ASK-ELLE: a Haskell Tutor

Proefschrift

ter verkrijging van de graad van doctor
aan de Open Universiteit
op gezag van de rector magnificus
prof. mr. A. Oskamp
ten overstaan van een door het
College voor promoties ingestelde commissie
in het openbaar te verdedigen

op vrijdag 23 november 2012 te Heerlen
om 13:30 uur precies

door

Alex Gerdes

geboren op 10 juli 1978 te Emmen

Promotor

Prof. dr. J.Th. Jeuring	Open Universiteit Universiteit Utrecht

Overige leden beoordelingscommissie

Prof. dr. dr.h.c. ir. M.J. Plasmeijer	Radboud Universiteit
Prof. dr. S.D. Swierstra	Universiteit Utrecht
Prof. dr. S.J. Thompson	University of Kent
Prof. dr. M.C.J.D. van Eekelen	Open Universiteit Radboud Universiteit
Prof. dr. ir. S.M.M. Joosten	Open Universiteit

Printed by Gildeprint drukkerijen, Enschede.
Cover by Peter Gerdes.

ISBN 978-94-6108-371-5

CONTENTS

Acknowledgements

The last five years I participated as a 'promovendus' in a ongoing research project on generating semantically rich feedback. I thoroughly enjoyed this period, and never regretted the decision of taking this academic job. It involved hard work, a lot of travelling, many interesting discussions, and much coding. Together this culminated in this thesis. During my research I received help from many people, for which I am very grateful.

First of all I would like to thank my promotor Johan Jeuring, whom I owe very much. You have always been very thorough and precise when reviewing my work, and guided me into the right direction when needed. You always had time for me, even when you moved to Sweden for a while. You constantly pushed me to do better. That was not always easy, but I value it very much. Johan, I am very lucky to have had you as my supervisor; I could not have wished for a better one!

I would also like to thank the members of the examination committee (Marko van Eekelen, Stef Joosten, Rinus Plasmeijer, Doaitse Swierstra, and Simon Thompson) for reading and approving the manuscript, and providing me with several useful comments.

Also, I would like to thank Bastiaan Heeren for the many discussions we had about our research and the implementation of our software. I learned a lot from those discussions; you are a gifted programmer. Furthermore, I would like to thank Lex Bijlsma for being the perfect line-manager, your response time is legendary. A big thank you goes to Chrisja Muris, Harrie Passier, Sylvia Stuurman, Josje Lodder, and the other colleagues at the Open University. You made my time working in Heerlen very enjoyable.

I am thankful for Doaitse Swierstra and the Utrecht University for having me as guest during my PhD-study. The Software Technology group is a very inspiring environment and an excellent place to do research. I would like to thank José Pedro Magalhães, Sean Leather, Andres Löh, Wouter Swierstra, Stefan Holdermans, Arie Middelkoop, Jurriaan Hage, Alexey Rodriguez Yakushev, and all the other colleagues for the really nice time I had in Utrecht. I would also like to thank my roommate (and 'lotgenoot') Thomas van Noort. Another thanks goes to Bram Schuur and Bram Vaessen for assisting me with the experiments.

I had the pleasure of meeting Mary Sheeran at the Central European Functional Programming summer school 2011 in Budapest. This meeting eventually led to a dream job at QuviQ in Göteborg. Mary I can't thank you enough for that! I would also like to thank Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson (my new colleagues at QuviQ) for the warm welcome. You have made the move to Sweden a lot less difficult. Tack så jätte mycket!

Finally, I would like to thank my parents Rudie and Willy, my sister Marloes, my brother Peter (who was kind enough to design the beautiful cover of this thesis),

my parents in law Hillie and Harm, and the rest of my family and friends for their support, understanding, and time. John, Jos, and Gosé, I am grateful that you are my friends. Especially I would like to thank my wife Anita and my children Mats, Lise, and Sofie – you make it all worthwhile. Anita, thank you for who you are; you'll never know the extent of my gratitude!

Alex Gerdes Herrljunga, Autumn 2012

1 Introduction

Learning to program is challenging. A first course in programming is often a major stumbling block (Proulx, 2000), and the results of such a course are often disappointing (McCracken et al., 2001). There is no final answer (yet) to the question how programming is learned best, and what makes programming hard (Fincher and Petre, 2004). The topic of how students learn to program has been studied extensively, by computer scientists, educational scientists, and cognitive psychologists. 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.

To support learning programming, many intelligent programming tutors have been developed. There exist tutors for Prolog (Hong, 2004), Lisp (Anderson et al., 1986), Pascal (Johnson and Soloway, 1985; Soloway et al., 1981), Java (Holland et al., 2009; Kölling et al., 2003; Sykes and Franek, 2004), Haskell (López et al., 2002; Xu and Sarrafzadeh, 2004), and many more programming languages. Some of these tutors are well-developed and extensively tested in classrooms, but most haven't outgrown the research prototype phase, and are not maintained anymore. Studies show the positive effects of various tutors on learning programming. These 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 (Corbett et al., 1988),
- using intelligent tutors requires 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; Hattie and Timperley, 2007).

Learning through feedback is essential, also for learning programming. In Rules of the Mind (Anderson, 1993), Anderson discusses the ACT-R principles of tutoring, and the effectiveness of feedback in intelligent tutoring systems (ITSs). One of the tutoring principles deals with student errors. If a student made a slip in performing a step she should be allowed to correct it without further assistance. However, if a student needs to learn the correct rule, the system should give a series of hints with increasing detail, or show how to apply the correct rule. Finally, it should also be possible to give an explanation of an error made by the student. Anderson observed no positive effects in learning with deferred feedback, but observed a decline in learning rate instead. Erev et al. (2006) also claim that immediate feedback is often to be preferred.

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, it is often hard for a teacher to deploy an intelligent tutor in a course (Anderson et al., 1995). It is usually quite difficult for teachers to adapt or add programming exercises to an intelligent programming tutor, and to adapt the

feedback given by the tutor. 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 doesn't follow the desired path. All this knowledge then has to be translated into the internals of a tutor, which implies a lot of work. For example, completely specifying feedback in (much simpler) mathematical exercises results in exercise files of hundreds of lines (Cohen et al., 2003). Another important aspect for the acceptance of a programming tutor is that it should offer sufficient freedom to students: a student should be able to use her own names, to use her own favourite programming style, her own refinement step-size, etc.

The functionality and help offered by programming tutors varies. The following aspects play a role:

- *Adaptability*: can a teacher add her own exercises to a tutor, and can she adapt the behaviour so that particular ways for solving an exercise are enforced or disallowed?

  Anderson et al. (1995) mention the lack of adaptability as one of the main reasons for the slow uptake of their tutors outside their own teaching environment. Bokhove and Drijvers (Bokhove and Drijvers, 2010) list teacher adaptability as one of the four fundamental requirements for mathematical learning environments, and Lowes (Lowes, 2007) found that among a group of almost a hundred teachers of online courses, almost 70% regularly adapt their assignments.

- *Development process*: does the tutor support the incremental development of programs, where a student can obtain feedback or hints on incomplete programs, can a student follow his or her preferred way to solve a particular programming problem, does the tutor support refactoring a program, can a student submit a complete solution to a problem in the tutor?

  The incremental approach to developing programs supports *knowledge compilation* (Anderson et al., 1995), and is fundamental for learning according to the ACT-R theory.

- *Correctness*: does the tutor guarantee that a student solution is correct, can it check that a student has followed good programming practices, can it verify that the student solution has the desired efficiency, does it give an explanation why a program is incorrect, does it give counterexamples for incorrect programs, and/or does it detect at which point of a program a particular property is violated?

  The possibility to detect property violations and to generate counterexamples at intermediate steps meets the desire for tutoring flexibility expressed by both

students and teachers (Gerdes et al., 2012b). Furthermore, using properties it is easier for a teacher to specify a problem for a tutor, and she can specify more substantial programming problems.

Functional programming. In this thesis we focus on teaching and learning *functional programming*. Functional programming languages are fundamentally different from languages from the more widespread imperative programming paradigm, such as C or Java. Instead of imperative statements, programmers define functions that are applied to arguments. There are no assignment statements or mutable state. A program is a (main) function that is defined in terms of other functions. A functional programming language allows a programmer to define what a program should accomplish, rather than describing how to accomplish it.

A popular functional programming language is Haskell (Peyton Jones, 2003). It is a lazy, pure, and statically typed programming language with a relatively small core based on the λ-calculus. Laziness implies that Haskell defers the evaluation of expressions until their results are needed. Arguments supplied to a function are only evaluated if their values are used. Another prominent feature of Haskell is that it is strongly and statically typed. Haskell's type system does not allow implicit type conversions, and type errors are detected at compile time.

Functional programming, and in particular Haskell, is taught at many universities[1] around the world. Haskell is a good language to teach many important concepts in computer science, because it has advanced means to define abstractions, types, polymorphism, recursion, etc. There exist many good text books on Haskell (Bird, 1998; Hudak, 2000; Hutton, 2007; O'Sullivan et al., 2008; Thompson, 1999), which introduce and explain these concepts.

Research questions. In this thesis we investigate the following research questions:

How can we design and implement a functional programming tutor

- *that automatically gives semantically rich feedback to students incrementally solving an exercise,*
- *to which teachers can easily add exercises,*
- *in which teachers can easily fine-tune feedback?*

We have built a functional programming tutor in which we address these research questions. We have developed ASK-ELLE , a tutor for Haskell that supports the stepwise development of simple functional programs. The tutor targets beginning computer science students. This thesis gives a complete overview of ASK-ELLE.

[1] `http://www.haskell.org/haskellwiki/Haskell_in_education`

We explain all components of ASK-ELLE, along with examples and implementation details. The next section gives a short introduction to ASK-ELLE. After that, Section 1.2 discusses other relevant programming tutors and relates our work to these tutors. In Section 1.3 we present the structure of this thesis, give an overview of the contents of all chapters, and describe the relations between the chapters. In the last section of this chapter, Section 1.4, we describe the origin of the chapters.

1.1 ASK-ELLE

VanLehn (2006) proposes a standardisation of the terminology to describe ITSs. He distinguishes two loops in an ITS: an outer and an inner loop. The main task of the outer loop is to select an appropriate task for the student. The inner loop is responsible for giving hints and feedback on student steps.

ASK-ELLE is an interactive system that supports students while solving introductory functional programming exercises. The tutor is offered as a web application. See Figure 1.1 for a screenshot. ASK-ELLE has a relatively simple outer loop, which allows the student to choose an exercise from a subset of the exercises found on the Haskell-99 list[2]. The inner loop of ASK-ELLE is our main contribution. Using the system, students learning functional programming

- develop their programs incrementally,
- receive feedback about whether or not they are on the right track,
- can ask for a hint when they are stuck,
- see how a complete program is stepwise constructed.

All of this functionality is calculated automatically from annotated solutions specified by a teacher.

ASK-ELLE itself is implemented as a functional program, and uses fundamental concepts from software technology such as rewriting, parsing, strategies, program transformations and higher-order combinators such as the *fold*. The tutor is built on top of our general software framework for specifying domain reasoners (Heeren et al., 2010), and uses the Helium compiler for Haskell (Heeren et al., 2003). Helium gives excellent syntax-error and type-error messages, and reports dependency analysis problems in a clear way.

Model solutions. For any programming problem there are many solutions. Some of these solutions are syntactical variants of each other, but other solutions implement different ideas to solve a problem. A teacher can specify her exercises in

[2] `http://www.haskell.org/haskellwiki/99_Haskell_exercises`

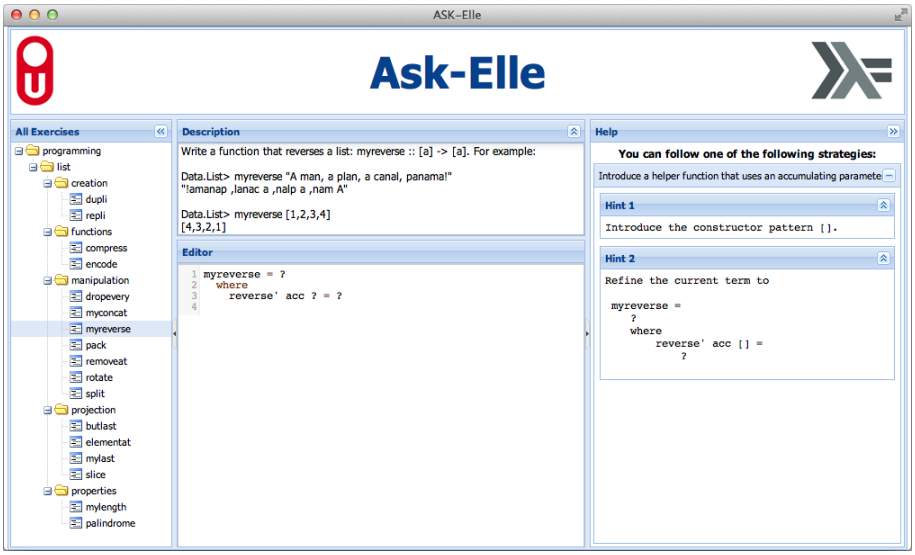


Figure 1.1: ASK-ELLE, a web-based functional programming tutor

ASK-ELLE by giving a set of *model solutions* for a problem. A model solution is a program that an expert writes, using good programming practices.

Our tutor supports the incremental construction, in a top-down fashion, of model solutions. It recognises incomplete versions of these solutions, together with all kinds of syntactical variants. We support the refinement of programs. Instead of showing that a program ensures a post-condition by testing the program or proving the program to be correct, we assume a program to be correct if we can determine it to be equal to a model solution. The tutor aims to be as flexible as possible for teachers as well as for students. For example, a student may use her own names for functions and variables, and may use different, but equivalent, language constructs.

The tutor generates feedback based on a set of model solutions for a particular programming problem. A teacher can adapt feedback by annotating the model solutions. This requires translating annotated model solutions to a form which we can use to track intermediate student steps. We use *strategies* to track the intermediate steps taken by a student.

Programming strategies. A procedure to solve an exercise often consists of multiple steps. For example, developing an explicit recursive function on lists often consists of introducing a case distinction between the empty list and the non-empty list, and a recursive call in the non-empty list case, amongst others. A procedure

may also contain a choice between different (sequences of) steps, such as either using a higher-order function, or an explicit recursive definition. Sometimes, the order in which the steps are performed is not relevant, as long as they are performed at some point.

We have developed a strategy language for describing procedures as rewrite strategies (Heeren et al., 2008). Our strategy language is domain independent, and has been used to describe strategies for exercises in mathematics, logic, and biology, in addition to programming. A strategy for a functional program describes how a student should construct a functional program for a particular problem. The basic elements of the strategy language are *rewrite or refinement rules* . We use these strategies to support students using our intelligent programming tutor to incrementally develop a program. We can automatically *derive* a strategy from a set of model solutions.

Refinement and rewrite rules. A student develops a program by making small, incremental, changes to a program. Alternatives in teaching programming are to give a student an incomplete program, and ask her to complete the program, or to give a student a program, and ask her to change the program at a particular point. In such assignments, a student refines or rewrites a program. After each refinement, a student can ask the tutor whether or not the refinement is bringing her closer to a correct solution, or, if the student doesn't know how to proceed, ask the tutor for a hint. Rewriting preserves the semantics of a program; refining makes a program more defined.

Normalisation. To verify that a program submitted by a student follows a strategy, we apply all refinement rules allowed by the strategy to the previous submission of the student, normalise the programs thus obtained, and compare each of these programs against the normalised submitted student program. Normalisation returns a canonical form of a program. Using normalisation, we want to recognise as many syntactical variants of Haskell programs as possible. For example, sometimes a student doesn't explicitly specify all arguments to a function, and for that purpose we use η-reduction when analysing a student program. Normalisation uses various *program transformations* to reach a canonical form of a Haskell program. We are not interested in the actual canonical form; we are striving to transform as many equivalent programs as possible to the same form.

1.2 Related work on programming tutors

If ever the computer science education research field (Fincher and Petre, 2004) finds an answer to the question of what makes programming hard, and how

programming environments can support learning how to program, it is likely to depend on the age, interests, major subject, motivation, and background knowledge of a student. Programming environments for novices come in many variants, and for many programming languages or paradigms (Guzdial, 2004; Kelleher and Pausch, 2005; Pears et al., 2007). Programming environments like Scratch and Alice (Utting et al., 2010) target younger students than we do, and emphasise the importance of constructing software with a strong visual component, with which students can develop software to which they can relate.

We have not found other tutors that support the stepwise development of programs, automatically calculating feedback based on teacher-specified annotated solutions. We compare our approach to a number of examples which we believe are relevant to the work presented in this thesis, knowing that we leave out several notable others.

1.2.1 Lisp tutoring

Our tutor resembles the Lisp tutor (Anderson et al., 1986) in that it supports the stepwise development of programs, and gives hints at intermediate steps. The Lisp tutor supports the incremental development of programs and can explain common errors. Adding new material to the tutor is still quite some work, and a teacher cannot easily adapt the feedback. Using our approach based on strategies, the interaction style becomes flexible, and adding exercises becomes relatively easy. By generating strategies from model solutions we think it is easier to add programming exercises to our tutor. Moreover, teachers can easily fine-tune the generated feedback.

Soloway (1985) describes programming plans for constructing Lisp programs. These plans are instances of the higher-order function *foldr* and its companions.

1.2.2 Prolog tutoring

The Prolog tutor (Hong, 2004) characterises and classifies programs that use the same programming technique and share a common pattern of code. It defines a set of grammar rules that capture a programming technique for each class of programs. These grammar rules can be used for programming technique recognition, program construction, and program parsing. The system conducts tutoring in two different modes: guided programming and automated error analysis.

The programming techniques developed by the Prolog tutor are similar to our strategies. These 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.

1.2.3 Scheme tutoring

DrScheme (Findler et al., 2002), continued as DrRacket, is an interactive programming environment targeting introductory level students. It offers a syntax checker with lexical scope analysis. Students can evaluate an expression in a read-eval-print loop, to gain insight in the semantics of the language. In addition to evaluating expressions, students can also step through an evaluation. The environment comes with a static debugger, which basically is a soft type inferencer. To teach students the syntax and semantics of Scheme, the programming environment supports four increasing levels of exposure to the programming language.

In contrast to our approach, the environment does not explicitly offer programming exercises. The programming environment does not provide feedback or hints with respect to the solution of an exercise.

1.2.4 Java tutoring

J-Latte (Holland et al., 2009) is an intelligent tutoring system for Java programming language. It offers two modes: a concept mode, in which a student designs a program on a high-level without any statements, and a code mode in which a student completes the code generated based on the result from the concept mode. J-Latte verifies complete student Java programs against constraints.

Another Java programming tutor is JITS (Sykes and Franek, 2004). It only supports a subset of the Java programming language. Similar to our approach, JITS uses expert model solutions to generate feedback. In addition to these model solutions, a number of incorrect responses can be specified for an exercise. JITS uses techniques from the artificial intelligence field to make a decision what feedback the tutor should give to the student.

Both of the Java tutors require students to submit full programs (which may be very small), which are then scrutinised to determine the students intent (JITS) or whether or not properties are violated (J-Latte). Although small steps may be supported in these tutors, students cannot submit incomplete programs and get feedback or hints on how to proceed.

1.2.5 Pascal tutoring

PROUST (Johnson and Soloway, 1985) is a Pascal tutor. PROUST tries to reconstruct the, possibly erroneous, steps that a student took in the construction of a program. The system attempts to relate a program to the intentions of the student, bridging the gap between what is meant and what is actually realised. To do so, the system matches programming plans to complete a student program. These plans are related to our programming strategies in the sense that they capture expert

knowledge about how to solve a particular problem. In contrast to PROUST, we use strategies to generate semantically rich feedback that we give to students during the construction of a program.

1.2.6 Haskell tutoring

The Haskell tutors WHAT (López et al., 2002) and Haskell-Tutor (Xu and Sarrafzadeh, 2004) focus on building a student model, which is used to determine what kind of exercises should be offered to the student. The feedback these two tutors offer is limited: López et al. (2002) use unit testing to detect errors in student programs, and Xu and Sarrafzadeh (2004) displays errors generated by the compiler (Helium). Both tutors do not support the stepwise development of a program.

1.2.7 Mathematics tutoring

Intelligent tutors for mathematics such as MATHPERT (Beeson, 1990), APLUSIX (Chaachoua et al., 2004), ACTIVEMATH (Melis et al., 2001), 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,
- checking correctness of intermediate steps is much easier because many mathematical problems are solved by applying meaning-preserving transformations 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.

1.3 Structure of this thesis

This thesis is divided into two parts. The first part, chapters 2 to 5, describes our general software framework for specifying *domain reasoners*, on which our functional programming tutor is built. We have used this framework to specify domain reasoners for various mathematical domains (Heeren et al., 2008; Heeren and Jeuring, 2008, 2009, 2010, 2011). Chapter 2 introduces the most prominent

component of our domain reasoners: strategies. We have developed a language in which we can specify strategies. We introduce rewrite rules, which are the symbols of the strategy language, and give the semantics of the language. The following chapter in this part, Chapter 3, highlights some implementation details of a recogniser for our strategy language. In Chapter 4 we explain exercises in the context of our framework. We describe the different components of an exercise, such as the set of rewrite rules. We give a formal model of an exercise, and specify relevant properties. In Chapter 5 we introduce the concept of a domain reasoner, and explain the various components of it. The communication with a domain reasoner goes via (web)services. We formalise the available services, and describe them in detail. To make the explanation of a domain reasoner more concrete, we show an example domain reasoner for the domain of arithmetic with fractions.

