Constructing Strategies for Programming

Alex Gerdes

Bastiaan Heeren

Johan Jeuring

Technical Report UU-CS-2008-049
January 2009

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

ISSN: 0924-3275

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

CONSTRUCTING STRATEGIES FOR PROGRAMMING
Preparation of Camera-Ready Contributions to INSTICC Proceedings

Alex Gerdes, Bastiaan Heeren
Faculty of Computer Science, Open Universiteit, The Netherlands
alex.gerdes@ou.nl, bastiaan.heeren@ou.nl

Johan Jeuring
Faculty of Computer Science, Open Universiteit, The Netherlands, and
Department of Information and Computing Science, Universiteit Utrecht, The Netherlands
johanj@cs.uu.nl

Keywords: Strategies, intelligent tutoring systems, programming, feedback.

Abstract: Learning to program is difficult. To support learning programming, many intelligent tutoring systems for learning programming have been developed. Research has shown that such tutors have positive effects on learning. However, intelligent tutors for learning programming are not widely used. Building an intelligent tutor for a programming language is a substantial amount of work, and utilising it in a course is often hard for a teacher. In this paper we illustrate how to construct strategies for solving programming exercises and how these strategies can be used to automatically support students using an intelligent programming tutor to incrementally develop a program. Using strategies for programming, specifying an exercise becomes relatively easy, and more flexible.

1 INTRODUCTION

Learning to program is difficult. A first course in programming is often a major stumbling block. To support learning programming, many intelligent tutoring systems for learning programming have been developed. Studies show the positive effects of various tutors on learning programming. However, intelligent tutors for learning programming are not widely used. Building an intelligent tutor for a programming language is a substantial amount of work, and using an intelligent tutor in a course is often hard for a teacher. Most teachers want to adapt or extend an intelligent programming tutor to their needs, which is often hard or impossible.

Part of the reason why it is hard for a teacher to adapt or extend an intelligent programming tutor to his needs, is that it is often a lot of work to specify a new exercise, together with the desired behaviour upon errors. In this paper we show how we can construct strategies for solving programming exercises, and how these strategies can be used to automatically give feedback and hints to students using an intelligent programming tutor to incrementally develop a program. We restrict ourselves to a tutor for learning the functional programming language Haskell (Peyton Jones et al., 2003). We believe, however, that our approach based on programming strategies is also applicable to other programming languages and programming paradigms.

This paper has the following contributions:

- We present a strategy language and some program refinement rules, and we discuss how these can be used in an intelligent tutoring system for learning programming.
- We show how strategies can be derived automatically from model solutions.
- We develop special programming strategies for higher-order functions, and suggest a taxonomy for classifying programming tasks.

The last two contributions are somewhat technical in nature, and we illustrate these by means of some concrete examples.

This paper is organised as follows. Section 2 discusses how students learn programming and how intelligent tutoring systems for programming can help students. We then introduce strategies in Section 3, and we discuss the role strategies play in programming. Section 4 shows how we derive strategies from model solutions, and how strategies for common higher-order functions are specialised. These

specialised strategies lead to a taxonomy, which can be used for classifying programming tasks. Examples of how our strategies can be used are given in Section 5. The last section discusses related work and concludes.

2 LEARNING TO PROGRAM

Programming is a complex cognitive skill (Merriënboer et al., 1992). Especially novice students encounter difficulties when trying to translate a problem description into a series of solution steps to solve the problem. A first course in programming is often a major stumbling block (Proulx, 2000).

The topic of how students learn to program has been studied extensively, by computer scientists, educational scientists, and cognitive psychologists. How do students acquire a complex skill like programming? When a student has to write a program that takes a list of integers as argument, and returns the sum of the integers, one of the first steps is to distinguish the empty list from the non-empty list case. Distinguishing these two cases can be viewed as a *production rule*. (Anderson, 1993) and his colleagues have developed the ACT-R theory, which says that the knowledge underlying a skill begins with an elaborated example, followed by problem solving by analogy. By applying the skill, the student internalises the production rule used in the exercise. With practice, production rules acquire strength and become more attuned to the circumstances in which they apply. Learning complex skills can be decomposed into learning individual production rules, and strategies for combining them. A similar approach to learning programming is taken by (Merriënboer et al., 1992). They present a four-component instructional design model for the training of complex cognitive skills. For learning computer programming, the design model emphasises the importance of worked-out examples. In a later stage steps are removed from the worked-out examples. These missing steps have to be added by a student. Only after these stages should students work out complete programs themselves.

