or deletes a certain subtree at a certain position. If we assign a cost to each of the edit steps (probably also based on the size of the inserted or deleted tree), then we can define the minimum edit distance between two values as the cost of the cheapest edit sequence that will transform one value into another. This technique is called minimum-edit-distance.

For example, if a student rewrites the formula $\neg q \vee \neg(t \wedge t)$ into $\neg q \vee (\neg t \wedge \neg t)$, we apply all rewrite rules for logical expressions to the deleted subexpressions in the old expression, $\neg(t \wedge t)$. Only the De Morgan rule on the right argument matches, giving $\neg t \vee \neg t$. Since there is only one possible rewrite, this is by definition the closest expression to the new expression entered by the student, so we report this as the rule we think the student intended to apply.

Term traversals. As mentioned in the previous section, a strategy for rewriting logical formulas to DNF is to first eliminate all $\neg$'s that are not in front of an expression variable by means of any of the rules for $\neg$, and then bottom-up eliminate all $\vee$'s that appear below top level, using the rule that says that $\wedge$ distributes over $\vee$. To perform a rewrite rule bottom-up, or top-down, we first have to traverse to the lowest level in the term, try the rewrite rule, and then move up. It follows that we need functions for traversing terms. Here we use the usual strategic combinators for term transformations [11].

Selections. One of the extensions to the exercise assistant that we foresee is that students may select a subexpression, and ask for possible transformations for that subexpression. In the web-enabled version of the exercise assistant, the validity of a subexpression should preferably be checked at the client-side.

The web-enabled version of the exercise assistant uses exactly the same source code as the desktop variant. An application server in Haskell [5] offers light-weight web services: a service without an input parameter, offering a new exercise, and a service with an expression and a rewriting of that expression as input parameters, offering semantically rich feedback. The client-side of the web-enabled version uses Javascript to use these light-weight web services, using the Ajax approach [3] for an optimal response-time.

We want to develop a third web service, which offers a set of possible transformations for a given subexpression within in expression. It will try the rewrite rules for the domain of the expressions on the subexpression, and return the rwrite rules that match. With respect to response time, it is desirable to let a client check whether or not a given selection of an expression is a valid subexpression. In that case, the client will also need to check for syntax errors before sending a step in the solution process to the server.

Because Javascript is a language that treats functions as first-class elements, we are able to evaluate the validity of a subexpression in a given domain using the same generic techniques as the framework in Haskell.

4 Implementing generic components

The four components described in the previous section are functions that are different for each domain on which they are instantiated, but they only depend on the structure of the domain. A function that only depends on the structure of a domain is called a generic function. Such a

generic function is written once, and an instance of a generic function on a particular domain can be automatically generated. Generic programming techniques are widely available for Haskell [4].

We have implemented all the components from the previous section as generic functions.

5 Future work and conclusions

We have introduced a generic framework for developing exercise assistants that give semantically rich feedback. Exercise assistants are useful in many, different domains. Implementations of the different exercise assistants can reuse a lot of code, for example for the user interface and the web application. We have distinguished four components that need to be reimplemented for each exercise assistant. We have implemented each of these components as a generic function, and we now have the tools to automatically generate exercise assistants from domain descriptions.

Although all the tools for generating exercise assistants have been implemented, we still have to test the generation process, and put all the different components together.

Acknowledgments. Thomas van Noort and John van Schie implemented a generic rewriting library. Eric Bouwers and Mark Snyder implemented a first version of a generic minimum edit distance library. Martijn van Steenbergen and Jeroen Leeuwestein implemented a generic selection library.

References

[1] J. S. Brown and K. VanLehn. Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science*, 4:379–426, 1980.

[2] A. Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press, 1983.

[3] J. Garrett. Ajax: A new approach to web applications, 2005. `http://www.adaptivepath.com/publications/essays/archives/000385.php`.

[4] R. Hinze, J. Jeuring, and A. Löh. Comparing approches to generic programming in Haskell. In R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors, *Generic Programming, Advanced Lectures*, volume 4719 of *LNCS*. Springer-Verlag, 2007.

[5] A. Jacobson. HAppS – haskell application server. `http://happs.org/`, 2006. `http://happs.org/`, accessed June 2007.

[6] J. Jeuring and W. Pasman. Strategy feedback in an e-learning tool for mathematical exercises. In *Proceedings Workshop on e-Learning, Leipzig*, 2007. Also appeared as Technical report Utrecht University UU-CS-2007-007.

[7] J. Lodder, J. Jeuring, and H. Passier. An interactive tool for manipulating logical formulae. In M. Manzano, B. Pérez Lancho, and A. Gil, editors, *Proceedings of the Second International Congress on Tools for Teaching Logic*, 2006.

[8] E. Mory. Feedback research revisited. In D. Jonassen, editor, *Handbook of research for educational communications and technology*, 2003.

[9] H. Passier and J. Jeuring. Feedback in an interactive equation solver. In M. Seppälä, S. Xambo, and O. Caprotti, editors, *Proceedings of the Web Advanced Learning Conference and Exhibition, WebALT 2006*, pages 53–68. Oy WebALT Inc., 2006.

[10] S. Peyton Jones et al. *Haskell 98, Language and Libraries. The Revised Report*. Cambridge University Press, 2003. A special issue of the Journal of Functional Programming.

[11] E. Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005. Special issue on Reduction Strategies in Rewriting and Programming.