The second part of this thesis, chapters 6 to 11, discusses our Haskell tutor. Chapter 6 introduces our programming tutor, and shows an example of an interactive session with a hypothetical student, and reports on the results and conclusions of a series of experiments conducted. Teachers have some control over the feedback generated by the programming tutor. Chapter 7 shows how teachers can specify programming exercises and fine-tune the generated feedback by annotating model solutions. The following chapters detail different aspects of the programming tutor. Chapter 8 shows how we use our strategy language for the functional programming domain, and how we derive a programming strategy from a set of model solutions. In Chapter 9 we explain our normalisation procedure. Before the student submission is examined, we normalise the submission to ignore insignificant differences. In addition to the flexibility offered to teachers, the tutor also aims to be flexible towards students. Chapter 10 highlights an aspect of this flexibility towards students: we discuss how we recognise an arbitrary number of steps by a student in one go. The last chapter, Chapter 11, is about an alternative usage of the techniques behind our programming tutor. In this chapter we explain how we use the same technology to automatically assess Haskell programming exercises.

1.4 Origin of chapters

This thesis is largely based on a number of reviewed and published articles. Many parts of the articles have been revised and rewritten. The chapters of this thesis are based on the following publications:

Chapter 2: Gerdes et al. (2010a); Jeuring et al. (2012); Heeren et al. (2010)

Chapter 3: Heeren and Jeuring (2008). *This chapter is a revised version of the paper written by Heeren and Jeuring. This text is not written but*

only revised by the candidate[3]*. It is included to complete the description of domain reasoners (Part I).*

Chapter 4: Gerdes et al. (2010a); Heeren and Jeuring (2009)

Chapter 5: Gerdes et al. (2008, 2010a); Heeren and Jeuring (2010)

Chapter 6: Gerdes et al. (2012b); Jeuring et al. (2012)

Chapter 7: Gerdes et al. (2012b,a)

Chapter 8: Gerdes et al. (2009); Jeuring et al. (2012)

Chapter 9 : Gerdes et al. (2010b); Jeuring et al. (2012)

Chapter 10: Gerdes et al. (2012a)

Chapter 11: Gerdes et al. (2010b)

The contribution of the candidate to the above articles is the following:

Gerdes et al. (2008, 2009, 2010a,b, 2012a,b) The candidate is the main author of these articles, and responsible for most of the presented work.

Heeren and Jeuring (2009) The text on views, in Section 4.1.5, originates from this article. It is not written but only revised by the candidate. We include this text because views are an important part of an exercises specification.

Heeren et al. (2010) The candidate merged and extended previous work into this journal article.

Heeren and Jeuring (2010) The text on feedback scripts, in Section 5.3, originates from this article. It is not written but only revised by the thesis author. We include this text to complete the description of domain reasoners.

Jeuring et al. (2012) The candidate is the author of chapters 4 and 5, and some parts of chapter 6 of these lecture notes. The candidate is also responsible for implementing the Haskell tutor, as well as adapting the general framework where necessary.

[3] That is the author of this thesis.

Part I

Domain reasoners

2 STRATEGIES

Tools like APLUSIX (Chaachoua et al., 2004), ACTIVEMATH (Melis et al., 2001), MATHPERT (Beeson, 1998), the Freudenthal digital math environment (DME) (Freudenthal Institute, 2004; Boon and Drijvers, 2005), our tool for rewriting logic expressions (Lodder et al., 2006), and our programming tutor ASK-ELLE (Gerdes et al., 2012b) support solving exercises incrementally. Ideally such a tool gives detailed feedback on several levels. For example, when a student rewrites $\frac{1}{2} + \frac{2}{3}$ into $\frac{}{6} + \frac{2}{3}$, the tool should tell the student that there is a missing numerator. If the same expression is rewritten into $\frac{3}{5}$, it should tell the student that a common error has been made when adding the two fractions: the fractions should have the same denominator before adding them. Finally, if the student rewrites $\frac{1}{2} + \frac{2}{3}$ into $\frac{2}{4} + \frac{2}{3}$, it should tell the student that although this step is not wrong, it is not the step expected by the strategy. The strategy expects the fractions to have a common denominator, instead of a common numerator.

The first kind of error is a syntax error, and there exist good error-repairing parsers that suggest corrections to formulas with syntax errors (Swierstra and Duponcheel, 1996). The second kind of error is a rewriting error: the student rewrites an expression using a non-existing or buggy rule (Brown and Burton, 1978; Hennecke, 1999). There already exist some interesting techniques for finding the most likely error when a student incorrectly rewrites an expression. The third kind of error is an error on the level of the procedural skill or strategy for solving this kind of exercises. This chapter discusses how we can formulate and use strategies

to construct the third kind of feedback.

Strategies specify how a wide range of exercises can be solved incrementally, such as bringing a logic proposition to disjunctive normal form, calculating with fractions, or defining a functional program. A strategy captures expert knowledge about how to solve a particular problem. It describes which steps a student can take to solve an exercise, and in what order. When a student solves an exercise stepwise, we can check whether or not a step follows the strategy. In this chapter we introduce a language for specifying strategies for solving exercises. This language makes it easier to automatically calculate feedback, for example when a user makes an erroneous step in a calculation. We use strategies to derive *semantically rich feedback* for exercises (Heeren et al., 2010). We can automatically generate worked-out examples, give hints, track the progress of a student by inspecting submitted intermediate answers, and give suggestions in case the student deviates from the strategy.

The strategy language is implemented as an embedded domain-specific language (EDSL) (Hudak, 1996). A strategy describes valid sequences of rewrite rules, which turns tracking intermediate steps into a parsing problem. Computer Science has almost 50 years of experience in parsing sentences of context-free languages, including error-repairing parsers. This view on interactive exercises allows us to take advantage of the experience in parsing sentences of context-free languages, and transfer this knowledge and technology to the domain of stepwise solving exercises. We can use this knowledge to give feedback on the level of strategies. In this chapter we work out the similarities between parsing and solving exercises incrementally, and we discuss generating feedback using strategies. The strategy language can be used for many domains, 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. The specification of a strategy and the calculation of feedback is separated: we can use the same strategy specification to calculate different kinds of feedback.

This chapter is organised as follows. Section 2.1 introduces strategies, and discusses how they can help to improve feedback in intelligent tutoring systems. We continue with some example strategies from the domain of arithmetic (Section 2.2). Then we present our language for specifying strategies in Section 2.3. We do so by defining a number of strategy combinators, by showing how the various example strategies can be specified in our language, and by defining the semantics of these strategy combinators. Section 2.4 defines a number of strategy functions, which we use to extend the semantics. We explain how we use strategies to rewrite expressions. In Section 2.5 we discuss several kinds of feedback and hints that can be generated using our strategy language. Finally, we discuss related work in Section 2.6.

2.1 Strategies and feedback

Whatever aspect of intelligence you attempt to model in a computer program, the same needs arise over and over again (Bundy, 1983):

- 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.

In the case of exercises, the knowledge about how to guide reasoning is often captured by a so-called procedure or procedural skill. A procedure describes how basic steps may be combined to solve a particular problem. A procedure is often called a *strategy* (or meta-level reasoning, meta-level inference (Bundy, 1983), procedural nets (Brown and Burton, 1978), plans, tactics, etc.), and we will use this term in this thesis.

Many subjects require a student to learn strategies. At elementary school, students have to learn how to calculate a value of an expression, which may include fractions. At high school, students learn how to solve a system of linear equations, and at university, students learn how to apply Gaussian elimination to a matrix, or how to rewrite a logical expression to disjunctive normal form. Strategies are not only important for mathematics, logic, and computer science, but also for physics, biology (Mendel's laws), and many other subjects. Strategies are taught at any level, in almost any subject, and range from simple – for example the simplification of arithmetic expressions – to very complex – for example a complicated linear algebra procedure.

Intelligent tutoring systems for learning strategies. Strategic skills are almost always acquired by practising exercises, and indeed, students usually equate mathematics with solving exercises. In schools, the dominant practice still is a student performing a calculation using pen-and-paper, and the teacher correcting the calculation (the same day, in a couple of days, after a couple of weeks). There exist many software solutions that support practising exercises on a computer. The simplest kinds of tools offer multiple-choice questions, possibly with an explanation of the error if a wrong choice is submitted. A second class of tools asks for an answer to a question, again, possibly with an analysis of the answer to give feedback when an error has been made. The class of tools we consider are tools that support the incremental, stepwise calculation of a solution to an exercise, thus mimicking the pen-and-paper approach more or less faithfully. Since intelligent tutoring systems (e-learning systems, interactive learning environments, etc.) for practising procedural skills seem to offer many advantages, hundreds of tools that support practising

strategies in mathematics, logic (see (Ditmarsch, 2009) for a list of almost 50 tools for teaching logic), physics, etc. have been developed.

Feedback in intelligent tutoring systems. Of the intelligent tutoring systems that support incrementally solving exercises there are only very few that mimic the incremental pen-and-paper approach and that give detailed feedback at intermediate steps based on student input. The DME (Freudenthal Institute, 2004) labels intermediate steps with a green (correct) or red (wrong) symbol. MATHDOX (Cohen et al., 2003) can provide more detailed feedback, which has to be specified together with the exercise, leading to exercise files of hundreds of lines. ACTIVEMATH (Goguadze et al., 2005) gives feedback for classes of exercises, but doesn't take student input into account when providing feedback.

Although all these kinds of feedback at intermediate steps are valuable, it is unfortunate that the full possibilities of e-learning tools are not used. There are several reasons why the given feedback is limited. The main reasons probably are that supporting detailed feedback for each exercise is very laborious, providing a comprehensive set of possible bugs for a particular domain requires a lot of research (see for example Hennecke's work (Hennecke, 1999) on student bugs in calculating fractions), and automatically calculating feedback for a given exercise, strategy, and student input is very difficult.

Representing strategies. Representing a domain and the rules for manipulating an expression in the domain is often relatively straightforward. Specifying a strategy for an exercise is more challenging in many cases. To specify a strategy, we need the power of a full programming language: many strategies require computations of values. However, to calculate feedback based on a strategy, we need to know more than that it is a program. We need to know its structure and basic components, which we can use to report errors.

An embedded domain-specific language for specifying strategies. This chapter discusses the design of a language for specifying strategies for exercises. The domains and rules vary for the different subjects, but the basic constructs for describing strategies are the same for different subjects ('first do this, then do that', 'either do this or that'). So, the strategy language can be used for any domain (mathematics, logic, physics, functional programming, etc.). It consists of several basic constructs from which strategies can be built. These basic constructs are combined with program code in a programming language to be able to specify any strategy. The strategy language is formulated as an EDSL in a programming language to easily facilitate the combination of program code with a strategy. Here 'domain-specific' means specific for the domain of strategies, not specific for the

domain of exercises. The separation into basic strategy constructs and program code offers us the possibility to analyse the basic constructs, from which we can derive several kinds of feedback.

What kind of feedback? We can automatically calculate the following kinds of feedback, many of which are part of the tutoring principles of Anderson.

- Is the student still on the right path towards a solution? Does the step made by the student follow the strategy for the exercise? What is the next step the student should take?
- We produce hints based on the strategy.
- We can give an indication of the progress, based on the position on the path from the starting point to the solution of an exercise.
- If a student enters a wrong final answer, we can ask the student to solve subproblems of the original problem.

We do not build a model of the student to try to explain the error made by the student. According to Anderson, an informative error message is better than bug diagnosis. However, we do intend to include facilities for building a student model in the future, to offer the possibility to select tasks that are suitable for a student.

How do we calculate feedback using strategies? The strategy language is defined as an EDSL in Haskell (Peyton Jones, 2003). Using the basic constructs from the strategy language, we can create something that resembles a context-free grammar (CFG). The sentences of this grammar are sequences of rewrite steps (applications of rules). We can thus check whether or not a student follows a strategy by parsing the sequence of rewrite steps, and checking that the sequence of rewrite steps is a prefix of a sentence from the context-free grammar. We use top-down recursive parsing to track student behaviour and give feedback, because we want to support the top-down, incremental construction of derivations.

Many steps require student input, for example when a student wants to multiply the numerator and denominator of a fraction by a number. This part of the transformation cannot be checked by means of a CFG, and here we make use of the fact that our language is embedded into a full programming language, to check input values supplied by the student. The separation of the strategy into a context-free part, using the basic strategy combinators, and a non-context-free part, using the power of the programming language, offers us the possibility to give the kinds of feedback mentioned above.

2.2 Example strategies

In this section we present two strategies for rewriting simple arithmetic expressions. The first strategy expresses how to simplify arithmetic expressions with powers, and the second strategy describes a procedure for calculating a simpler form of a fraction. Although the example strategies are relatively simple, they are sufficiently rich to demonstrate the main components of our strategy language.

The domain. Before we can define a strategy, we first have to introduce the domain of arithmetic expressions and a collection of available rules. An arithmetic expression is a variable, a constant (a natural number), or one of the following operations: the addition of two expressions, the subtraction of two expressions, the division of two expressions (a fraction), the multiplication of two expressions, the negation of an expression, or the power of an expression to another expression. This results in the following grammar:

$$\begin{array}{lcl} \mathit{Expr} & ::= & \mathit{Var} \mid \mathbb{N} \mid \mathit{Ops} \\ \mathit{Var} & ::= & a \mid b \mid \ldots \mid x \mid y \mid \ldots \\ \mathit{Ops} & ::= & \mathit{Expr} + \mathit{Expr} \mid \mathit{Expr} - \mathit{Expr} \mid \dfrac{\mathit{Expr}}{\mathit{Expr}} \mid \mathit{Expr} \cdot \mathit{Expr} \\ & \mid & \mathit{Expr}^{\mathit{Expr}} \mid -\mathit{Expr} \end{array}$$

If necessary, we write parentheses to resolve ambiguities. Examples of valid expressions are $\frac{1}{3} \cdot (\frac{2}{3} + \frac{3}{4})$ and $a^x \cdot a^y$.

The rules. Figure 2.1 presents, amongst others, a small collection of basic rules. All variables in these rules are meta-variables and range over arbitrary arithmetic expressions. The rules are expressed as equivalences, but are only applied from left to right. For some rules we assume we have a commutative variant, for instance, $0 + a = a$ for rule UNITADD. Using these rules, we can simplify arithmetic expressions.

Every real-life learning environment for this domain has to be aware of a much richer set of rules. In particular, we have not given rules for commutativity and associativity of some of the operations, and many rules for other operations are omitted, such as taking the logarithm.

Strategy 1: simplifying expressions with powers. The first strategy applies the basic rules for powers from Figure 2.1 exhaustively in a bottom-up manner. We proceed with applying rules as long as a rule can be applied *somewhere*. If no rule can be applied anymore, we have a simplified arithmetic expression. The strategy is

Units and zeroes:	UNITADD: $a + 0 = a$		UNITSUB: $a - 0 = a$	
	UNITMUL: $a \cdot 1 = a$		UNITPOW: $a^1 = a$	
	ZEROMUL: $a \cdot 0 = 0$		ZEROPOW: $a^0 = 1$	
Fraction rules:	ADD: $\frac{a}{c} + \frac{b}{c} = \frac{a+b}{c}$		RENAME: $\frac{b}{c} = \frac{a \cdot b}{a \cdot c}\ (a \neq 0)$	
	MUL: $\frac{a}{b} \cdot \frac{c}{d} = \frac{a \cdot c}{b \cdot d}$		SIMPL: $\frac{a+b}{b} = 1 + \frac{a}{b}$	
Power rules:	ADDPOW: $a^x \cdot a^y = a^{x+y}$		DISTPOW: $(a \cdot b)^x = a^x \cdot b^x$	
	MULPOW: $(a^x)^y = a^{x \cdot y}$			

Figure 2.1: Basic rules for simple arithmetic expressions

very liberal, and approves many sequence of rules. Using this strategy to simplify $(a^3 \cdot a^4)^2$ results in the following derivation:

$$(a^3 \cdot a^4)^2 \overset{\text{ADDPOW}}{\Longrightarrow} (a^7)^2 \overset{\text{MULPOW}}{\Longrightarrow} a^{14}$$

In this derivation the addition of integers (i.e., $3 + 4$) is performed silently.

Strategy 2: adding fractions. The second strategy captures the procedure for adding fractions in arithmetic expressions. If the result after adding two fractions is an improper fraction (the numerator is larger than or equal to the denominator), then it should be converted to a mixed number. Figure 2.1 displays a number of basic rewrite rules on fractions. A possible strategy to solve this type of exercise is the following:

Step 1. Find the least common denominator (LCD) of the fractions: let this be n

Step 2. rename the fractions such that n is the denominator

Step 3. Add the fractions by adding the numerators

Step 4. Simplify the fraction if it is improper

These steps may be performed *somewhere* in the expression. In addition to these four steps the unit and zero rules may be used *whenever* possible. For example, the addition of the fractions in $\frac{2}{3} + \frac{2}{5}$ is performed as follows:

$$\tfrac{2}{3} + \tfrac{2}{5} \overset{\text{RENAME}}{\Longrightarrow} \tfrac{10}{15} + \tfrac{2}{5} \overset{\text{RENAME}}{\Longrightarrow} \tfrac{10}{15} + \tfrac{6}{15} \overset{\text{ADD}}{\Longrightarrow} \tfrac{16}{15} \overset{\text{SIMPL}}{\Longrightarrow} 1 + \tfrac{1}{15}$$

2.3 A language for strategies for exercises

The two example strategies for rewriting arithmetic expressions given in the previous section give an intuition for strategies for exercises. In this section we define a language for specifying such strategies. We explore a number of combinators to combine simple strategies into more complex ones. We start with a set of basic combinators, and gradually move on to more powerful combinators. We define the semantics of these combinators, and the laws they satisfy. Our strategy language is very similar to the language for specifying CFGs, and we will describe the equivalent concepts when applicable. This strategy language has been used extensively in domain reasoners for various mathematical domains (Heeren et al., 2010).

We use a collection of standard combinators to combine strategies, resulting in more complex strategy descriptions. The semantics of the combinators is given in terms of the *language* of a strategy. The language of a strategy is a set of sentences, where each sentence is a sequence of symbols. The symbols in our language are rewrite or refinement rules. We use $a, b, c, ...$ to denote symbols, and x, y, z for sentences (sequences of such symbols). As usual, we write ϵ for the empty sequence, and xy (or ax) for concatenation. Function $\mathcal{L}$ generates the language of a strategy. Note that we also use x and y for variable names. For example in λ-abstractions, which are given as argument to the *fix* combinator.

2.3.1 Rules

The basic components of our strategy language are rewrite and refinement rules. Such a rule is the smallest building block to construct composite strategies, and corresponds to a terminal symbol in a CFG. A rule rewrites or refines an expression. A rule is applicable to an expression if the left-hand side of the rule matches, possibly using unification. If the expression matches, it is replaced by the right-hand side of the rule, in which the metavariables are substituted by the matched expressions.

We distinguish two kinds of rules: *minor* rules and *major* (normal) rules. Major rules typically are the rules a student applies, such as the rules for manipulating arithmetic expressions listed in Figure 2.1. Minor rules are used to perform administrative tasks, such as moving down into a term, updating an environment, or automatically simplifying a term, such as replacing $x + x$ by $2 \cdot x$. The implementer of a strategy determines whether or not a rule is major or minor. The predicate *isMinor* is used to determine whether or not a rewrite or refinement rule is minor. The example derivation for simplifying the expression with powers given above only shows the major rules. The minor rules that move the focus into a term, for example for moving from $(a^3 \cdot a^4)^2$ to $a^3 \cdot a^4$ to apply rule ADDPOW, are not shown in the derivation. A major rule may be turned into a minor rule to decrease the

granularity of intermediate steps (e.g., rewriting $3 + 4$ to 7 in the example derivation of simplifying an arithmetic expression with powers), or increase the difficulty of an exercise. It is advisable to make only rules that the user can apply major, since major rules will be shown to the user in derivations, and be given as hints. For example, if the focus in the editor cannot be set by the user, it is unwise to make a rule that changes the focus in a term a major rule.

When tracking a student working on an exercise we maintain an environment, for example for storing extra information. An environment stores additional information at intermediate steps in a derivation, such as auxiliary results. An environment is defined as follows:

Definition 2.1. *An environment is a set of key/value pairs, which can be added, removed, consulted, and updated.*

Recall the strategy for adding fraction from Section 2.2. The first step is to determine the LCD. The rule *LCD*, which stores the LCD in the environment, is implemented as a minor rule. In our approach a rewrite rule does not operate on just an expression, but on the product of an environment and an expression. A rewrite rule is defined as follows.