2.1 Intelligent tutoring systems for programming

There exist intelligent tutors for Lisp (Anderson et al., 1986), Prolog (Hong, 2004), Pascal (Johnson and Soloway, 1985), Java (Sykes and Franek, 2004; Kölling et al., 2003), Haskell (López et al., 2002), and many more programming languages. Some of these tutors are well-developed tutors extensively tested in classrooms, others have not outgrown the research prototype phase yet. Evaluation studies have indicated that:

- working with an intelligent tutor supporting the construction of programs is more effective when learning how to program than doing the same exercise "on your own" using only a compiler, or just pen-and-paper (Anderson and Skwarecki, 1986);
- students using intelligent tutors require less help from a teacher while showing the same performance on tests (Odekirk-Hash and Zachary, 2001);
- using such tutors increases the self-confidence of female students (Kumar, 2008);
- the immediate feedback given by many of the tutors is to be preferred over the delayed feedback common in classroom settings (Mory, 2003).

Despite the evidence for positive effects of using intelligent programming tutors, they are not widely used. An important reason is that building an intelligent tutor for a programming language is difficult and a substantial amount of work (Pillay, 2003). Furthermore, using an intelligent tutor in a course is often hard for a teacher. Most teachers want to adapt or extend an intelligent programming tutor to their needs. Adding an exercise to a tutor requires investigating which strategies can be used to solve the exercise, what the possible solutions are, and how the tutor should react to behaviour that does not follow the desired path. All this knowledge then has to be translated into the internals of the tutor, which implies a substantial amount of work. For example, completely specifying feedback in (much simpler) mathematical exercises results in exercise files of hundreds of lines (Cohen et al., 2003).

In comparison, intelligent tutors for mathematics such as ActiveMath (Melis et al., 2001), APlusix (Chaachoua et al, 2004), MathPert (Beeson, 1990), to mention just a few, are much more widely spread and used than intelligent programming tutors. Mathematics has a number of advantages compared with programming: the mathematical language of expression is much more stable than most programming languages, many mathematical problems are relatively easy compared with programming problems, often there is a unique solution to a mathematical problem, and, finally, checking correctness of intermediate steps is much easier because many mathematical problems are solved by applying meaning-preserving transformations or rewrite steps to an expression. These properties of mathematics make it easier to give

feedback to users of an intelligent tutor, both at intermediate steps as at the end.

3 STRATEGIES FOR PROGRAMMING

Procedural skills can be described by production systems. (Anderson, 1993) shows that many of the characteristics of such systems are similar to how students solve problems, and hence that they are psychologically plausible. However, psychological plausibility does not imply ease of usability. Models in ACT-R are rather low-level, and tend to get quite large.

A procedural skill is often called a strategy, and there exist programming languages supporting the formulation of strategies (Visser et al., 1998; Borovanský et al., 2001). Using such a language for defining procedural skills is much easier than using production systems, since common programming language techniques, such as abstraction, modularity, and typing are readily available. As long as the feedback students get is psychologically plausible, the form of a language for describing procedural skills can be optimised to make it as easy as possible to specify such skills, and to make it easy to produce the desired feedback.

(Heeren et al., 2008) have developed an embedded domain-specific language for specifying strategies for exercises. The strategy language can be used for any domain based upon rewrite rules, and can be used to automatically calculate feedback on the level of strategies, given an exercise, the strategy for solving the exercise, and student input (Heeren and Jeuring, 2008). The specification of a strategy and the calculation of feedback is separated: the same strategy specification can be used to calculate different kinds of feedback.

We can use this strategy language to specify exercises in programming: the only additional concept we have to add to this language is refinement rules, which refine programs. For example, we can split a problem into two subproblems, solutions of which constitute a solution to the original problem.

Using this strategy language for specifying programming exercises offers the possibility to efficiently calculate feedback while incrementally developing a program, and to significantly reduce the effort in adding new programming exercises to an intelligent tutor for programming. In practice, all programs are developed incrementally, so we think incremental development is a realistic assumption. A program that is developed incrementally contains parts that are undefined, or holes, and replacing these holes by 'more defined' programs are the steps a student takes when solving a programming problem. Replacing holes can be done by means of applying refinement rules offered by the programming tool, or by typing in the part of the desired program at that point. The exact input method is immaterial for our approach.