Definition 2.2. *A rewrite rule r is a binary relation on the product of an environment Γ and an expression: $(\Gamma_1 \times e_1) \overset{r}{\rightsquigarrow} (\Gamma_2 \times e_2)$. A rewrite rule is tagged with a boolean indicating whether or not it is a minor rule. If the rewrite rule does not change the environment we use $e_1 \overset{r}{\rightsquigarrow} e_2$ as a short-hand notation for $(\Gamma \times e_1) \overset{r}{\rightsquigarrow} (\Gamma \times e_2)$.*

Consider the derivation of adding two fractions from the previous section. The full derivation, including minor rules, is as follows:

$$(\emptyset, \tfrac{2}{3} + \tfrac{2}{5}) \overset{LCD}{\rightsquigarrow} (\{(n, 15)\}, \tfrac{2}{3} + \tfrac{2}{5}) \overset{\text{RENAME}}{\rightsquigarrow} (\{(n, 15)\}, \tfrac{10}{15} + \tfrac{2}{5}) \overset{\text{RENAME}}{\rightsquigarrow}$$

$$(\{(n, 15)\}, \tfrac{10}{15} + \tfrac{6}{15}) \overset{\text{ADD}}{\rightsquigarrow} (\{(n, 15)\}, \tfrac{16}{15}) \overset{\text{SIMPL}}{\rightsquigarrow} (\{(n, 15)\}, 1 + \tfrac{1}{15})$$

The language of a strategy consisting of a single rule is just that rule:

$$\mathcal{L}(r) = \{r\}$$

2.3.2 Choice

The choice combinator <|> allows solving a problem in two different ways. In CFGs, choice is introduced by having multiple production rules for a non-terminal symbol, which can be combined by means of the |-symbol, which explains our

notation. The language generated by choice is the union of the languages of the arguments:

$$\mathcal{L}\left(\sigma <|> \tau\right) \;=\; \mathcal{L}\left(\sigma\right) \cup \mathcal{L}\left(\tau\right)$$

Where σ and τ are meta-variables that range over strategies. The δ combinator is a strategy that always fails. Its set of sentences is empty:

$$\mathcal{L}\left(\delta\right) \;=\; \emptyset$$

It is a unit element of $<|>$:

$$\begin{aligned} \delta <|> \sigma \;&=\; \sigma \\ \sigma <|> \delta \;&=\; \sigma \end{aligned}$$

These properties hold for all strategies σ.

2.3.3 Sequence

Often, an exercise is solved in a particular order: when adding two fractions we first need to rename the fractions so that they have the same denominator, before we can add the numerators.

The *sequence* combinator, denoted by $<\star>$, applies its second argument strategy after its first, thus allowing programs that require multiple refinement steps to be applied in some order. The right-hand side of a production rule in a CFG consists of a sequence of symbols. The sentences in the language of sequence are concatenations of sentences from the languages of the component strategies:

$$\mathcal{L}\left(\sigma <\star> \tau\right) \;=\; \{xy \mid x \in \mathcal{L}\left(\sigma\right), y \in \mathcal{L}\left(\tau\right)\}$$

The ε combinator is a strategy that always succeeds. Its set of sentences contains just the empty sentence:

$$\mathcal{L}\left(\varepsilon\right) \;=\; \{\epsilon\}$$

The δ combinator is a zero element of $<\star>$, and ε is a unit element. The following properties hold for all strategies σ:

$$\begin{aligned} \delta <\star> \sigma \;&=\; \delta \\ \sigma <\star> \delta \;&=\; \delta \end{aligned}$$

$$\begin{aligned} \varepsilon <\star> \sigma \;&=\; \sigma \\ \sigma <\star> \varepsilon \;&=\; \sigma \end{aligned}$$

2.3.4 Interleave

In the example strategy for adding fractions we state that whenever possible the rules for units and zeroes may be applied. The steps for adding fractions, and removing units or zeroes may be taken in any order. In the following example:

$$\tfrac{2}{3} + \tfrac{3}{4} + 0$$

we may first do the fraction addition and then remove the zero, or vice versa. To support this behaviour, we introduce the *interleave* combinator, denoted by <%>. This combinator expresses that the steps of its argument strategies have to be applied, but that the steps can be interleaved. For example, the result of interleaving a strategy *abc* that recognises the sequence of three symbols *a*, *b*, and *c*, with the strategy *de* that recognises the sequence of two symbols *d* and *e* (that is, *abc* <%> *de*) results in the following set:

$$\{abcde, abdce, abdec, adbce, adbec, adebc, dabce, dabec, daebc, deabc\}$$

Interleaving sentences. To define the semantics of interleave, we first define an interleave operator on sentences. The interleaving of two sentences (x <%> y) can be defined conveniently in terms of left-interleave (denoted by x %> y, and also known as the left-merge operator (Bergstra and Klop, 1985)), which expresses that the first symbol should be taken from the left-hand side operand. The algebra of communicating processes (ACP) field traditionally defines interleave in terms of left-interleave (and "communication interleave") to obtain a sound and complete axiomatisation (Fokkink, 2000).

$$\begin{aligned}
\epsilon \mathbin{<\%>} x &= \{x\} \\
x \mathbin{<\%>} \epsilon &= \{x\} \\
x \mathbin{<\%>} y &= x \mathbin{\%>} y \cup y \mathbin{\%>} x \quad (x \neq \epsilon \wedge y \neq \epsilon)
\end{aligned}$$

$$\begin{aligned}
\epsilon \mathbin{\%>} y &= \emptyset \\
ax \mathbin{\%>} y &= \{az \mid z \in x \mathbin{<\%>} y\}
\end{aligned}$$

The set *abc* %> *de* (where *abc* and *de* are now sentences) only contains the six sentences that start with symbol *a*. The number of interleavings for two sentences of lengths n and m equals $\frac{(n+m)!}{n!m!}$. This number grows quickly with longer sentences. Hoare (1985) gives an alternative definition for interleaving. The interleaving of two sequences is defined by these three laws:

$$\begin{aligned}
\epsilon \in (y \mathbin{<\%>} z) &\Leftrightarrow y = z = \epsilon \\
x \in (y \mathbin{<\%>} z) &\Leftrightarrow x \in (z \mathbin{<\%>} y) \\
ax \in (y \mathbin{<\%>} z) &\Leftrightarrow (\exists\, y' : y = ay' \wedge x \in (y' \mathbin{<\%>} z)) \\
&\quad \vee \ (\exists\, z' : z = az' \wedge x \in (y \mathbin{<\%>} z'))
\end{aligned}$$

Interleaving sets. The operations for interleaving sentences can be lifted to work on sets of sentences by considering all combinations of elements from the two sets. Let X, Y, and Z be sets of sentences. The lifted operators are defined as follows:

$$\begin{aligned} X \mathbin{<\%>} Y &= \bigcup \{ x \mathbin{<\%>} y \mid x \in X, y \in Y \} \\ X \mathbin{\%>} Y &= \bigcup \{ x \mathbin{\%>} y \mid x \in X, y \in Y \} \end{aligned}$$

For instance, $\{a, ab\} \mathbin{<\%>} \{c, cd\}$ yields a set containing 14 elements:

$$\{abc, abcd, ac, acb, acbd, acd, acdb, ca, cab, cabd, cad, cadb, cda, cdab\}$$

From these definitions, it follows that the lifted operator for interleaving is commutative, associative, and has $\{\epsilon\}$ as identity element. The left-interleave operator is not commutative nor associative, but has the interesting property that $(X \mathbin{\%>} Y) \mathbin{\%>} Z$ is equal to $X \mathbin{\%>} (Y \mathbin{<\%>} Z)$.

Atomicity. Interleaving assumes that there exist atomic steps , and we introduce a construct to introduce atomic blocks within sentences. In such a block, no interleaving should occur with other sentences. We write $\langle x \rangle$ to make sequence x atomic: if x is a singleton, the angle brackets may be dropped. Atomicity obeys some simple laws:

$$\begin{aligned} \langle \epsilon \rangle &= \epsilon && \text{(the empty sequence is atomic)} \\ \langle a \rangle &= a && \text{(all primitive symbols are atomic)} \\ \langle x \langle y \rangle z \rangle &= \langle xyz \rangle && \text{(nesting of atomic blocks has no effect)} \end{aligned}$$

In particular, it follows that $\langle \langle x \rangle \rangle = \langle x \rangle$. Atomic blocks nicely work together with the definitions given for the interleaving operators, including the lifted operators: sentences now consist of a sequence of atomic blocks, where each block itself is a non-empty sequence of symbols. For instance, $a\langle bc \rangle \mathbin{<\%>} \langle de \rangle f$ will return:

$$\{abcdef, adebcf, adefbc, deabcf, deafbc, defabc\}$$

In the end, when no more interleaving takes place, the blocks have no longer any meaning, and can be discarded.

The interleaving operators. The semantics of the interleaving operators is defined in terms of the lifted operators:

$$\begin{aligned} \mathcal{L}\left(\langle \sigma \rangle\right) &= \{ \langle x \rangle \mid x \in \mathcal{L}\left(\sigma\right) \} \\ \mathcal{L}\left(\sigma \mathbin{<\%>} \tau\right) &= \mathcal{L}\left(\sigma\right) \mathbin{<\%>} \mathcal{L}\left(\tau\right) \\ \mathcal{L}\left(\sigma \mathbin{\%>} \tau\right) &= \mathcal{L}\left(\sigma\right) \mathbin{\%>} \mathcal{L}\left(\tau\right) \end{aligned}$$

The interleave combinator satisfies several laws: it is commutative and associative, and has ε as identity element:

$$\begin{array}{lcl} \sigma \mathbin{<\%>} \tau & = & \tau \mathbin{<\%>} \sigma \\ \sigma \mathbin{<\%>} (\tau \mathbin{<\%>} \upsilon) & = & (\sigma \mathbin{<\%>} \tau) \mathbin{<\%>} \upsilon \\ \sigma \mathbin{<\%>} \varepsilon & = & \sigma \end{array}$$

Interleaving distributes over choice

$$\sigma \mathbin{<\%>} (\tau \mathbin{<|>} \upsilon) \;=\; (\sigma \mathbin{<\%>} \tau) \mathbin{<|>} (\sigma \mathbin{<\%>} \upsilon)$$

Left-interleave also distributes over choice. The operator that makes a strategy atomic is idempotent, and distributes over choice $\langle \sigma \mathbin{<|>} \tau \rangle = \langle \sigma \rangle \mathbin{<|>} \langle \tau \rangle$. Many more properties can be found in the literature on ACP (Bergstra and Klop, 1985).

2.3.5 Label

When solving an exercise, a student may ask for a hint at any time. A tutor should take the actions of the student until she asks for a hint into account. The steps that a student has taken correspond to a particular location in the strategy. We mark positions in the strategy with a *label*, which allows us to describe feedback. The *label* combinator takes a string (or a value of another type that is used for labelling purposes) and a strategy as arguments, and offers the possibility to attach a text to the argument strategy.

$$\mathcal{L}\,(\mathit{label}\ \ell\ \sigma) \;=\; \{\,\textsc{Enter}_\ell\ x\ \textsc{Exit}_\ell \mid x \in \mathcal{L}\,(\sigma)\,\}$$

This interpretation introduces the special rules ENTER and EXIT (parameterised by some label ℓ) that show up in sentences. The ENTER and EXIT rules are minor rules that are only used for tracing positions in strategies. A label is *active* in a sentence if a sentence contains the ENTER rule for a this label, but not the EXIT rule. Except for tracing, the label combinator is semantically the identity function.

2.3.6 Recursion

One aspect we haven't discussed yet is recursion. Recursion is used for example to specify that a user replaces *all* occurrences of a particular expression in a program by another expression. Recursion is specified by means of the fixed-point operator *fix*, which takes as argument a function that maps a strategy to a new strategy. The language of $\mathit{fix}\,f$ is defined by:

$$\mathcal{L}\,(\mathit{fix}\,f) \;=\; \mathcal{L}\,(f\ (\mathit{fix}\,f))$$

The *fix* operator is mainly used to express repetition in strategies. It is the responsibility of the user to specify meaningful fixed-points. Having a fixed-point combinator implies that we are vulnerable to non-termination. By using the *fix* combinator we make recursion explicit. This allows the strategy recogniser to deal with recursive strategies in a special way. For example, in our recogniser for the strategy language we specify a cutoff for the fixed-point operator. The recogniser stops the execution of a strategy when the cutoff is reached, and reports an error message. The recogniser does so by counting the number of times that the *fix* combinator has been unrolled.

2.3.7 Applicability checks

The applicability check symbol ($\sim\sigma$) in our strategy language allows us to specify that a certain strategy is not applicable to the current expression.

Definition 2.3. *An applicability check takes a strategy as argument and produces a rule that returns its input expression unchanged if the strategy is* not *applicable to the expression. Otherwise, if the strategy is applicable, the rule fails.*

Depending on the applicability of a strategy the check either returns a singleton set containing the expression it is applied to, or the empty set. A more general variant of this combinator is *check*, which receives a predicate as argument instead of a strategy. The language of an applicability check is just that check:

$$\mathcal{L}\,(\sim\sigma) \;=\; \{\sim\sigma\}$$

The set of symbols in our language is extended with applicability checks.

Having defined the applicability check, we are now able to specify greedy strategy combinators. Greedy combinators will apply their argument strategies whenever possible. The next subsection shows some examples of greedy strategy combinators, such as *repeat*.

2.3.8 Derived Combinators

Extended Backus-Naur form (EBNF) extends the notation for grammars, and offers three new constructions that are often encountered in practice: zero or one occurrence (option), zero or more occurrences (closure), and one or more occurrences (positive closure). We introduce three new strategy combinators: *many* σ repeats strategy σ zero or more times, *many1* applies σ at least once, and *option* σ may or may not apply strategy σ. We define these combinators using the basic combinators:

$$\begin{aligned}
&\mathit{many}\ \ \sigma = \mathit{fix}\ (\lambda x \to \varepsilon \mathbin{<|>} (\sigma \mathbin{<\star>} x))\\
&\mathit{many1}\ \sigma = \sigma \mathbin{<\star>} \mathit{many}\ \sigma\\
&\mathit{option}\ \ \sigma = \sigma \mathbin{<|>} \varepsilon
\end{aligned}$$

Observe the use of the fixed-point combinator *fix* in the definition of *many*. Unfolding the *many* σ strategy results in:

$$\begin{aligned} &\mathit{many}\ \sigma \\ &\quad = \varepsilon \mathbin{<|>} (\sigma \mathbin{<\star>} \mathit{many}\ \sigma) \\ &\quad = \varepsilon \mathbin{<|>} (\sigma \mathbin{<\star>} (\varepsilon \mathbin{<|>} (\sigma \mathbin{<\star>} \mathit{many}\ \sigma))) \\ &\quad = \ldots \end{aligned}$$

It is quite common for an EDSL to introduce a rich set of combinators on top of a (small) set of basic combinators. The derived strategy combinators are useful in formulating rewrite strategies for exercises.

The next combinators we introduce make use of the primitive applicability check symbol ($\sim$). Using this symbol, we can define the left-biased choice combinator ($\triangleright$) as follows:

$$\sigma_1 \triangleright \sigma_2 = \sigma_1 \mathbin{<|>} (\sim\sigma_1 \mathbin{<\star>} \sigma_2)$$

The strategy σ_2 is only considered when σ_1 is not applicable. Other combinators, such as *try* and *repeat* are similar to their EBNF counterparts.

$$\begin{aligned} &\mathit{try}\ \sigma \quad = \sigma \triangleright \varepsilon \\ &\mathit{repeat}\ \sigma = \mathit{fix}\ (\lambda x \to \mathit{try}\ (\sigma \mathbin{<\star>} x)) \end{aligned}$$

The *try* combinator applies a strategy when it is applicable. The *repeat* combinator applies a strategy as many times as possible.

2.3.9 Navigation

In many domains, terms can be combined to form new terms. For instance, an arithmetic expression may have several subexpressions. We do not only want to apply rules and strategies to the top-level term, but also to subterms. We therefore need some additional combinators to indicate that a strategy or rule should be applied somewhere inside a term. Besides the derived combinators from the previous subsection, we add a set of traversal combinators to our strategy language. Traversal combinators traverse a term, and for example perform rewrite rules or strategies *somewhere* or *bottomUp*. We use a number of administrative rules for navigating through the abstract syntax tree (AST) of an expression: UP, DOWN, LEFT, and RIGHT. The minor rule DOWN takes a function as argument, which decides which child to select based on the environment. Using DOWN we construct the minor rule DOWNS that selects all children. Recall that a rule is a binary relation, and can return multiple results.

Navigation is implemented by means of the zipper (Huet, 1997), which is an efficient data structure to set and move a focus in an expression. The zipper can

be seen as a combination of an expression and its context. An alternative way to navigate is to use position information of (sub)expressions. An implementation using the latter approach uses a list of integers denoting a path from the top of the expression to the subexpression in focus. This approach is not as efficient and type-safe as a zipper, since the AST needs to be traversed to retrieve the subexpression in focus, and since it is possible to specify paths that do not correspond to a position in the tree.

We can lift rewrite rules to work on a zipper. Lifted rewrite rules are defined as follows:

Definition 2.4. *Let Γ be an environment, and ϕ a zipper, which is an expression in focus together with its context. A lifted rewrite rule is a binary relation on the product of an environment and a zipper: $(\Gamma_1 \times \phi_1) \overset{r}{\rightsquigarrow} (\Gamma_2 \times \phi_2)$. The rewrite rule r is applied to expression in focus.*

Traversal combinators. Many traversal combinators use the *once* combinator:

$$once\ \sigma = \textsc{Downs} <\star> \sigma <\star> \textsc{Up}$$

The *once* combinator takes a strategy as argument, and applies it to a direct child of the expression currently in focus. After applying *once* the focus is again at the top-level expression. The *once* combinator applies a strategy to a direct child of an expression. If there are multiple children, *once* returns multiple results. *once* makes use of the minor rule DOWNS. So an application of a strategy constructed with the *once* combinator may have more than one result, depending on whether or not strategy σ is applicable to a child.

The traversal combinator *somewhere* applies a strategy to a single subexpression (including the expression itself).

$$somewhere\ \sigma = fix\ (\lambda x \to \sigma <|> once\ x)$$

If we want to be more specific about where to apply a strategy, we can use *bottomUp* or *topDown*:

$$bottomUp\ \sigma = fix\ (\lambda x \to once\ x \triangleright \sigma)$$
$$topDown\ \sigma\ = fix\ (\lambda x \to \sigma \triangleright once\ x)$$

These combinators search for a suitable location to apply an argument strategy in a bottom-up or top-down fashion, without imposing an order in which the children are visited. These combinators do not apply their argument strategy exhaustively, but just once.

Navigation operators navigate through the abstract syntax of the domain on which the rewrite rules are specified. Recall the definition of simple arithmetic

expressions at the beginning of this chapter (Section 2.2). We define a zipper for navigation on this domain as follows:

$$
\begin{array}{rcll}
\phi & ::= & [\![Expr]\!] & \\
& | & \phi + Expr & | \; Expr + \phi \\
& | & \phi - Expr & | \; Expr - \phi \\
& | & \dfrac{\phi}{Expr} & | \; \dfrac{Expr}{\phi} \\
& | & \phi \cdot Expr & | \; Expr \cdot \phi \\
& | & \phi^{Expr} & | \; Expr^{\phi} \\
& | & -[\![Expr]\!] &
\end{array}
$$

In the grammar for the zipper, the expression between double square brackets is the expression in focus. The focus can appear in the left-hand or right-hand side of an operation, in the numerator or denominator of a fraction, or in the base or exponent of a power. The focus can be moved to a different part of the expression. The minor rules for navigation, such as Up and Left, are defined by analysing the various forms of the zipper. For example, applying the Up minor rule to $[\![a]\!]^7$ results in $[\![a^7]\!]$, and moving the focus to the right in $[\![3]\!] + 4$ yields $3 + [\![4]\!]$.

Using generic programming techniques (Hinze et al., 2007; Hinze and Jeuring, 2003), we can define these navigation functions once and for all, and use them on every domain.

2.3.10 Example strategies revisited

In Section 2.2 we presented two strategies for simplifying arithmetic expressions. Having defined a set of strategy combinators, we can now give a precise definition of these strategies in terms of our combinators. We use the rules defined in Figure 2.1.

Strategy 1: simplifying expressions with powers. The informal description of the procedure to simplify expression with powers can be translated straightforwardly in terms of our strategy combinators as follows:

```
simplifyPower =
  label ℓ (repeat (bottomUp (addPow <|> mulPow <|> distPow)))
    where ℓ = "simplifyPower"
```

Applying the *simplifyPower* strategy to the expression $(a^3 \cdot a^4)^2$ gives the following derivation:

$$[\![(a^3 \cdot a^4)^2]\!] \overset{\text{ENTER}_\ell}{\rightsquigarrow} [\![(a^3 \cdot a^4)^2]\!] \overset{\text{DOWN}}{\rightsquigarrow} [\![a^3 \cdot a^4]\!]^2 \overset{\text{APPCHECK}}{\rightsquigarrow}$$

$$[\![a^3 \cdot a^4]\!]^2 \overset{\text{ADDPOW}}{\rightsquigarrow} [\![a^7]\!]^2 \overset{\text{UP}}{\rightsquigarrow} [\![(a^7)^2]\!] \overset{\text{APPCHECK}}{\rightsquigarrow} [\![(a^7)^2]\!] \overset{\text{MULPOW}}{\rightsquigarrow}$$

$$[\![a^{14}]\!] \overset{\text{APPCHECK}}{\rightsquigarrow} [\![a^{14}]\!] \overset{\text{EXIT}_\ell}{\rightsquigarrow} [\![a^{14}]\!]$$

The APPCHECK (applicability check) is introduced by the *repeat* and *bottomUp* combinator. The navigation rules do not work directly on an expression, but on the zipper containing the expression. As a consequence, if a strategy uses a traversal combinator it is only applicable to an expression in a context.