Using strategies for programming we can track the progress of a student solving a programming problem. We can detect deviations from the strategy, and supply hints what to do next. How we react to a deviation is not part of the strategy, but of the didactic model, which determines how strategies are used. We might not be able to recognise all steps from beginning to end, but the longer we can, the better our feedback options are. We argue that the first steps in program development are the most important steps, which require detailed and good feedback. It is at this point where programming techniques have to be selected and applied.

4 CONSTRUCTING PROGRAMMING STRATEGIES

Before we explore strategies for programming exercises, let us first have a look at an incremental construction of a solution for the programming task of implementing insertion sort in Haskell. This is an example of a small, stand-alone exercise, typical for learning how to program in the language. This exercise can also be found in popular textbooks on Haskell (Hutton, 2007).

4.1 Insertion sort

We assume that the type of the function is given as part of the exercise:

$$isort :: Ord\ a \Rightarrow [a] \rightarrow [a]$$

This type declaration expresses that lists of arbitrary types (the type variable a) can be sorted as long as an ordering is defined on the elements of the list (the type class constraint $Ord\ a$). We start with an empty definition:

$$isort = ...$$

The ellipsis in the line above indicates that the definition is not yet complete. A possible first step is to assign a name to the function's first argument of type $[a]$, which results in:

$$isort\ xs = ...$$

Now that the list to be sorted has a name (xs), we have to decide what to do with it. The important step in

```
isort :: Ord a ⇒ [a] → [a]
isort [ ]       = [ ]
isort (x : xs) = insert x (isort xs)

insert :: Ord a ⇒ a → [a] → [a]
insert a [ ]       = [a]
insert a (x : xs)
  | a ⩽ x          = a : x : xs
  | otherwise = x : insert a xs
```

Figure 1: Model solution for insertion sort

completing the definition is to realise that we have to distinguish the empty list from the list with at least one argument. This step is part of the insertion sort algorithm, and a standard technique for processing lists.

```
isort [ ]       = ...
isort (x : xs) = ...
```

By discriminating these cases, we now have two parts that have to be completed. We focus on the more challenging definition for $x : xs$ (x is the first element of the list, xs is the remaining part). Here, the insight is that xs needs to be sorted first by applying the function *isort* recursively.

```
isort [ ]       = ...
isort (x : xs) = ... isort xs ...
```

After completing the base case for the empty list, and introducing a helper-function for inserting an element into a sorted list (*insert*), we arrive at the definition given in Figure 1.

4.2 Program refinement rules

The basic steps for constructing a solution for a programming task are program refinement rules, or rewrite rules. These rules typically replace an unknown part (ellipsis) by some expression. A program refinement rule can introduce one or more new unknown parts. We are finished with an exercise as soon as all unknown parts have been completed. The insertion sort example contains several program refinement rules: assigning a name to a function's argument, distinguishing the empty list from the non-empty list by means of pattern matching, and making a recursive call to the function. These rules are the basis for programming strategies. They are reusable and not specific for the insertion sort exercise.

4.3 Strategy combinators

The simplest strategies consist of a single rewrite rule. We use a collection of standard combinators to combine strategies, resulting in more complex strategy descriptions (Heeren et al., 2008). We briefly describe the combinators most relevant for this paper. The *sequence* combinator applies its argument strategies one after another, thus allowing programs that require multiple refinement steps. The *choice* combinator makes it possible to have multiple, possibly different refinement paths. The *parallel* combinator expresses that the steps of its argument strategies have to be applied, but that the steps can be interleaved. The last combinator, *label*, marks a position in the strategy, allowing us to customise this part of the strategy later on.

4.4 Automatically deriving programming strategies

A programming strategy describes sequences of refinement steps: applying all the steps of such a sequence results in a solution for the programming task. We could specify all allowed sequences that solve a programming task by hand, but it is less labour intensive to automatically derive a programming strategy from model solutions. The advantage of using model solutions is that it becomes relatively easy for a teacher to add new programming tasks to the tutoring system, since he will be familiar with the programming language. In fact, there is no need to learn a new formalism, or to change the implementation of the system. With the strategy combinator for choice, we can combine multiple model solutions. Figure 1 contains a model solution for the insertion sort programming task.

A model solution can be compiled into a programming strategy by inspecting its abstract syntax tree (AST), where each language construct is mapped to its corresponding refinement rule. By introducing choices in the strategy for certain language constructions, we gain some flexibility in the sequences of refinement steps that we accept. For example, the two definitions for the two cases for *isort* can appear in any order since swapping the two does not change the meaning of the function. If two unknown parts are introduced by a refinement rule, we use the parallel combinator by default to leave the order in which these holes are completed unspecified. We apply this principle, for instance, for the right-hand sides of *isort* that are introduced by pattern matching.

4.5 Strategies for higher-order functions

The function *isort* presented in Figure 1 is not the standard solution that an expert would give. Figure 2

```
isort :: Ord a ⇒ [a] → [a]
isort = foldr insert []
```

Figure 2: Model solution for insertion sort with *foldr*

```
foldr :: (a → b → b) → b → [a] → b
foldr cons nil = rec where
  rec []     = nil
  rec (x:xs) = cons x (rec xs)
```

Figure 3: The higher-order function *foldr*

contains an alternative, much more concise definition for *isort* that is based on the higher-order function *foldr* (also known as a catamorphism). The definition of this function can be found in Figure 3. The function *foldr* captures compositional computations over lists, and is a highly reusable function defined in the Haskell standard Prelude library. In fact, the definition for *isort* in Figure 2 formulates at a very high level what is essential about insertion sort: we start with the empty list (*foldr*'s second argument), and the helper-function *insert* is used for each element (*foldr*'s first argument).

Remember that we can automatically derive strategies from the model solution's AST, with which we can recognise the steps of a student solving a programming task. One approach would be to combine the strategies derived for the two model solutions. Instead, we specialise the strategy that we derive for the *foldr* function such that it recognises solutions with explicit recursion (as the code Figure 1), and also solutions in terms of *foldr*. We can even let the strategy accept alternative solutions in terms of *foldr*, such as a definition that gives a name to the argument list (η-expansion):

```
isort :: Ord a ⇒ [a] → [a]
isort xs = foldr insert [] xs
```

The advantage of this approach is that the specialised strategy for *foldr* has to be defined only once, but it can be reused for several programming tasks involving lists. For instance, the task of merging a list of lists by appending all the lists, or computing all the permutations of a list, are tackled by the same strategy for *foldr*. In classroom settings, we often experience that students find it difficult to define a function using *foldr*, and prefer to use explicit pattern matching and recursion. This is not always desirable, and it could even be a goal of a programming task to use functions such as *foldr*, just to become familiar with these higher-order functions. With the specialised strategies we can easily support these kinds of tasks, or provide help in rewriting a definition with explicit recursion into an application of *foldr*.

4.6 A taxonomy of strategies

The function *isort* is a catamorphism because it can be defined as a *foldr*, but what about the helper-function *insert*? Here too we use pattern matching on lists, and recursion on the tail of the list. Carefully inspecting the definition in Figure 1 reveals that there are two cases for the non-empty list. For one, we use recursion, but in case $a \leqslant x$ we use xs without calling *insert* recursively. Technically, this means that we cannot (conveniently) use *foldr*, but that we have to define it as a paramorphism instead. The function *para* captures another class of useful computations on lists, just as *foldr*, but in a more general way:

```
para :: (a → [a] → b → b) → b → [a] → b
para cons nil = rec where
  rec []     = nil
  rec (x:xs) = cons x xs (rec xs)
```

We give an alternative definition for *insert*, which is based on *para*:

```
insert :: Ord a ⇒ a → [a] → [a]
insert a = para f [] where
  f x xs rs
    | a ⩽ x     = a:x:xs
    | otherwise = x:rs
```

The recursion pattern of *insert* is nicely captured by *para*: the definition for *insert* and its helper-function f are not recursive.

The new definition for *insert* is not shorter than the original definition, nor is it more intuitive. Still, this version is to be preferred as a model solution as it separates the recursion pattern (*para*) from the instantiation that is specific for the programming task at hand (the local function f and the empty list). In Haskell, it is possible to specify recursion patterns as higher-order functions (such as *foldr* and *para*). Programming tasks that are expressed with the same higher-order function essentially belong to the same problem class. Identifying these problem classes helps with providing detailed feedback on (partial) solutions in an interactive way.