Strategy 2: adding fractions. The description of the procedure to add fractions in an arithmetic expression can be translated into a strategy specification as follows:

```
addFractions   =  label "addFractions"
               $  repeat (somewhere addTwoFractions)
              <%> repeat ⟨somewhere unitAndZeroes⟩
addTwoFractions = label "addTwoFractions"
                $  LCD
               <*> repeat (somewhere RENAME)
               <*> ADD
               <*> try SIMPL
unitAndZeroes  =  label "unitAndZeroes"
               $  UNITADD
              <|> UNITSUB
              <|> UNITMUL
              <|> ZEROMUL
```

The strategy *addFractions* consists of two substrategies: *addTwoFractions*, which expresses how to add two fractions in an expression, and *unitAndZeroes*, which cleans up an expression. These two substrategies are combined with the interleave operator. Furthermore, the strategy contains the labels `"addFractions"`, `"addTwoFractions"` and `"unitsAndZeroes"`, and uses the rules for fractions, units, and zeroes given in Figure 2.1. The LCD rule is somewhat different: it is a minor rule that does not change the term, but calculates the LCD and stores this in an environment. The rule RENAME for renaming a fraction uses the computed least common denominator, which is retrieved from the context, to determine the value of a in its right-hand side. Note that we have put the *somewhere unitAndZeroes* substrategy in an atomic block. This way we prevent interleaving of navigation rules, which might lead to unexpected behaviour.

2.3.11 Overview

The different components of our strategy language have been introduced in the previous subsections. We summarise the definitions of the various strategy combinators in the following definition.

Definition 2.5. *Let fix be the fixed-point combinator fix $f = f\ (fix\ f)$, and ℓ a label. A strategy is an element of the language of the following grammar:*

$$
\begin{array}{lcl}
\sigma & ::= & a \\
& | & \sigma \mathbin{<|>} \sigma \quad | \quad \delta \\
& | & \sigma \mathbin{<\star>} \sigma \quad | \quad \varepsilon \\
& | & label\ \ell\ \sigma \\
& | & fix\ f \\
& | & \langle \sigma \rangle \quad | \quad \sigma \mathbin{<\%>} \sigma \quad | \quad \sigma \mathbin{\%>} \sigma \\
a & ::= & r \\
& | & \sim \sigma
\end{array}
$$

The components of the grammar for σ are called strategy combinators. *Two (sub)strategies can be combined into a strategy using the sequence ($<\star>$) or choice ($<|>$) combinator, with ε (always succeeds) and δ (always fails) as unit elements, respectively. A strategy can be tagged with a label (label). The fix combinator returns the fixed-point of a function (f) that takes a strategy as argument and returns a strategy. Steps from two (sub)strategies may be interleaved using the interleave combinator ($<\%>$). Strategies that are marked atomic ($\langle\sigma\rangle$) cannot be interleaved. The left-interleave combinator ($\%>$) makes certain that the first step is taken from the left strategy, the remaining steps may be interleaved. The non-terminal symbol a is either a rewrite or refinement rule r, or an applicability check $\sim\sigma$ that takes a strategy as argument and returns a rule that checks if σ is not applicable.*

This definition corresponds to the definition of a context-free grammar, extended with interleaving, fixed-points, with an alphabet consisting of rewrite or refinement rules and applicability checks.

The following definition gives the semantics for our strategy combinators:

$$
\begin{array}{lcl}
\mathcal{L}\ (a) & = & \{a\} \\
\mathcal{L}\ (\sigma \mathbin{<|>} \tau) & = & \mathcal{L}\ (\sigma) \cup \mathcal{L}\ (\tau) \\
\mathcal{L}\ (\delta) & = & \emptyset \\
\mathcal{L}\ (\sigma \mathbin{<\star>} \tau) & = & \{xy \mid x \in \mathcal{L}\ (\sigma), y \in \mathcal{L}\ (\tau)\} \\
\mathcal{L}\ (\varepsilon) & = & \{\epsilon\} \\
\mathcal{L}\ (label\ \ell\ \sigma) & = & \{\textsc{Enter}_\ell\ x\ \textsc{Exit}_\ell \mid x \in \mathcal{L}\ (\sigma)\} \\
\mathcal{L}\ (fix\ f) & = & \mathcal{L}\ (f\ (fix\ f)) \\
\mathcal{L}\ (\langle\sigma\rangle) & = & \{\langle x \rangle \mid x \in \mathcal{L}\ (\sigma)\} \\
\mathcal{L}\ (\sigma_1 \mathbin{<\%>} \sigma_2) & = & \mathcal{L}\ (\sigma_1) \mathbin{<\%>} \mathcal{L}\ (\sigma_2) \\
\mathcal{L}\ (\sigma_1 \mathbin{\%>} \sigma_2) & = & \mathcal{L}\ (\sigma_1) \mathbin{\%>} \mathcal{L}\ (\sigma_2)
\end{array}
$$

This definition can be used to tell whether a sequence of rewrite or refinement rules and applicability checks follows a strategy or not: the sequence of rules and checks should be a sentence in the language generated by the strategy, or a prefix of a sentence, since we solve exercises incrementally.

2.3.12 Left-recursion

To use strategies for tracking student behaviour and giving feedback, we impose a restriction on the form of strategies. We do not allow left-recursive strategies. This is a common restriction, which is imposed by many parsing algorithms on context-free grammars.

A context-free grammar is left-recursive if it contains a nonterminal that can be rewritten in one or more steps using the productions of the grammar to a sequence of symbols that starts with the same nonterminal. The same definition applies to strategies. For example, the following strategy is left-recursive:

$$\mathit{leftRecursive} = \mathit{fix}\ (\lambda x \rightarrow x \mathbin{<\!\star\!>} \textsc{add})$$

The left-recursion is obvious in this strategy, since x is in the leftmost position in the body of the abstraction. However, in our semantics non-left-recursive strategies sometimes display left-recursive behaviour. Strategies with leading minor rules may or may not display left-recursive behaviour. Strictly speaking, these strategies are not left-recursive because the strategy grammar does not differentiate between minor and major rules. Left-recursive behaviour is not always easy to spot. For example, if we use a minor rule that increases a counter in the environment, which is an action that always succeeds, the strategy displays left-recursive behaviour. On the other hand, in *leftRecursive′*:

$$\mathit{leftRecursive}' = \mathit{fix}\ (\lambda x \rightarrow \textsc{Down} \mathbin{<\!\star\!>} x \mathbin{<\!\star\!>} \textsc{add})$$

the minor rule DOWN is applied repeatedly until we reach the leaf of an expression tree, and stop. This strategy does not behave like a left-recursive strategy.

Top-down recursive parsing using a left-recursive context-free grammar is difficult. A grammar expressed in parser combinators (Hutton, 1992) is not allowed to be left-recursive. Trying to determine the next possible symbol(s) of a left-recursive strategy will loop. This problem would probably disappear if we would use a bottom-up parsing algorithm, but that would lead to other restrictions, which sometimes are harder to spot and repair (compare determining whether or not a grammar is LR(1) with determining whether or not a grammar is left-recursive). Left-recursion can sometimes be solved by using so-called chain combinators (Fokker, 1995).

Combinator	Description
σ <*> τ	first σ, then τ
σ <|> τ	either σ or τ
ε	always succeeding strategy
δ	always failing strategy
fix f	fixed point combinator
label ℓ σ	attach label ℓ to σ
$\langle\sigma\rangle$	σ cannot be interleaved
σ <%> τ	interleave σ and τ
σ %> τ	as interleave, but start with a step from σ

Combinator	Description
$\sim\sigma$	succeed if σ is not applicable
many σ	apply σ zero or more times
many1 σ	apply σ one or more times
option σ	either apply σ or not
repeat σ	apply σ as long as possible
try σ	apply σ once if possible
$\sigma \triangleright \tau$	apply σ, or else τ
somewhere σ	apply σ at some location
topDown σ	apply σ top-down
bottomUp σ	apply σ bottom-up

Figure 2.2: Summary table of strategy combinators

2.3.13 Reflections

Figure 2.2 presents an overview of the strategy combinators introduced so far. Is this set of strategy combinators rich enough to describe many of the strategies that are used in exercises? We are not sure; other strategy implementers may want to use different abstractions. However, the language can be easily extended with more combinators. In fact, this is probably the greatest advantage of using an EDSL instead of defining a new, stand-alone language. We have implemented about 70 strategies for various mathematical domains. We believe that the set of combinators is sufficient for specifying the kind of strategies that are needed in interactive exercise assistants for mathematics and functional programming, which aim at providing advanced feedback.

Producing a strategy is like programming, and might require quite some effort. However, due to compositionality of the strategy combinators, strategies are reusable. For example, the strategies for adding fractions and working with powers can easily be integrated into a strategy that allows to apply both.

Our strategy language is particularly suited for describing procedures for solving exercises in well-structured domains. In such domains it is easy to formally describe the rewrite rules that can be applied to expressions within the domain. It is much harder to use strategies in domains with less structure. For example, we do not know how to write a strategy for constructing a unified modelling language-model (UML) for a piece of software, giving a textual description of the purposes of the software.

Our strategies should not be confused with tutorial strategies, which describe how to teach particular content. For example, a tutorial strategy may describe that it is better to first give a worked-out example, and only then require students to

take some intermediate steps themselves, as in the 4C/ID-model (Van Merriënboer and Kirschner, 2007). Or it may describe that before giving exercises, the complete procedure should be explained and the underlying theorems be proved. We think our strategies can be used in any of such tutorial strategies.

2.4 Strategy functions

The semantics for the different components of our strategy language has been introduced in the previous section. The language definition ($\mathcal{L}$) can be used to determine whether a sequence of rewrite or refinement rules and applicability checks is a sentence in the language generated by the strategy, or a prefix of a sentence. Not all sequences make sense, however. An exercise gives us an initial expression, and we are only interested in sequences of rules that can be applied successively to this expression. In the remainder of this section we complete the semantics by explaining how we use a strategy to transform an expression. We do so by defining a number of *strategy functions* that resemble functions on CFGs, such as *firsts*.

We start by defining a state, which we use to capture (intermediate) answers:

Definition 2.6. *A state is the product of an environment, an expression in focus (a zipper), and a strategy.*

2.4.1 Empty

To check if an expression can be accepted as a final answer (i.e., the last element of a derivation), we need to determine whether or not the empty sentence is a member of the language of a strategy. For this purpose we define a function *empty*, similar to the grammar function *empty* defined for CFGs. We first define an auxiliary function that returns all sentences that consist of minor rules only:

Definition 2.7. *Function minorSentences returns all sentences in the language of a strategy that consist of minor rules only:*

$$minorSentences\ (\sigma) = \{ r_1 \ldots r_n \mid r_1 \ldots r_n \ \in\ \mathcal{L}\ (\sigma), \forall\ i\ \in\ 1 \ldots n\ .\ isMinor\ (r_i) \}$$

Note that the set given by *minorSentences* includes the empty sentence ϵ if the language of σ contains the empty sentence ϵ. We need this function to determine if there are any trailing minor rules after the last major rewrite rule. The last major rewrite rule is the last action that a student needs to perform. In a sequence of rewrite rules the last major rule is not necessarily the last rule. There may be trailing minor rules after the last major rule. We define the *empty* function in terms of the *minorSentences* function:

Definition 2.8. *Function empty checks whether or not the language of a strategy contains the empty sentence ϵ, or a sentence consisting of minor rules only:*

$$empty\ (\sigma) = minorSentences\ (\sigma)\ \neq\ \emptyset$$

Both functions are easily lifted to take a state as argument. The lifted functions operate on the strategy in a state. For example, the lifted version of *empty* is defined as follows:

$$empty\ (\Gamma \times \phi \times \sigma) = empty\ (\sigma)$$

where Γ is an environment, and ϕ a zipper.

2.4.2 Firsts

The smallest action that can be performed with a strategy is a *step*: the application of a rewrite rule. Before we define that, we define a relation that splits a strategy into its first rule or applicability check and the remaining strategy. This function resembles the *firsts* grammar function.

Definition 2.9. *The relation $\mapsto$ splits a strategy into a rule or an applicability check and the remaining strategy: $\sigma_1 \mapsto a \lessdot\star\gtrdot \sigma_2$.*

$$a \mapsto a \lessdot\star\gtrdot \epsilon \qquad \frac{\sigma_1 \mapsto a \lessdot\star\gtrdot \sigma_3}{\sigma_1 \lessdot\star\gtrdot \sigma_2 \mapsto a \lessdot\star\gtrdot (\sigma_3 \lessdot\star\gtrdot \sigma_2)}$$

$$\frac{\epsilon\ \in\ \mathcal{L}\ (\sigma_1) \qquad \sigma_2 \mapsto a \lessdot\star\gtrdot \sigma_3}{\sigma_1 \lessdot\star\gtrdot \sigma_2 \mapsto a \lessdot\star\gtrdot \sigma_3} \qquad \frac{\sigma_1 \mapsto a \lessdot\star\gtrdot \sigma_3}{\sigma_1 \lessdot|\gtrdot \sigma_2 \mapsto a \lessdot\star\gtrdot \sigma_3}$$

$$\frac{\sigma_2 \mapsto a \lessdot\star\gtrdot \sigma_3}{\sigma_1 \lessdot|\gtrdot \sigma_2 \mapsto a \lessdot\star\gtrdot \sigma_3} \qquad \frac{\sigma_1 \mapsto a \lessdot\star\gtrdot \sigma_3}{\sigma_1 \lessdot\%\gtrdot \sigma_2 \mapsto a \lessdot\star\gtrdot (\sigma_3 \lessdot\%\gtrdot \sigma_2)}$$

$$\frac{\sigma_2 \mapsto a \lessdot\star\gtrdot \sigma_3}{\sigma_1 \lessdot\%\gtrdot \sigma_2 \mapsto a \lessdot\star\gtrdot (\sigma_1 \lessdot\%\gtrdot \sigma_3)} \qquad \frac{\sigma_1 \mapsto a \lessdot\star\gtrdot \sigma_3}{\sigma_1\ \%\gtrdot \sigma_2 \mapsto a \lessdot\star\gtrdot (\sigma_3 \lessdot\%\gtrdot \sigma_2)}$$

$$\frac{f\ (fix\ f) \mapsto a \lessdot\star\gtrdot \sigma}{fix\ f \mapsto a \lessdot\star\gtrdot \sigma} \qquad label\ \ell\ \sigma \mapsto \textsc{Enter}_\ell \lessdot\star\gtrdot (\sigma \lessdot\star\gtrdot \textsc{Exit}_\ell)$$

Definition 2.10. *The step operator $\xrightarrow{r}$ denotes the relation between the current state S_1 and a new state S_2 (obtained by applying the rewrite rule r).*

$$\frac{\sigma_1 \mapsto r \mathrel{<\!\star\!>} \sigma_2 \qquad (\Gamma_1 \times \phi_1) \overset{r}{\rightsquigarrow} (\Gamma_2 \times \phi_2)}{(\Gamma_1 \times \phi_1 \times \sigma_1) \xrightarrow{r} (\Gamma_2 \times \phi_2 \times \sigma_2)}$$

$$\frac{\begin{array}{c}\sigma_1 \mapsto\!\sim \sigma_2 \mathrel{<\!\star\!>} \sigma_3 \\ run\ (\Gamma_1 \times \phi_1 \times \sigma_2) = \emptyset \qquad (\Gamma_1 \times \phi_1 \times \sigma_3) \xrightarrow{r} (\Gamma_2 \times \phi_2 \times \sigma_4)\end{array}}{(\Gamma_1 \times \phi_1 \times \sigma_1) \xrightarrow{r} (\Gamma_2 \times \phi_2 \times \sigma_4)}$$

The *run* function used in the last deduction rule of the definition applies a strategy to a term in a context; its definition is given below in definition 2.12. If strategy σ_2 is not applicable to ϕ_1 then the *run* function returns the empty set.

The step relation $\xrightarrow{r}$ ignores whether or not a rule is minor, and it deals with minor and major rewrite rules in the same way. On many occasions when generating feedback we want to ignore minor rewrite rules, since we do not want to show such administrative steps to a user. We define a relation similar to step, which ignores minor rewrite rules.

Definition 2.11. *The big step operator $\xrightarrow{r}\!\!\!\!\rightarrow$ denotes the relation between a state S_1 and a new state S_2. The new state is obtained by possibly applying minor rules, followed by the application of a single major rewrite rule r. If r is the last major rule then trailing minor rules, if any, are applied as well.*

$$\frac{S_1 \xrightarrow{r_1} S_2 \qquad isMinor\ (r_1) \qquad S_2 \xrightarrow{r_2}\!\!\!\!\rightarrow S_3}{S_1 \xrightarrow{r_2}\!\!\!\!\rightarrow S_3}$$

$$\frac{S_1 \xrightarrow{r} S_2 \qquad \neg\ isMinor\ (r) \qquad minorSentences\ (S_2) = \emptyset}{S_1 \xrightarrow{r}\!\!\!\!\rightarrow S_2}$$

$$\frac{S_1 \xrightarrow{r} S_2 \qquad \neg\ isMinor\ (r) \qquad \vec{m} \in minorSentences\ (S_2) \qquad S_2 \xrightarrow{\vec{m}} S_3}{S_1 \xrightarrow{r}\!\!\!\!\rightarrow S_3}$$

where $\vec{m}$ is a sequence of minor rules and $\xrightarrow{\vec{m}}$ is the sequential application thereof.

It is important that trailing minor rules are applied when performing a big step. The application of trailing minor rules ensures that exhaustively applying the step or big step operator on a term, will end up in the same end state(s).

Definition 2.12. *The run function is the closure of* $\overset{r}{\twoheadrightarrow}$*, and generates all possible end states from a begin state S. An end state contains the empty strategy:* ε*.*

$$run\ (S) = \{\, (\Gamma \times \phi \times \varepsilon) \mid S\ \overset{\star}{\twoheadrightarrow}\ (\Gamma \times \phi \times \varepsilon) \,\}$$

where $\overset{\star}{\twoheadrightarrow}$ *is the transitive closure of* $\twoheadrightarrow$*.*

The *run* function is defined in terms of $\overset{r}{\twoheadrightarrow}$, which is in turn defined in terms of the step relation ($\overset{r}{\rightarrow}$). This step relation uses the *run* function again to determine whether or not a strategy is applicable to an expression. The step definition (2.10) and the definition of *run* given above are therefore mutually recursive. This is not a problem, because the *run* function is only applied to strategies that appear in applicability checks.

The above definitions are used in the specification of the services, which are the interface to our domain reasoners, for generating feedback. Chapter 5 will explain the services that we offer to intelligent tutoring systems.

We conclude with a soundness result. We give a theorem connecting the language concept, the function $\mathcal{L}$, to the strategy functions defined in this section. We start with the introduction of two lemmas that simplify the proof of the theorem.

Lemma 2.13. *A major language* $\mathcal{L}_m$ *is the language of a strategy without occurrences of minor rules:*

$$\mathcal{L}_m\ (\sigma) = \{\, [a \mid a \leftarrow as, \neg\ isMinor\ (a)] \mid as\ \in\ \mathcal{L}\ (\sigma) \,\}$$

Let F be the set of all splits given by the split relation for σ*:*

$$F\ (\sigma) = \{\, \sigma_i \mid \sigma \mapsto \sigma_i \,\}$$

Then the major language of σ *without the empty sentence is equal to the union of major languages of the elements in F:*

$$\mathcal{L}_m\ (\sigma)\ \setminus\ \{\epsilon\} \equiv \bigcup_{\sigma_f \in F(\sigma)} \mathcal{L}_m\ (\sigma_f)$$

Proof. By case analysis on the structure of σ. □

Definition 2.14. *Let S be a state. Then derivations* (S) *is the set of all possible sequences of states, together with the applied rule, that are generated with the big step operator:*

$$\begin{aligned} derivations\ (S) = \ & \textbf{if}\ empty\ (S) \\ & \textbf{then}\ \{[\,]\} \\ & \textbf{else}\ \{\, [(r, S')] +\!\!+ D \mid S \overset{r}{\twoheadrightarrow} S', D\ \in\ derivations\ (S') \,\} \end{aligned}$$

Theorem 2.15. *Let* $(\Gamma \times \phi \times \sigma)$ *be a state, and* $\vec{r}$ *be a sequence of rules* $[r_1 \ldots r_n]$ *such that* $(r_1, S_1) \ldots (r_n, S_n) \in$ *derivations* $(\Gamma \times \phi \times \sigma)$*. Then* $\vec{r} \in \mathcal{L}_m(\sigma)$*.*

Proof. We sketch a proof. Let σ be a strategy. We distinguish two cases in our hypothesis $[r_1 \ldots r_n] \in \mathcal{L}_m(\sigma)$:

$n = 0$: Based on Definition 2.14 we know that for every element in the derivation sequence $[r_1 \ldots r_n]$ a big step has to be performed. This means that the strategy σ is either ε or consists of a strategy with minor rules only. Based on Lemma 2.13 we know $\epsilon \in \mathcal{L}_m(\sigma)$.