We have introduced the functions *foldr* and *para* for our insertion sort problem, but more of these functions exist that characterise the structure of a computation. An anamorphism, for instance, helps in constructing lists from values, and has yet another recursion pattern. Augusteijn introduces various morphisms for defining other sorting algorithms (Augusteijn, 1998). The functions *map* and *filter* from

Haskell's standard library are specific instances of *foldr* but they are equally useful in classifying programming tasks. The higher-order functions give us a taxonomy of programming tasks.

The examples may give the impression that our approach only deals with computations involving lists. It is not accidental that we use lists in our examples: lists are frequently used by functional programmers, they are well supported by the language, and they are a popular subject for programming tasks. The technique we present here, namely specialising strategies for higher-order functions that capture a programming pattern, can also be applied to other data structures, such as binary trees. The same holds for other programming techniques, such as accumulating parameters, or divide and conquer algorithms.

5 USING PROGRAMMING STRATEGIES

Now that we have programming strategies available, we want to use these strategies to support a student with the stepwise construction of a program. A strategy describes the order of the refinement steps that a student has to take to construct a program. This organisation of steps enables feedback when solving a programming task.

Given a strategy, we can give various types of feedback. (Gerdes et al., 2008) give a list of feedback services derived from existing exercise assistants. This list includes different levels of feedback. In addition to feedback on the strategy level, we can also provide feedback on more basic levels. If a student makes an error on the level of syntax or types, this mistake is reported. Another basic form of feedback is to verify whether or not a refinement step is correct. The following paragraphs explain how programming strategies can be used to provide strategy related feedback.

Hints. At each point in the construction of the insertion sort function we can check whether the step taken by the student is expected, and we can give hints, in increasing specificity. For example, suppose a student asks for a hint when he is at the point of pattern matching:

$$isort\ xs = ...$$

The tutor starts with 'apply pattern matching on the argument', in this case *xs*. When a student asks for more detail we go down in the strategy, and give the two components of which the pattern matching consists, namely the empty list $[\,]$ and the non-empty list $(x:xs)$.

The steps in a strategy do not necessarily have to be sequential. As mentioned in subsection 4.3, it is also possible to do steps in parallel or to make a choice between different steps. For example, after applying the pattern matching refinement rule, the cases for the empty and non-empty list can be constructed in arbitrary order. When asking for a hint, both of these steps can be presented to the student.

Deviation from the strategy. Since a strategy outlines the steps to take, we can check if a step is in line with the strategy. If a student deviates from the strategy, there are two possibilities:

- a known refinement rule that is not part of the strategy has been applied,
- or we cannot explain the step made by the student, but cannot prove the program to be wrong.

In both cases we can either let the student go on, or report that we want the student to follow the strategy. An example of a deviation, from the *isort* strategy, is introducing three cases when pattern matching on the input list:

$$\begin{aligned} isort\ [\,] &= [\,] \\ isort\ [x] &= [x] \\ isort\ (x:xs) &= insert\ x\ (isort\ xs) \end{aligned}$$

This is a correct program that meets the requirements, but the second case is superfluous. We want to report this to the student.

Buggy strategies. Besides the correct strategy, we also specify known inappropriate ('buggy') strategies for solving the problem. Buggy strategies are used to catch common mistakes, which we use to explain what a student has done wrong. Consider the following definition:

$$\begin{aligned} isort\ [\,] &= ... \\ isort\ (x:xs) &= insert\ x\ xs \end{aligned}$$

Although this is a valid and type correct program, it does not have the expected behaviour. This is an example of a buggy strategy in which a student forgets to call the function recursively on the tail of the list $(isort\ xs)$.

Every expected deviation from the strategy can be turned in to a buggy strategy. The deviation presented before, in which a superfluous third case is given to implement *isort*, could just as well be an example of a buggy strategy. Buggy strategies make it possible to give more detailed feedback.

Customising strategies. We can calculate many types of feedback from a programming strategy specification. The implementor of an exercise assistant decides what feedback to use. For example, an exercise assistant may want to give feedback at each intermediate step or let a student complete an exercise and give feedback afterwards, by showing a complete derivation of the program.

From a didactic point of view it might be desirable to force a student to take a specific route towards the complete definition of a program. Strategies help to allow or disallow certain solution paths. Possible variations are:

- The order in which the main function (*isort*) and its helper-functions (*insert*) are developed is constrained, reflecting top-down versus bottom-up development styles.
- It is optional to enforce a student to give explicit type signatures (e.g. $isort :: Ord\ a \Rightarrow [a] \rightarrow [a]$) of the (helper-)functions he needs to define. We can ask to give the signatures in advance, at some point, or after completion.
- For functions with multiple cases (e.g., the empty list and non-empty list), it is possible to express the order in which the cases should be completed. For example, the simplest case first.

These examples give an indication of the kind of feedback that can be constructed from a programming strategy.

6 CONCLUSIONS, RELATED AND FUTURE WORK

Conclusions. We have shown how we can use strategies for programming to give students feedback while incrementally developing programs for introductory programming problems. Strategies and feedback are separated, so that users (teachers) can tune the feedback the intelligent programming tutor gives to students. Recursion combinators play a fundamental role in our approach, and offer the possibility to easily add flexibility to an intelligent tutoring system for programming, because they can capture many different forms of strategies in abstract terms.

We have developed a proof-of-concept implementation of an intelligent tutoring system for introductory programming tasks. This system supports the strategies that are described in this paper.

Related work. The Lisp tutor (Anderson et al., 1986) is an intelligent tutoring system that supports the incremental construction of Lisp programs. The interaction style of the tutor is a bit restrictive, and adding new material to the tutor is still quite some work. Using our approach based on strategies, the interaction style becomes flexible, and adding exercises becomes relatively easy.

In tutoring tools for Prolog, a number of strategies for Prolog programming have been developed (Hong, 2004). Strategies are matched against complete student solutions, and feedback is given after solving the exercise. We expect these strategies can be translated to our strategy language, and can be reused for a programming language like Haskell. (Soloway, 1985) describes programming plans for constructing Lisp programs. These plans are instances of the higher-order function *foldr* and its companions. Our work structures the strategies described by Soloway.

Automatic grading of student programs cannot be used to obtain feedback on partial programs at intermediate steps in the development of programs. But we use the work of (Xu and Chee, 2003), in particular their approach to abstract syntax tree construction from model solutions, to generate first approximations for program construction strategies. We then add development order and/or type-based strategies, several abstractions, and possibly buggy strategies to the strategies thus obtained.

Future work. Strategies constructed from model solutions might be rather strict, and enforce particular solutions. We can add more flexibility by specifying programming problems by means of *contracts* (Meyer, 1992), and then check at each intermediate step that the contract is not violated. We can then offer all possible refinement rules to the students, and give feedback at steps that violate a contract. We have yet to investigate how we can statically, incrementally, check contracts.

A model solution must be expressed in terms of higher-order functions to take advantage of the specialised strategies for these functions. Alternatively, we can try to recognise recursion patterns in model solutions. We also plan to investigate how well our approach works for other programming languages, such as Java. Although Java has no support for higher-order functions, we can use strategies to capture common, high-level programming techniques.

The proof-of-concept implementation has to be extended with contracts and further developed to be used in tests in class-rooms. Only when we have a well-developed prototype can we investigate how our work scales. The primary goal is to support introductory programming; so our approach need not scale to full-blown software engineering projects.

REFERENCES

Anderson, J. R. (1993). *Rules of the Mind*. Lawrence Erlbaum Associates.

Anderson, J. R., Conrad, F. G., and Corbett, A. T. (1986). Skill acquisition and the LISP tutor. *Cognitive Science*, 13:467–505.

Anderson, J. R. and Skwarecki, E. (1986). The automated tutoring of introductory computer programming. *Communications of the ACM*, 29(9):842–849.

Augusteijn, L. (1998). Sorting morphisms. In *3rd International Summer School on Advanced Functional Programming, volume 1608 of LNCS*, pages 1–27. Springer-Verlag.

Beeson, M. J. (1990). A computerized environment for learning algebra, trigonometry, and calculus. *Journal of Artificial Intelligence and Education*, 1:65–76.