$n > 0$: This case is proven by induction on the length n of the sequence $[r_1 \ldots r_n]$ using Lemma 2.13.

□

2.4.3 Non-determinism

Almost all exercises can be solved in several, correct ways. For example, consider the expression $(a^3 \cdot a^4)^2$ again, and suppose it should be solved with the strategy *simplifyPower′* which is obtained by replacing *bottomUp* by *somewhere* in the strategy *simplifyPower*.

simplifyPower′ = *label* `"simplifyPower"` $
  repeat (*somewhere* (ADDPOW <|> MULPOW <|> DISTPOW))

One of the questions we want to be able to answer is: what is the next step to solve the exercise? This type of feedback is given by one of our feedback services. In the above example there are two possibilities: DISTPOW can be applied to the entire expression, and ADDPOW is applicable to the subexpression $a^3 \cdot a^4$. Both steps are correct, but which do we choose? Making a random choice would make our feedback framework non-deterministic. This problem may show up whenever we use the choice combinator, which may also be introduced by other combinators (such as DOWNS), to combine various solution strategies. Applying a strategy that uses the choice combinator to an expression may result in multiple answers.

To prevent non-deterministic behaviour we introduce a rule ordering: a total order, denoted by $<$, on rules. For now, the ordering is only on the rules themselves, but one can imagine taking the environment into account in the rule ordering. We use the name of a rule, which is unique, to compare rewrite rules. If we need to make a choice between the same rules, that are applicable to different subexpressions, we choose the one that is applicable to the subexpression that is nearest to the root of the expression. The nearest subexpression is the one that can be reached with the fewest navigation steps in a pre-order (left-to-right) traversal.

An example rule ordering for the power domain is: ADDPOW < MULPOW < DISTPOW. A strategy implementer can define precedence among the rules using a rule ordering. When we need to make a choice between rules, we use the ordering to choose the smallest rule. In the example case, the first step in our example is ADDPOW.

We could have decided to always choose the left operand of a choice combinator to prevent non-determinism. However, we treat the choice combinator as a commutative operator, i.e., there is no semantic difference between $a <|> b$ and $b <|> a$. Always choosing the left operand of a choice would mean that for two semantically equivalent strategies we could get different results when asking for the next step. We prevent this undesired behaviour by making the choice explicit using rule ordering.

2.5 Feedback based on strategies

This section briefly sketches how we use the strategy language, as introduced in the previous sections, to give feedback to users of our tutors, or to users of other learning environments that make use of our domain reasoners. We have implemented several kinds of feedback. Most of these categories of feedback appear in the tutoring principles of Anderson (Anderson, 1993), or in existing tools supporting the stepwise construction of a solution to an exercise.

We do not try to tackle the problem of how feedback should be presented to a student. We look at the first step needed to provide feedback, namely to diagnose the problem, and relate the problem to the rules and the strategy for the exercise. We want users of our domain reasoners to determine how these findings are presented to the user. For example, we could generate a table from a strategy with the possible problems, and let teachers fill this table with the desired feedback messages.

We discuss a number of possible ways in which our strategies can be used for generating feedback automatically. This discussion is quite informal: in Chapter 5 we explain how these feedback categories relate to our services.

2.5.1 Feedback after a step

After each step performed by a student, we check whether or not this step is valid according to the strategy. For steps involving argument- and variable-value computations we have to calculate the correct values of these components, and check these values against the values supplied by the student. Such calculations are easily and naturally expressed in our framework.

Checking whether or not a step is valid amounts to checking whether or not the sequence of steps supplied by the student is a valid prefix of a sentence of the language specified by the context-free grammar corresponding to the strategy.

Hence, this is essentially a recognition problem. As soon as we detect that a student no longer follows the strategy, we have several ways to react. We can force the student to undo the last step, and let the student strictly follow the strategy. Alternatively, we can warn the student that she has made a step that is invalid according to the strategy, but let the student proceed on her own path. For instance, a student has to simplify $\frac{1}{2} + \frac{3}{4}$ using the *addFractions* strategy. When rewriting the term to $\frac{3}{6} + \frac{3}{4}$, the student can be warned that although the step is correct, it is better to do something else (namely rename the fractions such that they share a common denominator).

2.5.2 Hint

A student can ask for a hint. Given an exercise and a strategy for solving the exercise, we calculate the 'best' next step. The best next step is an element of the first set of the context-free grammar specified by the strategy. For example, suppose a student has no clue how to perform the addition in $\frac{1}{2} + \frac{2}{5}$, and he presses the hint button. The system gives the hint: "rename both fractions so that they have a common denominator". The fact that the fractions need a common denominator can be calculated from the strategy. If this does not help the student, the system can give a more specific message that suggests a specific rewrite rule, or the result of applying that rule.

2.5.3 Ready

After having performed one or more steps, a student can indicate to have finished the exercise. For example, a student submits $\frac{3}{2}$ as the final answer. Based on the strategy, a tutor can tell the student that the exercise is not yet finished. In this example, a tutor can tell exactly, based on the strategy for adding fractions, which step still needs to be performed: the improper fraction should be simplified.

2.5.4 Completion problems

A worked-out solution can be generated from a strategy, showing all the steps from the initial term to the expected answer. A worked-out solution is a presentation of a sentence that is generated by the strategy. The derivations given in Section 2.2 are examples of a worked-out solutions.

Sweller et al. (1998), based on their cognitive load theory, describe a series of guidelines to create learning materials. The basic idea is that a student profits from having example solutions played for him or her, followed by an exercise in which the student fills out some missing steps in a solution (Merriënboer et al., 1992). We can use the strategy to play a solution for a student, and we can play all but

the middle two (three, last two, etc.) steps, and ask the student to complete the exercise.

2.5.5 Progress

Given an exercise and a strategy for solving the exercise, we can determine the minimum number of steps necessary to solve the exercise, and show this information in a progress bar. Each time a student performs a correct step, the progress bar is updated.

2.6 Related work on strategies

There are other approaches (Self, 1991; Kautz and Allen, 1986) for expressing procedural skills, which also transform the recognition of student steps into a parsing problem. They too use grammars to specify how a procedure can be decomposed into steps, and regard a particular sequence of steps as a sentence that has to be parsed with respect to this grammar. However, providing feedback to students about several aspects of an exercise has been and still is an active area of research.

Explaining syntax errors has been studied in several contexts, most notably in compiler construction (Swierstra and Duponcheel, 1996), but also for intelligent tutoring systems (Horacek and Wolska, 2006). Some work has been done on trying to explain errors made by students on the level of rewrite rules (Bouwers, 2007; Hennecke, 1999; Issakova, 2007; Passier and Jeuring, 2006).

Our language is very similar to strategic programming languages such as STRATEGO (Lämmel et al., 2002; Visser et al., 1998) and ELAN (Borovanský et al., 2001). Our strategy language differs from Stratego in the sense that we, in addition to the final term, also focus on the intermediate rewrite steps. Other similar languages are used in parser combinator libraries (Hutton, 1992; Swierstra and Duponcheel, 1996), boiler-plate libraries (Lämmel and Jones, 2003), workflow applications (Plasmeijer et al., 2007), theorem proving (tacticals (Paulson, 1996)) and data-conversion libraries (Cunha and Visser, 2007).

Already around 1980, but also later, VanLehn et al. (VanLehn, 1990), and Anderson and others from the Advanced Computer Tutoring research group at Carnegie Mellon university (CMU) (Anderson, 1993; Anderson et al., 1995) worked on representing procedures or procedural networks. VanLehn et al. already noticed that 'The representation of procedures has an impact on all parts of the theory.' Anderson et al. report that the technical accomplishment was 'no mean feat'. Both VanLehn et al. and Anderson et al. deploy collections of condition-action rules, or production systems. In Mind Bugs (VanLehn, 1990), VanLehn states several assumptions about

languages for representing procedures. In Rules of the Mind (Anderson, 1993), Anderson formulates similar assumptions. Their leading argument for selecting a language for representing procedures is that it should be psychologically plausible. We think our strategy language can be viewed as a production system. But our leading argument is that it should be easy to calculate feedback based on the strategy. Moreover, it should be easy to specify, reuse, and adapt strategies. Using an EDSL similar to a language for context-free grammars for specifying a strategy simplifies calculating feedback. Furthermore, our language satisfies the assumptions about representation languages given by VanLehn, such as the presence of variables in procedures, and the possibility to define recursive procedures.

Tacticals (Paulson, 1996) and proof plans and methods (Bundy, 1988) are used to automatically prove theorems. On an abstract level, these plans and methods play the same role as strategies: we can view a strategy as a proof plan for proving that an answer is the solution to an exercise. Bundy (Bundy, 2002) discusses how proof plans are used to support interactive proving, by letting the user help the theorem prover whenever the theorem prover cannot make progress anymore, or by letting the prover explain why a particular method can or cannot be applied. As far as we found, proof plans are not used to teach theorem proving, or to recognise proving steps made by a student. Aspinall et al. (2008) introduce the tactic language Hitac that can be used to construct hierarchical proofs, so called hiproofs. To evaluate Hitac programs two semantics are given: a big step semantic that captures the intended meaning and a small step semantic that covers the details of the proof. As far as we found, the tactic language is not used to generate feedback, or to recognise proving steps made by a student. Moreover, we provide our functionality to external learning environments.

Zinn (2006) writes strategies as Prolog programs, in which rules and strategies ('task models') are intertwined. His system gives detailed feedback and supports buggy rules, with results similar to those we obtain. We believe that explicitly modelling the strategy language makes it easier to specify strategies, which in Zinn's approach have to be programmed directly in Prolog. Furthermore, the implementation of our strategy language provides us with many possibilities for giving feedback that can be reused for all other domains.

3 A STRATEGY RECOGNISER

In this chapter we discuss the design and implementation of a strategy recogniser. With this implementation we precisely describe how the different types of feedback (listed in Section 2.5) can be realised. We highlight the most important components and some design choices.

Instead of designing our own recogniser, we could reuse existing parsing libraries and tools. There are many excellent parser generators and various parser combinator libraries (Swierstra and Duponcheel, 1996; Hutton, 1992), and these are often highly optimised and efficient in both their time and space behaviour. However, the problem we are facing is quite different from other parsing applications. To start with, efficiency is not a key concern as long as we do not have to enumerate all sentences. Because we are recognising applications of rewrite or refinement rules applied by a student, the length of the input is very limited. Our experience until now is that speed poses no serious constraints on the design of the library. A second difference is that we are not building an abstract syntax tree.

The following issues are important for a strategy recogniser, but are not (sufficiently) addressed in traditional parsing libraries:

1. We are only interested in sequences of transformation rules that can be applied successively to some initial term, and this is hard to express in most libraries. Parsing approaches that start by analysing the grammar for constructing a parsing table will not work in our setting because they cannot take the current term into account. Moreover, it is often unclear how to transform strategies

with labels.

2. The ability to diagnose errors in the input highly influences the quality of the feedback services. It is not enough to detect that the input is incorrect, but we also want to know at which point the input deviates from the strategy, and what is expected at this point. Some of the more advanced parser tools have error correcting facilities, which helps diagnosing an error to some extent.

3. Exercises are solved incrementally, and therefore we do not only have to recognise full sentences, but also prefixes. We cannot use backtracking and look-ahead because we want to recognise strategies at each intermediate step. If we would use backtracking, we might give a hint that does not lead to a solution, which is undesirable in a learning environment.

4. Labels help to describe the structure of a strategy in the same way as non-terminals do in a grammar. For a good diagnosis it is vital that a recogniser knows at each intermediate step where it is in the strategy.

5. A strategy should be serialisable, for instance because we want to communicate with other online tools and environments.

In earlier attempts to design a recogniser library for strategies, we tried to reuse an existing error-correcting parser combinator library (Swierstra and Duponcheel, 1996), but failed because (some) of the reasons listed above.

3.1 Representing grammars

Because strategies are a special kind of grammars, we start by exploring a suitable representation for grammars. The datatype for grammars is based on the alternatives of the strategy language discussed in Section 2.3, except that there is no constructor for labels.

```
data Grammar a = Symbol a
               | Succeed
               | Fail
               | Grammar a  :|:  Grammar a
               | Grammar a  :*:  Grammar a
               | Grammar a  :%:  Grammar a
               | Grammar a :%>:  Grammar a
               | Atomic (Grammar a)
               | Rec Int (Grammar a)   -- recursion point
               | Var Int               -- bound by corresponding Rec
```

The type variable *a* in this definition is an abstraction for the type of symbols: for strategies, the symbols are rules, but also ENTER and EXIT steps that are associated with a label. For now we will postpone the discussion on labels in grammars.

Another design choice is how to represent recursive grammars, for which we use the constructors *Rec* and *Var*. A *Rec* binds all the *Var*s in its scope that have the same integer. We assume that all our grammars are closed, i.e., there are no free occurrences of variables. This datatype makes it easy to manipulate and analyse grammars. Alternative representations for recursion are higher-order fixed point functions, or nameless terms using De Bruijn indices.

We use constructors such as :*: and :|: for sequence and choice, respectively, instead of the combinators <*> and <|> introduced earlier. Haskell infix constructors have to start with a colon, but the real motivation is that we use <*> and <|> as *smart constructors* later.

The repetition combinator *many*, which we defined in Section 2.3.8, can be encoded with the *Grammar* datatype in the following way:

$$
\begin{aligned}
&\mathit{many} :: \mathit{Grammar}\ a \to \mathit{Grammar}\ a \\
&\mathit{many}\ \sigma = \mathit{Rec}\ 0\ (\mathit{Succeed} :|: (\sigma :*: \mathit{Var}\ 0))
\end{aligned}
$$

Later we will see that smart constructors are more convenient for writing such a combinator.

3.1.1 Empty and firsts

We use the functions *empty* and *firsts* to recognise sentences. The function *empty* tests whether the empty sentence is part of the language: $\mathit{empty}\ (\sigma) = \epsilon \in \mathcal{L}\ (\sigma)$. The function *empty* that we defined earlier in Section 2.4 distinguishes between major and minor rules, whereas the definition used here does not.

The direct translation of this specification of *empty* to a functional program, using the definition of language $\mathcal{L}$, gives a very inefficient program. Instead, we derive the following recursive function from this characterisation, by performing case analysis on strategies:

$$
\begin{aligned}
&\mathit{empty} :: \mathit{Grammar}\ a \to \mathit{Bool} \\
&\mathit{empty}\ (\mathit{Symbol}\ a) = \mathit{False} \\
&\mathit{empty}\ \mathit{Succeed} \quad\ = \mathit{True} \\
&\mathit{empty}\ \mathit{Fail} \quad\ \ = \mathit{False} \\
&\mathit{empty}\ (\sigma :|: \tau) \quad = \mathit{empty}\ \sigma \vee \mathit{empty}\ \tau \\
&\mathit{empty}\ (\sigma :*: \tau) \quad = \mathit{empty}\ \sigma \wedge \mathit{empty}\ \tau \\
&\mathit{empty}\ (\sigma :\%: \tau) \quad = \mathit{empty}\ \sigma \wedge \mathit{empty}\ \tau \\
&\mathit{empty}\ (\sigma :\%>: \tau) = \mathit{False} \\
&\mathit{empty}\ (\mathit{Atomic}\ \sigma) = \mathit{empty}\ \sigma
\end{aligned}
$$

$$\begin{aligned}
&\mathit{empty}\ (\mathit{Rec}\ i\ \sigma) &&= \mathit{empty}\ \sigma\\
&\mathit{empty}\ (\mathit{Var}\ i) &&= \mathit{False}
\end{aligned}$$

The left-interleave operator requires the language of its left-hand strategy argument to not contain the empty string. Hence, such a strategy cannot recognise the empty sentence. The definition for the pattern $\mathit{Rec}\ i\ \sigma$ may come as a surprise: it calls *empty* recursively on σ without changing the *Var*s that are bound by this *Rec*. Because there is no need to inspect recursive occurrences to determine the empty property, we define $\mathit{empty}\ (\mathit{Var}\ i)$ to be *False*.

Given some strategy σ, the function *firsts* returns every symbol that can start a sentence for σ, paired with a strategy that represents the remainder of that sentence, see Section 2.4.2. As for the function *empty*, the direct translation of this specification into a functional program is infeasible. We again derive an efficient implementation for *firsts* by performing a case analysis on strategies.

Defining *firsts* for the two interleaving cases is somewhat challenging: this is exactly where we must deal with interleaving and atomicity. More specifically, we cannot easily determine the firsts for strategy $\sigma \mathbin{\%\!>} \tau$ based on the firsts for σ and τ since that would require more information about the atomic blocks in σ and τ. For a strategy $\sigma \mathbin{\%\!>} \tau$, we split σ into an atomic part and a remainder, say $\mathit{Atomic}\ \sigma' \mathbin{<\!\star\!>} \sigma''$. After σ' we can continue with $\sigma'' \mathbin{<\!\%\!>} \tau$. Note that σ' cannot be the empty sentence. This approach is summarised by the following property, where the use of symbol a takes care of the non-empty condition:

$$(\langle a \mathbin{<\!\star\!>} \sigma\rangle \mathbin{<\!\star\!>} \tau) \mathbin{\%\!>} v = \langle a \mathbin{<\!\star\!>} \sigma\rangle \mathbin{<\!\star\!>} (\tau \mathbin{<\!\%\!>} v)$$

The function *split* transforms a strategy into triples of the form (a, x, y), which should be interpreted as $\langle a \mathbin{<\!\star\!>} x\rangle \mathbin{<\!\star\!>} y$. We define *split* for each case of the *Grammar* datatype.

$$\begin{aligned}
&\mathit{split} :: \mathit{Grammar}\ a \to [(a, \mathit{Grammar}\ a, \mathit{Grammar}\ a)]\\
&\mathit{split}\ (\mathit{Symbol}\ a) &&= [(a, \mathit{Succeed}, \mathit{Succeed})]\\
&\mathit{split}\ \mathit{Succeed} &&= [\,]\\
&\mathit{split}\ \mathit{Fail} &&= [\,]\\
&\mathit{split}\ (\sigma \mathbin{:\!|\!:} \tau) &&= \mathit{split}\ \sigma +\!\!+ \mathit{split}\ \tau\\
&\mathit{split}\ (\sigma \mathbin{:\!\star\!:} \tau) &&= [(a, x, y \mathbin{:\!\star\!:} \tau) \mid (a, x, y) \leftarrow \mathit{split}\ \sigma] +\!\!+\\
&&&\quad \mathbf{if}\ \mathit{empty}\ \sigma\ \mathbf{then}\ \mathit{split}\ \tau\ \mathbf{else}\ [\,]\\
&\mathit{split}\ (\sigma \mathbin{:\!\%\!:} \tau) &&= \mathit{split}\ (\sigma \mathbin{:\!\%\!>\!:} \tau) +\!\!+ \mathit{split}\ (\tau \mathbin{:\!\%\!>\!:} \sigma)\\
&\mathit{split}\ (\sigma \mathbin{:\!\%\!>\!:} \tau) &&= [(a, x, y \mathbin{:\!\%\!:} \tau) \mid (a, x, y) \leftarrow \mathit{split}\ \sigma]\\
&\mathit{split}\ (\mathit{Atomic}\ \sigma) &&= [(a, x \mathbin{:\!\star\!:} y, \mathit{Succeed}) \mid (a, x, y) \leftarrow \mathit{split}\ \sigma]\\
&\mathit{split}\ (\mathit{Rec}\ i\ \sigma) &&= \mathit{split}\ (\mathit{replaceVar}\ i\ (\mathit{Rec}\ i\ \sigma)\ \sigma)\\
&\mathit{split}\ (\mathit{Var}\ i) &&= \mathit{error}\ \texttt{"unbound Var"}
\end{aligned}$$

For a sequence $\sigma \mathbin{:\!\star\!:} \tau$, we determine which symbols can appear first for σ, and we change the results to reflect that τ is part of the remaining grammar. Furthermore,

if σ can be empty, then we also have to look at the symbols that can appear first for τ. For choices, we simply combine the results for both operands. If the grammar is a single symbol, then this symbol appears first, and the remaining parts are *Succeed* (we are done). To find the symbols that can appear first for *Rec i* σ, we have to look inside the body σ. All occurrences of this recursion point are replaced by the grammar itself before we call *split* again. The replacement is performed by a helper-function: *replaceVar i* σ τ replaces all free occurrences of *Var i* in τ by σ. Hence, if we encounter a *Var*, it is unbound, which we do not allow, since we assume our grammars are closed.

We briefly discuss the definitions for the constructs related to interleaving, and argue why they are correct:

- *Case* $(Atomic\ \sigma)$. Because atomicity distributes over choice, we can consider the elements of *split* σ (the recursive call) one by one. The transformation

  $$\langle\langle a \mathbin{<\star>} x\rangle \mathbin{<\star>} y\rangle \;=\; \langle a \mathbin{<\star>} (x \mathbin{<\star>} y)\rangle \mathbin{<\star>} \varepsilon$$

  is proven by first removing the inner atomic block, and basic properties of sequence.

- *Case* $(\sigma_1$ *:%:* $\sigma_2)$. Expressing this strategy in terms of left-interleave is justified by the definition of $\mathcal{L}\ (\sigma_1 \mathbin{<\%>} \sigma_2)$.

- *Case* $(\sigma_1$ *:%>:* $\sigma_2)$. Left-interleave can be distributed over the alternatives. Furthermore, $(\langle a \mathbin{<\star>} x\rangle \mathbin{<\star>} y) \mathbin{\%>} \tau = \langle a \mathbin{<\star>} x\rangle \mathbin{<\star>} (y \mathbin{<\%>} \tau)$ follows from the definition of left-interleave on sentences (with atomic blocks).

With the function *split*, we can now define the function *firsts*, which is needed for the generation of most of the feedback types:

```
firsts :: Grammar a → [(a, Grammar a)]
firsts σ = [(a, x :*: y) | (a, x, y) ← split σ]
```

In Section 2.3.12 we discuss restrictions imposed on strategies. It should now be clear from the definition of *firsts* why left-recursion is problematic. Applying the *split* function to a left-recursive strategy will cause the split function to loop without returning any values, hence the *firsts* function will loop as well. For example, consider the *many* combinator. A strategy writer has to use this combinator with great care to avoid constructing a left-recursive grammar: if grammar σ accepts the empty sentence, then running the grammar *many* σ can result in non-termination. The problem with left recursion can be partially circumvented by limiting the number of recursion points (*Rec*s and *Var*s) that are unfolded in the definition of *split* $(Rec\ i\ \sigma)$. When is the limit reached, the case for the *Rec* constructor in the *split* function stops unfolding and returns the empty list.

3.2 Dealing with labels

We use label information to trace where we are in a strategy, by inserting ENTER and EXIT steps for each labelled substrategy. These labels enable us to attach specialised feedback messages to certain locations in the strategy.

Labels are not added to the *Grammar* datatype, but on the level of rules, for which we introduce the following datatype:

```
type Rule a   = a → [a]
data Step l a = ENTER l | Step (Rule a) | EXIT l
```

The type argument *l* represents the type of information associated with each label. For our strategies we assume that this information is a string. The type *Rule* is parametrised by the type of values to which the rule can be applied. The type declaration for *Rule* is a simplification of the implementation, which is more involved. In the implementation we maintain meta information about the rule, such as an identifier and whether or not the rule is minor. With the *Step* datatype, we can now specify a type for strategies:

```
type LabelInfo   = String
data Strategy a = S {unS :: Grammar (Step LabelInfo a)}
```

The *Strategy* datatype wraps a grammar, where the symbols of this grammar are steps. The following function helps to construct a strategy out of a single step:

```
fromStep :: Step LabelInfo a → Strategy a
fromStep = S ∘ Symbol
```

Wrapping strategies quickly becomes cumbersome when defining functions over strategies. We therefore introduce a type class for type constructors that can be converted into a *Strategy*:

```
class IsStrategy f where
  toStrategy :: f a → Strategy a
instance IsStrategy Rule where
  toStrategy = fromStep ∘ Step
instance IsStrategy Strategy where
  toStrategy = id
```

In addition to the *Strategy* datatype, we define the *LabeledStrategy* type for strategies that have a label. A labelled strategy can be turned into a (normal) strategy by surrounding its strategy with ENTER and EXIT steps.

```
data LabeledStrategy a = Label {labelInfo :: LabelInfo, unlabel :: Strategy a}
instance IsStrategy LabeledStrategy where
  toStrategy (Label a σ) = fromStep (ENTER a) <*> σ <*> fromStep (EXIT a)
```

In the next section we present smart constructors for strategies, including the strategy combinator <*> for sequences, which is used twice in the instance declaration for *LabeledStrategy*.

3.3 Smart constructors

A smart constructor is a function that in addition to constructing a value performs some checks, simplifications, or conversions. We use smart constructors for simplifying strategies. We introduce a smart constructor for every alternative of the strategy language given in Section 2.3. Definitions for ε and δ are straightforward, and are given for consistency:

```
ε, δ :: Strategy a
ε = S Succeed
δ = S Fail
```

The general approach is that we use the *IsStrategy* type class to automatically turn the subcomponents of a combinator into a strategy. As a result, we do not need a strategy constructor for rules, because *Rule* was made an instance of the *IsStrategy* type class. The context will turn a rule into a strategy, if required. This approach is illustrated by the definition of the *label* constructor, which is overloaded in its second argument:

```
label :: IsStrategy f ⇒ LabelInfo → f a → LabeledStrategy a
label σ = Label σ ∘ toStrategy
```

All other constructors return a value of type *Strategy*, and overload their strategy arguments. We define helper-functions for lifting unary and binary constructors (*lift1* and *lift2*, respectively). These lift functions turn a function that works on the *Grammar* datatype into an overloaded function that returns a strategy.

```
lift1    :: IsStrategy f
         ⇒ (Grammar a → Grammar a) → f a → Strategy a
lift1 op = S ∘ op ∘ unS ∘ toStrategy
lift2    :: (IsStrategy f, IsStrategy g)
         ⇒ (Grammar a → Grammar a → Grammar a) → f a → g a → Strategy a
lift2 op = lift1 ∘ op ∘ unS ∘ toStrategy
```

For choices, we remove occurrences of *Fail*, and we associate the alternatives to the right.

```
(<|>) :: (IsStrategy f, IsStrategy g) ⇒ f a → g a → Strategy a
(<|>) = lift2 op
  where
    op :: Grammar a → Grammar a → Grammar a
    op Fail      τ     = τ
    op σ         Fail  = σ
    op (σ :|: τ) v     = σ `op` (τ `op` v)
    op σ         τ     = σ :|: τ
```

The smart constructor <⋆> for sequences removes the unit element *Succeed*, and propagates the absorbing element *Fail*.

```
(<⋆>) :: (IsStrategy f, IsStrategy g) ⇒ f a → g a → Strategy a
(<⋆>) = lift2 op
  where
    op :: Grammar a → Grammar a → Grammar a
    op Succeed  τ          = τ
    op σ        Succeed    = σ
    op Fail     _          = Fail
    op _        Fail       = Fail
    op (σ :⋆: τ) v         = σ `op` (τ `op` v)
    op σ        τ          = σ :⋆: τ
```

The binary combinators for interleaving, <%> and %>, are defined in a similar fashion. The smart constructor *atomic*, which was denoted by $\langle\cdot\rangle$ in Section 2.3, takes only one argument. It is defined in the following way:

```
atomic :: IsStrategy f ⇒ f a → Strategy a
atomic = lift1 op
  where
    op :: Grammar a → Grammar a
    op (Symbol a)  = Symbol a
    op Succeed     = Succeed
    op Fail        = Fail
    op (Atomic σ)  = op σ
    op (σ :|: τ)   = op σ :|: op τ
    op σ           = Atomic σ
```

This definition is based on several properties of atomicity, such as idempotence and distributivity over choice.

The last combinator we present is for recursion. Internally we use numbered *Rec*s and *Var*s in our *Grammar* datatype, but for the strategy writer it is more convenient to write the recursion as a fixed-point, without worrying about numbering variables. For this reason we do not define direct counterparts for the *Rec* and *Var* constructors, but only the higher-order function *fix*. This combinator is defined as follows:

```
fix :: (Strategy a → Strategy a) → Strategy a
fix f = lift1 (Rec i) (make i)
where
  make = f ∘ S ∘ Var
  is   = usedNumbers (unS (make 0))
  i    = if null is then 0 else maximum is + 1
```

The trick is that function f is applied twice. First, we pass f a strategy with the grammar *Var* 0, and we inspect which numbers are used, using the function *usedNumbers* (variable *is* of type $[Int]$). We omit the definition of *usedNumbers*. Based on this information, we now determine the next number to use (variable *i*). We apply f for the second time using grammar *Var i*, and bind these *Var*s to the top-level *Rec*. Note that this approach does not work for fixed-point functions that perform case analysis on their arguments, since the bound variables of f are replaced by a *Var*. We have not encountered any problems with this restriction.

We can now define the repetition combinator *many* in terms of the smart constructors. Observe that *many*'s argument is also overloaded because of the smart constructors.

```
many :: IsStrategy f ⇒ f a → Strategy a
many σ = fix $ λx → ε <|> (σ <*> x)
```

3.4 Running a strategy

A strategy is a grammar over rewrite rules and ENTER and EXIT steps for labels. We can *run* a strategy, that is, we can apply the rules to an initial term in the order as specified in the strategy. The result of running a strategy is a term that the strategy considers to be a solution. We first define a type class to overload the function that we apply to terms. The type class *Apply* has a method *apply* that returns a list of results, obtained by applying a rule or a strategy to a value. To apply a rule to a value, we give an instance declaration of *Apply* for the *Step* datatype, where the ENTER and EXIT steps return a singleton list with the current term, i.e., they do not have an effect.

```
class Apply f where
  apply :: f a → a → [a]
```

```
instance Apply Rule where
  apply r x = r x
instance Apply (Step l) where
  apply (Step r) = apply r
  apply _        = return
```

We can now give an implementation for running grammars with symbols in the *Apply* type class. The implementation uses the functions *empty* and *firsts*.

```
run :: Apply f ⇒ Grammar (f a) → a → [a]
run σ a = [a | empty σ] ++ [c | (r, τ) ← firsts σ, b ← apply r a, c ← run τ b]
```

The list of results returned by *run* consists of two parts: the first part is the singleton list containing the term a, provided *empty* σ holds. The second part takes care of the alternatives that start with symbols that appear first in σ. Let r be one of the symbols that can appear first in strategy σ. We apply r to the current term a, yielding a new term b. We run the remainder of the strategy (that is, τ) on this new term.

We make *Strategy* and *LabeledStrategy* instances of class *Apply* using the *run* function.

```
instance Apply Strategy where
  apply = run ∘ unS
instance Apply LabeledStrategy where
  apply = apply ∘ toStrategy
```

The function *run* may produce an infinite list. In most cases, however, we are only interested in a single result, and rely on lazy evaluation to only calculate the first element of the list. The term obtained for an empty strategy is put at the front to return terms that are obtained by applying a few rewrite rules early.

3.5 Tracing a strategy

The *run* function defined in the previous section ignores the labels. However, if we want to recognise (intermediate) terms submitted by a student, and give feedback if the answer is incorrect, then labels become important. We extend the definition of *run* to keep a trace of the steps that have been applied:

```
runTrace :: Apply f ⇒ Grammar (f a) → a → [(a, [f a])]
runTrace σ a =
  [(a, [])       | empty σ] ++
  [(c, (r : rs)) | (r, τ) ← firsts σ, b ← apply r a, (c, rs) ← runTrace τ b]
```

In case of a strategy, we can thus obtain the list of ENTER and EXIT steps seen so far. We illustrate this by means of an example.

We return to the strategy for adding two fractions (*addFractions*, defined in Chapter 2). Suppose that we run this strategy on the term $\frac{2}{5} + \frac{2}{3}$. This would give us the following derivation:

$$\frac{2}{5} + \frac{2}{3} \;=\; \frac{6}{15} + \frac{2}{3} \;=\; \frac{6}{15} + \frac{10}{15} \;=\; \frac{16}{15} \;=\; 1 + \frac{1}{15}$$

The final answer, $1 + \frac{1}{15}$, is indeed what we would expect. This result is returned twice because the strategy does not specify which of the fractions should be renamed first, and as a result we get two different derivations. It is informative to see the intermediate steps returned by function *runTrace*.

```
[ Enter_ℓ0,      Enter_ℓ1,      Step LCD,      Exit_ℓ1,      Enter_ℓ2
, Step Down,     Step RENAME,   Step Up,       Step Down,    Step Right
, Step RENAME,   Step Up,       Step AppCheck, Exit_ℓ2,      Enter_ℓ3
, Step ADD,      Exit_ℓ3,       Enter_ℓ4,      Step SIMPL,   Exit_ℓ4
, Exit_ℓ0 ]
```

The list has twenty steps, but only four correspond to actual steps appearing in the derivation showed to the student: the rules of those steps are underlined. The other rules are minor: the navigation rules UP, RIGHT and DOWN are introduced by the *somewhere* combinator, and APPCHECK comes from the use of *repeat*. Also observe that each ENTER step has a matching EXIT step. A label may be visited multiple times by a strategy.

We determine at each point in the derivation where we are in the strategy by enumerating the ENTER steps for which the corresponding EXIT steps have not been performed yet. Based on this information we can fine-tune the feedback messages that are reported when a student submits an incorrect answer, or when she asks for a hint on how to continue.

4 Exercises

An exercise can be regarded as a task that a student needs to carry out, possibly using an ITS. According to VanLehn (2006), a task is a multi-minute action that consists of multiple steps. A student performs these tasks to increase understanding and improve skills. Our domain reasoners reuse the term exercise for a broader concept. In addition to a task description, an exercise contains many more components, such as the set of allowed rules, the strategy for solving the task, and a task generator. An exercise groups together all components that are necessary to present and reason about a task, such as giving guidance and feedback.

Although we can derive many types of feedback from a strategy, as shown in the previous chapter, there are other components necessary to deliver the feedback when solving an exercise. For example, when a student makes a syntax error or takes an unknown step, we would like to give detailed feedback. This feedback cannot be generated from a strategy alone. We need additional components, such as a parser to give feedback about syntax errors.

An exercise contains all exercise-specific functionality. The most important component of an exercise is its strategy. Additional rewrite rules, that is rules that are not used in the strategy, can be added to an exercise to help detect possible deviations from the strategy. We not only specify proper rewrite rules, but also *buggy rules*. A buggy rule captures a common misconception. If we detect an application of a buggy rule, we report this to the user. We also need predicates for checking whether or not an expression is a suitable starting expression that can be

solved by the strategy, and whether or not an expression is in solved form. These two predicates can be defined as views. A view (Heeren and Jeuring, 2009) defines an equivalence relation by choosing a canonical form of mathematical expressions. We use this equivalence relation for diagnosing intermediate answers. We use this relation to compare a student submission with the preceding expression. We also need a similarity relation, which checks whether two terms are similar enough to be considered the same. A similarity relation is possibly more liberal than syntactic equality. The similarity relation can also be defined as a view, and is used to transform intermediate terms produced by a strategy to their canonical forms. What remains to be supplied for an exercise is its metadata, such as an identifier that can serve as a reference, and a short description. For certain domains it is convenient to have a dedicated parser and pretty-printer for the terms. Although not of primary importance, it can be convenient to have a randomised expression generator for the exercise. The last component of an exercise is a function that returns an ordering on rules. We define an exercise as follows.

Definition 4.1. *An exercise consists of an identification code, a strategy, a rule set, a buggy rule set, an equivalence relation, a similarity relation, a predicate suitable, a predicate finished, a parser and pretty-printer, an expression generator, and a rule ordering function.*

The following shows a concrete example of an exercise definition for an exercise in the domain of arithmetic with fractions:

$$
\begin{array}{ll}
\mathit{addFractionsExercise} = & \\
\quad (\texttt{addFracEx} & \text{-- ID code} \\
\quad , \mathit{addFractions} & \text{-- Strategy} \\
\quad , \{ \textsc{reciPow} : \ \frac{a}{b} = a \cdot b^{-1} \} & \text{-- Rule set} \\
\quad , \{ \textsc{b1} : \ \frac{a}{b} + \frac{c}{d} \ \neq \ (\frac{a+c}{b+d}), \textsc{b2}, \textsc{b3} \} & \text{-- Buggy rules} \\
\quad , \mathit{eqExpr}, \mathit{simExpr}, \mathit{suitableExpr}, \mathit{finishedExpr} & \text{-- Checks} \\
\quad , \mathit{parseExpr}, \mathit{printExpr} & \text{-- Parser and pretty printer} \\
\quad , \mathit{genExpr} & \text{-- Generator} \\
\quad , \textsc{zeroMul} < \textsc{unitMul} < \textsc{unitAdd} < \textsc{unitSub}) & \text{-- Rule ordering}
\end{array}
$$

We will describe every component of this example exercise specification in detail in the remainder of this chapter.

Many of the components specified in an exercise, such as the rewrite rules, operate on the abstract syntax of the domain of the exercise. In our running example this is the domain of arithmetic. The following Haskell data type declaration is used for the abstract syntax of the domain of simple arithmetic expressions:

```
data Expr = Var String     | Nat Integer    | Negate Expr
          | Expr :+: Expr | Expr :-: Expr | Expr :*: Expr
```

```
                | Expr :^: Expr  | Expr :/: Expr
    deriving Eq
```

This definition is based on the grammar for arithmetic expressions defined in Section 2.2. As in the grammar, the constructor for constants (*Nat*) should only be used for positive natural numbers. Constructor *Negate* negates an expression. Making negation explicit allows to specify rewrite rules more conveniently, e.g., using pattern matching. We have made the *Expr* datatype an instance of the *Num* and *Fractional* classes. We do not show the code for these instances.

We need a parser to convert the concrete syntax, i.e., the text a student writes, to this abstract syntax. There are many good parser libraries for Haskell, such as libraries developed by Swierstra and Duponcheel (1996), and Leijen and Meijer (2001). Our parser has the following type:

```
parseExpr :: String → Either String Expr
```

If the parser can successfully parse the student input it returns the abstract syntax, otherwise an error message is returned. This error message is reported to the student. The inverse of parsing is pretty printing (Oppen, 1980; Hughes, 1995), which converts the abstract syntax to a string. For external tools, however, exchanging messages using an abstract syntax (as opposed to concrete syntax), such as OpenMath objects (Society, 2006) for mathematical domains, is the preferred way of communication, avoiding the need for a parser and pretty-printer.

The remainder of this chapter discusses the remaining components that are collected in an exercise, and describes the details of the *addFractionsExercise* exercise example given above.

4.1 Strategy and rules

In the previous chapter we introduce rules and strategies. We now show how rules and strategies are specified in an exercise.

4.1.1 Specifying rules

Rewrite rules specify how terms can be manipulated, and are often given explicitly in textbooks. Well-known examples are rewriting $0 \cdot x$ into 0, associativity of addition, and the DeMorgan rules for rewriting logic expressions. These rules are the steps a student can take, and constitute the steps in worked-out solutions. A rewrite rule is sound if it preserves the semantics of the term, i.e., the original term is equivalent to the rewritten term, under the condition that the terms are well-defined. Soundness of rules can be checked with respect to some semantic

interpretation of an expression. A semantic interpretation can be context-specific (e.g., $x^2 = -3$ gives no solutions for x in $\mathbb{R}$). In Section 4.2 we introduce semantic checks, and we will give an example of a semantic interpretation.

Rewrite rules are atomic actions that are implemented in code. Clearly, this gives the implementer of a rule the full power of the underlying programming language. For example, the rewrite rule for adding fractions (ADD) is defined as follows:

```
ADD :: Rule Expr
ADD = ruleList "add" f
  where
    f ((a :/: b) :+: (c :/: d)) | b == d = [(a + c) :/: b]
    f _                                  = []
```

To represent a rewrite rule we use the abstract datatype *Rule a*, which is polymorphic in the type of the abstract syntax of the domain. The definition for the rewrite rule ADD pattern matches on the construction of an addition of two fractions, and returns a singleton list with the result of the addition. Recall that we allow rewrite rules to yield multiple results. If an expression is not the sum of two fractions, or the denominators are different, it cannot be pattern matched, in which case the ADD rewrite rule returns the empty list. The latter case signals that the rewrite rule was not applicable. The function *ruleList* :: *String* $\rightarrow (a \rightarrow [a]) \rightarrow$ *Rule a* transforms the function f into a rule.

Alternatively, we can specify rules with a left-hand side and a right-hand side, and rely on unification and substitution of terms to do the transformation (van Noort et al., 2010). This corresponds to the definition of rewrite rules in term rewrite systems (Baader and Nipkow, 1997). An example of this alternative definition is given in the following code:

```
ADD' :: Rule Expr
ADD' = rewriteRule "add"
  (λa b c → (a :/: b) :+: (c :/: b) :~> (a + c) :/: b)
```

This definition is more elegant than the previous one. The infix operator $(:\sim>)$ builds a rule specification based on the left-hand and right-hand side. The function *rewriteRule* takes a function that returns a rule specification as argument, and constructs a rewrite rule that can be used in a strategy. We do not need the guard that checks whether the denominators are the same. Based on this specification, the unification procedure makes sure that a rule is only applied to fractions with the same denominator. This way of defining rewrite rules is, however, less powerful than using a regular function.

Besides the rules that appear in the strategy for an exercise, we can specify rules for the purpose of being recognised. When a student deviates from the strategy

we use the semantic interpretation to check whether the submitted expression is equivalent. If so, we try to match one of the additional specified rules, and give a detailed feedback message. In this message we report that although a correct rule has been successfully applied, the rule is not accepted by the strategy. The example exercise contains a rule set with one additional rule, namely RECIPOW.

4.1.2 Buggy rules

In addition to the rules in a strategy and the extra rewrite rules, we can formulate *buggy rules*. These rules capture common misconceptions, such as the following unsound variants of rewriting arithmetic expressions:

$$\text{B1:} \quad \frac{a}{b} + \frac{c}{d} \neq \frac{a+c}{b+d}$$

$$\text{B2:} \quad a \cdot \frac{b}{c} \neq \frac{a \cdot b}{a \cdot c}$$

$$\text{B3:} \quad a + \frac{b}{c} \neq \frac{a+b}{c}$$

Buggy rules make it possible to detect mistakes that are made often. If the system detects that such a rule is applied, it will present a specialised feedback message. For example, suppose a student submits $\frac{3}{7}$ as an intermediate solution to the exercise $\frac{1}{2} + \frac{2}{5}$. Because the terms are not equivalent, the buggy rules are tried, and in this case rule *B1* matches. A special message associated with this rule (for example, "you added the denominators as well as the numerators; you should rename the fractions so that they have the same denominator and then add only the numerators") is reported to the student. In the example specification of an exercise the buggy rule set consists of buggy rules *B1*, *B2*, and *B3*. Note that these buggy rules should not appear in a strategy, since that would invalidate the strategy.