Borovanský, P., Kirchner, C., Kirchner, H., and Ringeissen, C. (2001). Rewriting with strategies in ELAN: A functional semantics. *International Journal of Foundations of Computer Science*, 12(1):69–95.

Chaachoua et al, H. (2004). Aplusix, a learning environment for algebra, actual use and benefits. In *ICME 2004: 10th International Congress on Mathematical Education*. Retrieved from `http://www.itd.cnr.it/telma/papers.php`, May 2008.

Cohen, A., Cuypers, H., Reinaldo Barreiro, E., and Sterk, H. (2003). Interactive mathematical documents on the web. In *Algebra, Geometry and Software Systems*, pages 289–306. Springer-Verlag.

Gerdes, A., Heeren, B., Jeuring, J., and Stuurman, S. (2008). Feedback services for exercise assistants. In Remenyi, D., editor, *The Proceedings of the 7th European Conference on e-Learning*, pages 402–410. Academic Publishing Limited.

Heeren, B. and Jeuring, J. (2008). Recognizing strategies. In Middeldorp, A., editor, *WRS 2008: Reduction Strategies in Rewriting and Programming, 8th International Workshop*.

Heeren, B., Jeuring, J., Leeuwen, A. v., and Gerdes, A. (2008). Specifying strategies for exercises. In *MKM 2008: Mathematical Knowledge management*, volume 5144 of *LNAI*, pages 430–445. Springer-Verlag.

Hong, J. (2004). Guided programming and automated error analysis in an intelligent Prolog tutor. *International Journal on Human-Computer Studies.*, 61(4):505–534.

Hutton, G. (2007). *Programming in Haskell*. Cambridge University Press.

Johnson, W. L. and Soloway, E. (1985). Proust: Knowledge-based program understanding. *IEEE Transactions on Software Engineering*, 11(3):267–275.

Kölling, M., Quig, B., Patterson, A., and Rosenberg, J. (2003). The BlueJ system and its pedagogy. *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology*, 13(4).

Kumar, A. N. (2008). The effect of using problem-solving software tutors on the self-confidence of female students. In *SIGCSE 2008: Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 523–527. ACM.

López, N., Núñez, M., Rodríguez, I., and Rubio, F. (2002). WHAT: Web-based Haskell adaptive tutor. In *AIMSA 2002: Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, pages 71–80. Springer-Verlag.

Melis, E., Andrès, E., Goguadze, G., Libbrecht, P., Pollet, M., and Ullrich, C. (2001). ACTIVEMATH: a generic and adaptive web-based learning environment. *International Journal of Artificial Intelligence in Education*, 12.

Merriënboer, J. J. v., Jelsma, O., and Paas, F. G. (1992). Training for reflective expertise: A four-component instructional design model for complex cognitive skills. *Educational Technology, Research and Development*, 40(2):23–43.

Meyer, B. (1992). *Eiffel: The Language*. Prentice Hall International.

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

Odekirk-Hash, E. and Zachary, J. L. (2001). Automated feedback on programs means students need less help from teachers. In *SIGCSE 2001: Proceedings of the 32nd SIGCSE technical symposium on Computer Science Education*, pages 55–59. ACM.

Peyton Jones et al., S. (2003). *Haskell 98, Language and Libraries. The Revised Report*. Cambridge University Press. A special issue of the Journal of Functional Programming, see also `http://www.haskell.org/`.

Pillay, N. (2003). Developing intelligent programming tutors for novice programmers. *SIGCSE Bull.*, 35(2):78–82.

Proulx, V. K. (2000). Programming patterns and design patterns in the introductory computer science course. In *SIGCSE 2000: Proceedings of the 31st SIGCSE technical symposium on Computer science education*, pages 80–84. ACM.

Soloway, E. (1985). From problems to programs via plans: the content and structure of knowledge for introductory LISP programming. *Journal of Educational Computing Research*, 1(2):157–172.

Sykes, E. and Franek, F. (2004). A prototype for an intelligent tutoring system for students learning to program in Java. *Advanced Technology for Learning*, 1(1).

Visser, E., Benaissa, Z.-e.-A., and Tolmach, A. (1998). Building program optimizers with rewriting strategies. In *ICFP 1998: International Conference on Functional Programming*, pages 13–26.

Xu, S. and Chee, Y. S. (2003). Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Transansactions on Software Engineering*, 29(4):360–384.