4.1.3 Rule ordering

The choice combinator (<|>) introduces a possible source of non-determinism. If the strategy offers multiple options to solve an exercise, we use the rule ordering to make a deterministic choice. We have introduced a rule ordering (<) in Section 2.4.3, which shows how we prevent non-deterministic behaviour when generating feedback. We have chosen to specify the following rule ordering in the example exercise definition: ZEROMUL < UNITMUL < UNITADD < UNITSUB. This means that the rewrite rule ZEROMUL has precedence over all the other rules, and UNITMUL has precedence over UNITADD and UNITSUB etc. If the ordering of a rule is not specified, then the rules

are ordered alphabetically by name. The rules that are specified in the ordering have a higher priority then unspecified rules.

4.1.4 Specifying strategies

Simple problems may be solved by applying a set of rules exhaustively, but this is generally not the case. A rewrite strategy guides the process of applying rewrite rules to solve a particular class of problems. For example, the second strategy from Subsection 2.3.10 describes how to solve the problem of adding fractions in simple arithmetic expressions. This strategy is included in the example exercise specification.

Recipes for solving a certain type of problem can be found in textbooks, but they are often not precise enough for the purpose of building a domain reasoner. Given a collection of worked-out solutions by an expert, one can try to infer the strategy that is used. Rewrite strategies are built from rewrite rules, using strategy combinators. The combinators are described extensively in the previous chapter. From a strategy description, multiple derivations may be generated or recognised.

Since strategies only structure the order in which rewrite rules are applied, soundness of a derivation follows directly from the soundness of the rules involved. A derivation is sound if all intermediate terms in the derivation are equivalent. Recall that a strategy not only prescribes which rule to apply, but possibly also where (that is, to which subterm). Also, strategies are designed with a specific goal in mind. The strategy for arithmetic with fractions, for instance, is expected to rewrite an expression until we are left with a single fraction (or mixed number). The solved form that a strategy is supposed to reach is the strategy's post-condition. This post-condition is specified in the *finished* predicate in an exercise. Similarly, a strategy may have certain assumptions about the starting term (e.g., the expression must at least contain two fractions, or only a single variable is involved), which is its pre-condition. The predicate *suitable* checks this pre-condition.

Buggy strategies. The idea of buggy rules can easily be extended to buggy strategies. A buggy strategy corresponds to a common procedural mistake. Applying a buggy rule results in an expression with a different semantics from the previous expression. Applying a rule from a buggy strategy results in an equivalent expression, but following a wrong strategy. If a step supplied by a student is invalid with respect to the strategy specified, but can be explained by a buggy strategy for the problem, we can give the error message belonging to the buggy strategy. This amounts to parsing, not just with respect to the correct strategy, but also with respect to known buggy strategies. Although we have explored the idea of buggy strategies, they are currently not available in the implementation of our strategy recogniser.

4.1.5 Views

Canonical forms and notational conventions are an integral part of mathematics. Examples of conventions in writing a polynomial are the order of its terms (sorted by the degree of the term), and writing the coefficient in front of the variable. Such conventions also play a role when discussing equations of the form $ax^2 + bx + c = 0$: it is likely that $2x^2 + 3x + 2 = 0$ is considered an instance of the form, and the atypical expression $-(-3)x + 1 \cdot 2x^2 = -2$ not. For example, the atypical expression uses a double negation instead of addition. These conventions allow for elegant specification of rewrite rules, but limit the applicability. It is very hard, if not impossible, to check whether a term is of a canonical form.

Canonical forms and notational conventions can be captured in a view (Heeren and Jeuring, 2009), which consists of a partial function for matching, and a (complete) function for building. Views are based on the views proposed by Wadler (1987). Matching may result in a value of a different type, such as the pair $(-3, 5)$ for the expression $-\frac{3}{5}$. In this example, the interpretation of the pair would be a fraction (or a division) of both parts. This interpretation makes it easy to inspect the individual parts. Having a value of a different type after matching can be useful when specifying a rewrite rule: the pair $(-3, 5)$, for instance, witnesses that a fraction was recognised at top-level. Building after matching gives a canonical form, and this composed operation should therefore be idempotent. The composition of both components of a view is assumed to preserve a term's semantics.

Primitive views can be composed into compound views, in two different ways. Firstly, a views fits the arrow interface (Hughes, 2000; Paterson, 2003), and its bidirectional variant. The combinators of this interface can be used for combining views, such as using views in succession. Secondly, a view can be parameterised with another view. Consider a view for recognising expressions of the form $ax + b$, returning a pair of expressions for a and b. Another view can then be used for these two parts (e.g., a view for rational numbers). Essentially, this pattern of usage corresponds to having higher-order views.

A view pairs a *match* and *build* function. For each view we assume that the two functions define a canonical form. A canonical form (or normal form) of an expression is a standard way of (re)presenting that expression. The function *canonical* returns the canonical form of an element under a given view:

$$
\begin{aligned}
&\mathit{canonical} :: (a \to \mathit{Maybe}\ b, b \to a) \to a \to \mathit{Maybe}\ a \\
&\mathit{canonical}\ (\mathit{match}, \mathit{build})\ a = \mathbf{do}\ b \leftarrow \mathit{match}\ a \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Just}\ (\mathit{build}\ b)
\end{aligned}
$$

We apply the *match* function of the view on an element, and on a successful match, we use the *build* function to return to the original domain. For convenience, we also define a simplification function, which is the identity function when matching fails:

$$\begin{aligned}&\mathit{simplify} :: (a \to \mathit{Maybe}\ b, b \to a) \to a \to a\\&\mathit{simplify}\ \mathit{view}\ a = \mathit{fromMaybe}\ a\ (\mathit{canonical}\ \mathit{view}\ a)\end{aligned}$$

The following properties of the *simplify* function should hold for all views, establishing a property for match and build pairs.

Property 1 (Idempotence)**.** *For every view v, simplify v is an idempotent function. If this is not the case, we say that view v is improper.*

Property 2 (Soundness)**.** *Simplification with a view v should preserve the semantics of a term. Let a be some element in the domain of view v, and let sem denote the semantics of that domain. Then sem a = sem* (*simplify v a*)*.*

Because each proper view defines a canonical form, we can use it to define an equivalence relation. Two elements can be tested for equivalence under a view by comparing their canonical forms.

We use views in various ways:

- as a rewrite rule, reducing a term to its canonical form (if possible);
- as an equivalence relation, comparing the canonical forms of two terms;
- as a predicate, checking whether a term has a canonical form;
- as a way to limit the set of necessary rewrite rules.

In the rest of this subsection we give an example of the last usage, and describe some common functions on views.

Consider the exercise of adding two fractions again. The first step in our example strategy for adding two fractions is to let the fractions have the same denominator, and for this purpose we compute the least common denominator. Given an expression of type *Expr*, the following Haskell function returns the least common denominator of two fractions:

$$\begin{aligned}&\textsc{lcd} :: \mathit{Expr} \to \mathit{Maybe}\ \mathit{Integer}\\&\textsc{lcd}\ ((a :/: \mathit{Nat}\ b) :+: (c :/: \mathit{Nat}\ d)) = \mathit{Just}\ (\mathit{lcm}\ b\ d)\\&\textsc{lcd}\ \_ = \mathit{Nothing}\end{aligned}$$

where *lcm* is a predefined function which calculates the lowest common multiple of two integers. The function LCD is partial, which is reflected by the *Maybe* type constructor. The function only works for sums of fractions: for all other values, the function cannot compute an least common denominator and returns *Nothing*. In fact, our definition of LCD is unsuitable for our *Expr* data type:

- Suppose we also want to use LCD when subtracting one fraction from another, e.g., $\frac{2}{3} - \frac{1}{4}$. This requires an extra case for our definition, in which we match on the constructor :−: at top-level.
- What if the first fraction is negative, as in $-\frac{1}{4} + \frac{2}{3}$? In combination with support for subtraction, this requires a substantial number of new cases.
- The denominator can also be negative ($\frac{1}{-4} + \frac{2}{3}$), leading to even more combinations that have to be considered.

In this context, pattern matching is cumbersome because the number of cases grows rapidly. Instead, we use views to gain flexibility, without obscuring LCD's definition. A view allows us to represent a collection of expressions by means of expressions of a particular canonical form.

At top-level, the function LCD is expecting an addition, and we can apply some algebraic laws to put an expression into the expected form (if possible). The function *matchPlus* tries to match an addition at top-level, and uses basic laws for negation to do so. If it succeeds, it returns a pair containing the operands of the addition.

```
matchPlus :: Expr → Maybe (Expr, Expr)
matchPlus (a :+: b)    = Just (a, b)
matchPlus (a :−: b)    = Just (a, Negate b)    -- push negation inside
matchPlus (Negate a) = do (x, y) ← matchPlus a
                          Just (Negate x, Negate y)    -- distribute negation
matchPlus _            = Nothing
```

In the case for negation, we call the function recursively on the negated term. If the call succeeds with a pair (x, y), both operands are negated. In the same fashion, we introduce a function to match a fraction. Here, we only push negations into the numerator.

```
matchDiv :: Expr → Maybe (Expr, Expr)
matchDiv (a :/: b)    = Just (a, b)
matchDiv (Negate a) = do (x, y) ← matchDiv a
                         Just (Negate x, y)    -- push negation inside
matchDiv _            = Nothing
```

We explicitly mark whether or not an expression is negative, by means of the *Negate* constructor. The *Nat* constructor should only be used for positive constants, otherwise we could construct negative numbers in two ways (*Negate* (*Nat a*) or *Nat* $(-a)$). If we want to match an integer value, pattern matching again becomes a bit awkward. The third match-function alleviates this problem. This function matches a natural number preceded by one or more negations, and returns an integer value.

```
matchNumber :: Expr → Maybe Integer
matchNumber (Nat n)    = Just n
matchNumber (Negate e) = do c ← matchNumber e
                            Just (−c)
matchNumber _          = Nothing
```

Note that $(-c)$ is the primitive negation operation applied to integer c.

With the helper-functions for matching expressions, we can define LCD. With some "plumbing" in the *Maybe* monad, this is not too difficult. However, there is a more elegant way. The type of match functions precisely fit the *Arrow* interface (Paterson, 2003), which is a general interface for computation. In our case, we model partiality by introducing the *Maybe* monad, which turns match functions into a Kleisli arrow: an arrow of type $a \to m\ b$ for some monad m. We can use the combinators from the *Arrow* type class to compose match functions. With the arrow combinators, we define *matchTwoFractions*, which views an expression as the sum of two fractions with constants in the denominators.

```
matchTwoFractions :: Expr → Maybe ((Expr, Integer), (Expr, Integer))
matchTwoFractions = runKleisli $
    Kleisli matchPlus >>> (matchFraction *** matchFraction)
  where
    matchFraction = Kleisli matchDiv >>> second (Kleisli matchNumber)
```

We can use this match function to give an improved definition for LCD:

```
LCD' :: Expr → Maybe Integer
LCD' e = do ((a, b), (c, d)) ← matchTwoFractions e
            Just (lcm b d)
```

So far, we have only looked at the matching function from a view. With each partial function from a to b, we associate a *build* function, which returns a value in the original domain. For example, the build function for a fraction can be defined straightforwardly:

```
build :: (Expr, Expr) → Expr
build (a, b) = a :/: b
```

4.2 Syntactic and semantic checks

If a student deviates from a strategy we can no longer use the strategy as a basis for generating feedback. The only fact we can derive from a strategy is that a student

took a step that is not expected. In this case the feedback that can be derived from a strategy is limited. We distinguish two cases when a student leaves the path that is specified by the strategy: either the student applies a correct, but unknown rewrite rule, or the student makes a mistake. In the latter case the rewritten expression is no longer equivalent to the previous expression. To determine this fact we need a semantic interpretation of the domain of the exercise. This interpretation can be used to define an equivalence function on expressions. In our running example we need a function that calculates whether or not two simple arithmetic expressions are equal. We define this function in terms of a view:

```
matchExpr :: Expr → Maybe Rational
matchExpr = calcExpr
  where
    calcExpr e =
      case e of
        Nat n     → return (n % 1)
        Negate e → liftM negate (calcExpr e)
        e1 :+: e2 → liftM2 (+) (calcExpr e1) (calcExpr e2)
        e1 :-: e2 → liftM2 (-) (calcExpr e1) (calcExpr e2)
        e1 :*: e2  → liftM2 (*) (calcExpr e1) (calcExpr e2)
        e1 :/: e2  → do x ← calcExpr e1
                        y ← calcExpr e2
                        guard $ y /= 0
                        return (x / y)
        e1 :^: e2 → do x ← calcExpr e1
                       y ← calcExpr e2
                       guard $ y > 0 && denominator y == 1
                       return (x^(numerator y))
        _ → Nothing
buildExpr :: Rational → Expr
buildExpr r
   | denominator r == 1 = Nat (numerator r)
   | numerator   r < 0  = Negate (Nat (abs (numerator r)) :/: Nat (denominator r))
   | otherwise          = Nat (numerator r) :/: Nat (denominator r)
normExpr :: Expr → Expr
normExpr = simplify (matchExpr, buildExpr)
```

We use these functions to define the equality function on simple arithmetic expressions, where == stands for the derived equality on *Expr*:

```
eqExpr :: Expr → Expr → Bool
eqExpr e1 e2 = normExpr e1 == normExpr e2
```

The *eqExpr* function normalises both expressions, and compares the normalised expressions with the derived equality function.

The equivalence function is used in the feedback generation. If a student deviates from the strategy we can report that an unknown, but correct step has been taken, or that the student has made a mistake. In the first case an ITS that uses our feedback functionality may decide to let the student carry on. An alternative is to force the student to stay on the path described by the strategy.

An equivalence function is of primary importance when generating feedback. An ITS should be able to determine whether or not the student has taken a correct step or made a mistake, and report this to the student. But what if the student (repeatedly) submits the same expression? The equivalence function is of no use in such a situation, because obviously the expression remains equivalent. Another situation in which the equivalence function does not provide relevant information, is when a student has taken a step that is considered insignificant by a tutor. An insignificant step is a correct rewrite step, but the application of that step does not bring the expression closer to the final answer. For example, suppose a student uses the commutativity property of $(+)$ to rewrite $2 + 3$ to $3 + 2$. To detect this kind of steps we specify a similarity function. The most straightforward definition for similarity is to define it in terms of syntactic equivalence. However, as the example makes clear, sometimes it is necessary to define a function that is more liberal than syntactic equivalence. In our running exercise example we use the derived equality function on the abstract syntax:

```
simExpr :: Expr → Expr → Bool
simExpr e1 e2 = e1 == e2
```

finishedExpr is a semantic function that determines if a student submission can be accepted as a final answer. It is defined as follows:

```
finishedExpr :: Expr → Bool
finishedExpr expr = isProperFrac expr' || isNat expr'
  where
    expr' = rewriteNeg expr
    isProperFrac expr =
      case expr of
        Nat c :+: (Nat a :/: Nat b) | a < b → True
        Nat a :/: Nat b             | a < b → True
        _                                   → False
    isNat expr =
      case expr of
        Nat _ → True
```

```
    _          → False
rewriteNeg expr =
  case expr of
    Negate a :-: b              → a :+: b
    Negate (Nat a) :/: Nat b    → Nat a :/: Nat b
    Nat a :/: Negate (Nat b)    → Nat a :/: Nat b
    Negate (Negate _)           → expr
    Negate a                    → a
    _                           → expr
```

The function *finishedExpr* pattern matches on the forms that are allowed as a final answer. An expression is finished if it is a mixed number, a proper fraction, or a constant. Some forms of negative expressions are allowed as a final answer. For example, the expression $\frac{1}{-2}$ is accepted as a final answer, whereas the expression $\frac{-1}{-2}$ is not.

The semantic and syntactic functions defined above are independent from the rules and the strategy. We can use these functions to test certain properties of the rules and the strategy. For example, every rewrite (and therefore also strategy) should be semantics preserving. After applying a rewrite rule to an expression the result should be equivalent to the original expression. In Section 4.3 we give a list of properties of exercises.

Expression generator. An exercise contains an expression generator. A generated expression should be solvable with the specified strategy. The generator that we have implemented is based on the QuickCheck (Claessen and Hughes, 2000) library. The definition below shows a generator for arithmetic expressions with fractions:

```
genExpr :: Int → Gen Expr
genExpr n
   | n == 0 = frequency [(1, liftM (Nat ∘ abs) arbitrary)
                        , (4, liftM2 (λa b → a :/: Nat b) arbitrary num)]
   | otherwise = oneof [genExpr 0
                        , liftM Negate rec
                        , binop (:+:)
                        , binop (:*:)
                        , binop (:-:)]
  where
    rec    = genExpr (n `div` 2)
    binop f = liftM2 f rec rec
    num = choose (1, 12)
```

```
instance Arbitrary Expr where
  arbitrary = sized genExpr
instance CoArbitrary Expr where
  coarbitrary (Nat n)     = variant 0 ∘ coarbitrary n
  coarbitrary (Var v)     = variant 1 ∘ coarbitrary v
  coarbitrary (Negate a)  = variant 2 ∘ coarbitrary a
  coarbitrary (a :+: b)   = variant 3 ∘ coarbitrary a ∘ coarbitrary b
  coarbitrary (a :*: b)   = variant 4 ∘ coarbitrary a ∘ coarbitrary b
  coarbitrary (a :-: b)   = variant 5 ∘ coarbitrary a ∘ coarbitrary b
  coarbitrary (a :/: b)   = variant 6 ∘ coarbitrary a ∘ coarbitrary b
```

This generator creates simple arithmetic expressions. The *frequency* function from the QuickCheck library allows us to specify that we prefer the generation of fractions above the generation of other expression constructs. When generating fractions we need to take care that we do not introduce a division by zero. As a simple solution, we generate only non-negative numbers for the denominator.

The last semantic function is *suitableExpr*, which checks whether or not the expression generator generates proper tasks.

```
suitableExpr :: Expr → Bool
suitableExpr = not ∘ finishedExpr
```

With this function we express that every expression that is not finished, i.e., at least one step has to be taken, is suitable. Function *suitableExpr* is rather basic, and probably accepts tasks we would not want to set. More appropriate tasks are obtained if we check for a minimal number of additions, or the inclusion of a negative number, in the definition of *suitableExpr*.

4.3 Properties

The following lemmas express properties that the components in an exercise should satisfy. The lemmas connect the various components of an exercise.

Definition 4.2. *Function unfocus converts a focused expression to a normal expression.*

Lemma 4.3. *Let S_0 be a state (Γ, ϕ, σ). If unfocus (ϕ) is a suitable start expression, then in all end states, the unfocused expression is finished:*

$$\forall\ (\Gamma', \phi', \epsilon)\ \in\ run\ (S_0)\ .\ finished\ (unfocus\ \phi')$$

Lemma 4.4. *Let $\equiv$ be the exercise's equivalence function, E the set of valid expressions, and R be the set of all rewrite rules for an exercise, i.e., the union of the rules used in the strategy and the additional rules set, and r an element of R. Then all r in R are semantics preserving:*

$$\exists\, e, e' \in E \,.\, e \overset{r}{\rightsquigarrow} e' \,.\, e \equiv e'$$

Definition 4.5. *Let d be a derivation, i.e., a sequence of pairs of rules and states. Then exprs (d) is the set of all expressions in a derivation:*

$$\mathit{exprs}\ (d) = \{\, \mathit{unfocus}\ (\phi) \mid (r, (\Gamma, \phi, \sigma)) \in d \,\}$$

Corollary 4.6. *Let S be a state. Then all expressions in all derivations are in the same equivalence class determined by the exercise's equivalence relation $\equiv$:*

$$\forall\, d \in \mathit{derivations}\ (S) \,.\, \forall\, e, e' \in \mathit{exprs}\ (d) \,.\, e \equiv e'$$

Lemma 4.7. *Let E be the set of expressions generated by the exercise's generator. All expressions $e \in E$ are suitable and not finished: $\mathit{suitable}\ (e) \wedge \neg\, \mathit{finished}\ (e)$.*

We use these properties to check the consistency of the definition of an exercise. An exercise definition is consistent if:

- the rules are semantics preserving,
- the application of a strategy to a generated expression leads to expressions that are finished,
- the generator generates suitable expressions that are not finished.

We have augmented the implementation of our strategy recogniser with tests that check if an exercise definition has the specified properties. Proving these lemmas for a particular exercise would guarantee that an exercise behaves as expected.

5 Domain reasoners

To support learning mathematics, many intelligent tutoring systems have been developed. These ITSs manage a collection of learning objects, and offer a variety of interactive exercises, together with a graphical user interface to enter and display mathematical formulas. Advanced systems also have components for exercise generation, for maintaining a student model, for configuring the tutorial strategy, and so on. ITSs often delegate dealing with exercise-specific problems, such as diagnosing intermediate answers entered by a student and providing feedback, to external components. These components can be computer algebra systems (CASs) or specialised *domain reasoners*. In general, a CAS will have no problems calculating an answer to a mathematics question posed at primary school, high school, or undergraduate university level. However, CASs are not designed to give detailed diagnoses or suggestions to intermediate answers. As a result, giving feedback using CASs is difficult. Domain reasoners, on the other hand, are designed specifically to give good feedback. An ITS typically uses multiple domain reasoners, and the behaviour of each of these is, to a large extent, determined by the domain.

A domain reasoner is responsible for the exercise-specific calculations, which are needed to provide feedback to students solving interactive exercises. A domain reasoner tracks the steps a student takes, generates hints, diagnoses errors, records progress, calculates worked-out solutions, etc. The functionality of the domain reasoner fundamentally depends on the domain, the rules that hold for the domain, and the strategies for solving exercises within the domain. According to Beeson's

design principles (Beeson, 1998) a domain reasoner should solve an exercise in the same way as a student does. For this purpose, we need, amongst others, fine control over the symbolic simplification procedures of the underlying mathematical machinery.

In the past few years, we have developed domain reasoners that help with diagnosing student behaviour in ITSs for calculating with fractions, performing Gaussian elimination, solving systems of linear equations and other linear algebra exercises, factoring polynomials, rewriting a logical term to disjunctive normal form, rewriting relation algebra terms, developing functional programs, and determining the derivative or integral of a function (Lodder et al., 2008; Gerdes et al., 2009; Heeren et al., 2010).

Chapter 4 introduces three fundamental concepts for constructing domain reasoners. Using these three concepts we can generate feedback. The core components of a domain reasoner are:

- A description of the domain, consisting of an abstract syntax together with a parser and a pretty-printer.
- Rules with which expressions in the domain are manipulated.
- Strategies for solving exercises in the domain.

Additionally, for each domain we need functionality for generating tasks, traversing terms, determining the equality of two terms, etc. Instances of these concepts are grouped together in an *exercise*, see Chapter 4. Some components are exercise-specific, such as the task generator and the strategy. Other components are domain-specific, such as the abstract syntax and the equality function, and may be shared among multiple exercises. A domain reasoner groups together one or more exercises.

Our domain reasoners offer feedback functionality via web services (Gerdes et al., 2008). These services act as an interface to our feedback functionality. We have specified a general set of feedback services, with which each of our domain reasoners can be accessed. Several intelligent tutoring systems, such as MATHDOX, ACTIVEMATH and the Freudenthal Institute Digital Mathematics Environment use our domain reasoning web services.

In addition to developers of intelligent tutoring systems, our domain reasoners are used by students and teachers. Students and teachers do not interact with our domain reasoners directly, but via an ITS. The different groups of users should be able to customise a domain reasoner (Pahl, 2003). They have various requirements with respect to customisation. For example, a student might want to see more detail at a particular point in an exercise, a teacher might want to enforce that an exercise is solved using a specific approach, and a developer of a mathematical

environment might want to compose a new kind of exercise from existing parts. Heeren and Jeuring (2010) show how our domain reasoners can be adapted and configured.

We use the input from students, via logs and statistics, to optimise our domain reasoners, for instance by distilling common mistakes and extracting buggy rules.

The next section (Section 5.1) gives an overview of the feedback services that we offer to ITSs. In Section 5.2 we show how to make a feedback service call. Section 5.3 explains how we use feedback scripts to translate the result of a feedback service call to a textual feedback message.

5.1 Feedback services

A mathematical learning environment may use domain reasoners for several classes of exercises. To minimise the effort of using multiple domain reasoners, domain reasoners share a set of *feedback services*, which are exercise independent. We have defined such a set of services around the exercise concept. These services provide the kinds feedback we have identified in Section 2.5.

There are no restrictions on the usage of our feedback services. An example of using our services is to start with giving correct/incorrect feedback, and to give semantically rich feedback on individual steps only when a student repeatedly fails to give a correct answer. Another possibility is to choose to only give feedback after the final submission of a student, showing diagnoses of all the steps.

We offer the following feedback services:

allfirsts. Suggests all next steps accepted by the strategy.

onefirst. Returns the possible next step according to the strategy, taking the rule ordering into account. This service uses the *allfirsts* service.

derivation. Returns a worked-out example, starting with the current term.

isfinished. Checks if the current expression is in a form accepted as a final answer.

stepsremaining. Computes how many steps remain to be done, according to the strategy. Only the derivation returned by the *derivation* service is considered.

apply. Applies a rule to a particular subterm of the current term. The rule does not need to be one of the rules that are accepted by the strategy. If the rule is not accepted by the strategy, we deviate from it. If the rule cannot be applied, the service call returns an error.

applicable. Collects the rewrite rules of a strategy and the additional rewrite rules, and reports which rules can be applied to the (sub)term that is in focus.

generate. Returns an initial state with a freshly generated term. This service takes an exercise code and an optional difficulty level as arguments.

diagnose. Diagnoses a term submitted by a student. The diagnose service checks whether or not the submitted term is:

- the result of a buggy rule application,
- not equivalent,
- similar to the previous term,
- accepted by the strategy,
- a deviation from the strategy using a known rule,
- correct.

Based on an exercise specification we can automatically calculate the feedback listed above. These services do not generate actual feedback messages. An ITS uses the results of the services, for example the name of an applicable rule, to generate a feedback message. For example, the service *stepsremaining* only gives the number of steps that need to be taken to solve the task. The ITS translates this number, say 5, into a proper feedback message: "There are at least 5 steps to be taken to solve the exercise.". To support ITSs in the generation of these feedback messages, we have a number of additional services that generate textual feedback messages based on the services listed above. We have defined textual variants of the *onefirst*, *derivation*, and *diagnose* services. The construction of these messages is configurable, see Section 5.3. We also have a number of administrative services. For example, we offer a service that lists all available exercises. We do not further discuss the definition of these administrative services.

Most of the feedback services are derived from the types of feedback that appear in the tutoring principles of Anderson (1993), and from the tutoring services found in VanLehn (2006). Some of the services cannot be found in these two sources, such as the generate service. These services are needed by the ITSs to which we offer our feedback functionality.

With the set of feedback services specified, we can provide the feedback that the current ITSs that use our domain reasoners require from us. It may well be that new users have extra or different requirements. We can easily adapt the existing services, or add new services that offer new kinds of feedback. For example, Section 6.2.1 shows the definition of a specialised service for the domain of functional programming.

5.1.1 Formalised Services

Feedback tops the list of factors leading to good learning (Biggs and Tang, 2007), but it is only effective when it is precise and to the point. To ensure that the feedback given by our domain reasoners is to the point and relevant, we give a precise definition of our services. The remainder of this section gives a formal definition of the feedback services. We specify the services in terms of the definitions given in Chapters 2 and 4.

allfirsts. The *allfirsts* service returns all next steps that are allowed by a strategy in a particular state:

$$\mathit{allfirsts}\ S_0 = \{\,(r, S) \mid S_0 \overset{r}{\twoheadrightarrow} S\,\}$$

The *allfirsts* service is defined in terms of the big step relation, which ignores minor rules. The rules in this set are paired with a new state, which is the result of applying the rewrite rule to the current state. Consider the following state S:

$$\begin{array}{l} S = (\emptyset \\ \quad , [\![\tfrac{2}{3} + \tfrac{3}{5}]\!] \\ \quad , \mathit{label}\ \texttt{"addTwoFractions"} \\ \quad\quad (\textsc{lcd} \lhd\star\rhd \mathit{repeat}\ (\mathit{somewhere}\ \textsc{rename}) \lhd\star\rhd \textsc{add} \lhd\star\rhd \mathit{try}\ \textsc{simpl}) \\ \quad) \end{array}$$

The strategy in this example state is the *addTwoFractions* strategy defined in Section 2.3.10. An *allfirsts S* service call gives the following result.

$$\begin{array}{l} \{\,(\textsc{rename} \\ \quad , (\{\,(n, 15)\,\} \\ \quad\quad , [\![\tfrac{10}{15}]\!] + \tfrac{3}{5} \\ \quad\quad , (\textsc{Up} \lhd\star\rhd \mathit{repeat}\ (\mathit{somewhere}\ \textsc{rename}) \lhd\star\rhd \textsc{add} \lhd\star\rhd \mathit{try}\ \textsc{simpl}) \\ \quad\quad)) \\ \quad , (\textsc{rename} \\ \quad\quad , (\{\,(n, 15)\,\} \\ \quad\quad\quad , \tfrac{2}{3} + [\![\tfrac{9}{15}]\!] \\ \quad\quad\quad , (\textsc{Up} \lhd\star\rhd \mathit{repeat}\ (\mathit{somewhere}\ \textsc{rename}) \lhd\star\rhd \textsc{add} \lhd\star\rhd \mathit{try}\ \textsc{simpl}) \\ \quad\quad\quad)) \\ \} \end{array}$$

The result is a set of two steps that a student can take. Both use the same rewrite rule RENAME, but they are applied to a different subterm. The rule LCD

does not show up in the set of *allfirsts* because it is a minor rule. The result of the application of this rule is the addition of a key/value pair, which stores the least common denominator in the environment. This information is in turn used by the RENAME rule. Not only the environment and the expression in focus are updated in the new state, but also the strategy. The strategy in the new state reflects what remains to be done. The UP minor rule is introduced by the *somewhere* combinator to get the focus back to its original place.

onefirst. The *onefirst* service returns a single possible next step that follows the strategy. This service uses the *allfirsts* service and the rule ordering.

$$\begin{aligned} &\mathit{onefirst}\ S_0 = (r, S) \\ &\quad \textbf{where}\ (r, S)\ \in\ \mathit{allfirsts}\ S_0\ \wedge\ \forall\ (r_i, S_i)\ \in\ \mathit{allfirsts}\ S_0\ .\ r\ \leqslant\ r_i \end{aligned}$$

Performing a *onefirst* service call on the state S from the previous paragraph, taking into account the rule ordering from the example exercise specification from Chapter 4, gives following result:

$$\begin{aligned} &(\text{RENAME} \\ &, (\{\,(n, 15)\,\} \\ &\ , [\![\tfrac{10}{15}]\!] + \tfrac{3}{5} \\ &\ , (\text{UP} \mathbin{<\!\star\!>} \mathit{repeat}\ (\mathit{somewhere}\ \text{RENAME}) \mathbin{<\!\star\!>} \text{ADD} \mathbin{<\!\star\!>} \mathit{try}\ \text{SIMPL}) \\ &\ )) \end{aligned}$$

Recall that if we need to choose between two occurrences of the same rule, the rule ordering takes the location of the subterm to which the rule is applied into account. The rule that can be applied to the subterm that is nearest to the root of the term is chosen. In the example, this is the left-most fraction.

derivation. The *derivation* service returns a worked-out solution of an exercise starting with the current expression.

$$\begin{aligned} \mathit{derivation}\ S_0 = (r_1, S_1)\ (r_2, S_2)\ \ldots\ (r_n, S_n)\ \textbf{where}\ \mathit{empty}\ (S_n)\ \wedge \\ \forall\ i\ \in\ 1 \ldots n\ .\ (r_i, S_i) = \mathit{onefirst}\ S_{i-1} \end{aligned}$$

A service call on S leads to:

$$\begin{aligned} &(\text{RENAME} \\ &, (\{\,(n, 15)\,\} \\ &\ , [\![\tfrac{10}{15}]\!] + \tfrac{3}{5} \\ &\ , (\text{UP} \mathbin{<\!\star\!>} \mathit{repeat}\ (\mathit{somewhere}\ \text{RENAME}) \mathbin{<\!\star\!>} \text{ADD} \mathbin{<\!\star\!>} \mathit{try}\ \text{SIMPL}) \\ &\ )) \\ &(\text{RENAME} \end{aligned}$$

$$
\begin{array}{l}
\quad , (\{\,(n, 15)\,\} \\
\quad\ , \frac{10}{15} + [\![\frac{9}{15}]\!] \\
\quad\ , (\textsc{Up} \mathbin{<\!\star\!>} \mathit{repeat}\ (\mathit{somewhere}\ \textsc{rename}) \mathbin{<\!\star\!>} \textsc{add} \mathbin{<\!\star\!>} \mathit{try}\ \textsc{simpl}) \\
\quad\ )) \\
\quad (\textsc{add} \\
\quad , (\{\,(n, 15)\,\} \\
\quad\ , [\![\frac{19}{15}]\!] \\
\quad\ , \mathit{try}\ \textsc{simpl} \\
\quad\ )) \\
\quad (\textsc{simpl} \\
\quad , (\{\,(n, 15)\,\} \\
\quad\ , [\![1 + \frac{4}{15}]\!] \\
\quad\ , \varepsilon \\
\quad\ ))
\end{array}
$$

isfinished. The *isfinished* service checks if the expression in a state is in a form accepted as a final answer. The *isfinished* service is an interface to the *finished* predicate defined in an exercise.

$$\mathit{isfinished}\ (\Gamma, [\![e]\!], \sigma) = \mathit{finished}\ (\mathit{unfocus}\ [\![e]\!])$$

stepsremaining. The *stepsremaining* service computes how many steps remain to be done according to the strategy. This is achieved by calculating the length of the derivation.

$$\mathit{stepsremaining}\ S_0 = \mathit{length}\ (\mathit{derivation}\ S_0)$$

apply. The *apply* service applies a rule to an expression in a state at a particular location, regardless of the strategy. The location is represented as a list of integers, where each integer n represents the number of steps to the right after a step downwards in a subexpression (the nth child). Starting at the root of an expression we can assign every subexpression a unique location.

The function *setFocus* puts the focus at a particular subexpression, using minor navigation rules.

$$
\begin{array}{l}
\mathit{setFocus}\ S\ [\,] \qquad = S \\
\mathit{setFocus}\ S\ (n : ns) = \mathit{setFocus}\ (\mathit{moveRight}\ n\ S')\ ns\ \textbf{where}\ S \overset{\textsc{Down}}{\to} S' \\
\mathit{moveRight}\ S\ n \mid n \leqslant 0 = S \\
\qquad\qquad\qquad\quad \mid n > 0 = \mathit{moveRight}\ S'\ (n-1)\ \textbf{where}\ S \overset{\textsc{Right}}{\to} S'
\end{array}
$$

The function *focusToTop* sets the focus to the root of an expression. We omit the definition of this function. The *apply* service is defined as follows:

$$\mathit{apply}\ r\ \mathit{loc}\ S_0 = S_1\ \textbf{where}\ \mathit{setFocus}\ (\mathit{focusToTop}\ S_0)\ \mathit{loc} \xrightarrow{r} S_1$$

For example, the following service call:

$$\mathit{apply}\ \textsc{simpl}\ [1,1]\ (\Gamma, [\![\tfrac{2}{3} - (\tfrac{1}{5} + \tfrac{2}{4})]\!], \sigma)$$

gives

$$(\Gamma, \tfrac{2}{3} - (\tfrac{1}{5} + [\![\tfrac{1}{2}]\!]), \sigma)$$

applicable. The *applicable* service takes a state and a location, and returns all major rules that can be applied to the subexpression at this location, independent of the strategy. Let R be the union of the rules in the strategy and the additional rule set, then *applicable* is defined as follows:

$$\mathit{applicable}\ \mathit{loc}\ S_0 = \{\, r \mid r \in R, S_1 \xrightarrow{r} S_2 \,\}$$
$$\textbf{where}\ S_1 = \mathit{setFocus}\ (\mathit{focusToTop}\ S_0)\ \mathit{loc}$$

generate. The *generate* service takes an exercise identification code and a difficulty level (optional), and returns an initial state with a freshly generated expression.

diagnose. The *diagnose* service diagnoses an expression submitted by a student. Possible diagnoses are:

- *Buggy*: a common misconception has been detected,
- *NotEq*: the current and submitted expression are not equivalent, so something is wrong,
- *Similar*: the submitted expression is similar to the current expression in the derivation,
- *Expected*: the submitted expression is expected by the strategy,
- *Detour*: the submitted expression was not expected by the strategy, but the applied rule was detected,
- *Correct*: the submitted expression is correct, but we cannot determine which rule was applied.

The *diagnose* service is defined as follows:

$$
\begin{array}{l}
\mathit{diagnose}\ (\Gamma, \phi, \sigma)\ \mathit{submitted} \\
\quad \mid e \not\equiv \mathit{submitted} \ \wedge \\
\qquad \exists\, b \in B\ .\ (\Gamma, e) \overset{b}{\rightsquigarrow} (\Gamma', \mathit{submitted}) = (\mathit{Buggy}, b) \\
\quad \mid e \not\equiv \mathit{submitted} \qquad\qquad\qquad\qquad\quad = \mathit{NotEq} \\
\quad \mid e \approx \mathit{submitted} \qquad\qquad\qquad\qquad\quad = \mathit{Similar} \\
\quad \mid \exists\, ((\Gamma', \phi', \sigma'), r) \in \mathit{allfirsts}\ (\Gamma, \phi, \sigma)\ . \\
\qquad \mathit{submitted} \approx \mathit{unfocus}\ (\phi') \qquad\qquad\quad = (\mathit{Expected}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad , (\Gamma', [\![\mathit{submitted}]\!], \sigma')) \\
\quad \mid \exists\, r \in R\ .\ (\Gamma, e) \overset{r}{\rightsquigarrow} (\Gamma', \mathit{submitted}) \quad = (\mathit{Detour}, r) \\
\quad \mid \mathit{otherwise} \qquad\qquad\qquad\qquad\qquad\quad = \mathit{Correct} \\
\quad \textbf{where}\ e = \mathit{unfocus}\ (\phi) \quad \text{-- the current expression}
\end{array}
$$

Where $\equiv$ is the equivalence relation, $\approx$ the similarity relation, R the set of additional rules, and B the buggy rule set specified in an exercise. The *diagnose* service not only returns the result of the analysis (e.g., *NotEq* or *Expected*), but also detailed information. If we detect the application of a (buggy) rule, we return this rule. If the submitted expression is expected by the strategy, we return the new state together with the applied rule.

5.2 Web services

Our feedback services are available in the form of online web services. Web services are easier to maintain and deploy than, for instance, a library. For example, updating the services after a bug fix only requires replacing a single binary on the deployment server. Another important reason why we use web services is *abstraction*. By using web services we abstract away from our implementation details. It enables users of our services to access our functionality without having to know about the details of a domain reasoner. The web service interface serves as a contract between the provider and consumer. We can standardise our interface while retaining the possibility to adapt our feedback engine. Developers of ITSs that use our services are fully in charge of how to use and present feedback to a student.

Web services support inter-operable machine-to-machine interaction over a network. Currently we support two communication protocols: JSON-RPC (Javascript object notation - remote procedure call) and XML-RPC. A communications protocol describes the format of the messages that are exchanged between communicating parties, and rules for exchanging those messages. The software framework used

for our domain reasoners has a modular architecture and can easily be extended with other protocols. In this document we use JSON-RPC for the examples. The JSON-RPC invocation of our feedback services can be done via a CGI (common gateway interface) binary using HTTP (hypertext transfer protocol).

Our domain reasoners can be reached via the following URL (uniform resource locator):

```
http://ideas.cs.uu.nl/cgi-bin/ideas.cgi?input=<JSON_input>
```

The general structure of the JSON input parameter that needs to be supplied in the URL is:

```
{ "method" : <service name>
, "params" : <list of parameters>
, "id"     : <request id> }
```

An interesting feature of our protocol is that it is *stateless*. When necessary the state is given as an extra parameter. We represent the state as a four tuple containing:

- an exercise identifier,
- a parameter that encodes the steps that a student has taken. This parameter corresponds to a location in the strategy. The encoding is rather simple: the first element of the list is the number of rules that have been applied, and the remaining elements indicate if we go left or right in case we encounter a choice combinator. If the element is 0 we go left, if it is 1 we go right.
- the current expression,
- an environment parameter that holds the key/value set.

This representation of the state is a serialised form of a state defined in Definition 2.6. Due to the stateless protocol, the state parameter can be saved and the exercise can be continued at a later point. This offers ITSs using our domain reasoners the possibility to save a student's work.

The following example is a JSON-RPC invocation of the *onefirst* service. It calls the service with a list of parameters, here a singleton list containing a four tuple describing the current state. The example shows the structure of the input parameter in a service request URL.

```
http://ideas.cs.uu.nl/cgi-bin/ideas.cgi?input=
  { "method" : "onefirst"
  , "params" : [["addFracEx", "[]", "2/3+3/5", ""]]
  , "id"     : 42 }
```

The URL needs to be escaped from illegal characters (like spaces and curly braces), but for presentation purposes we use the representation with these characters. The CGI binary has one parameter called `input`. Based on the example service call, our domain reasoner generates the following reply:

```
{ "result": [ "rename", "[1]", [ "addFracEx"
                               , "[5,1,1,1,0,1]"
                               , "(10/15 + 3/5))"
                               , {"n": "15"} ]
, "error": null
, "id": 42 }
```

Our domain reasoner replies that applying the rewrite rule `rename` is the first step that a student can take. Furthermore, the reply returns the location of the focus, and the new state after applying the suggested rewrite rule.

5.3 Feedback scripts

We offer textual versions for some feedback services, such as the *diagnose* service. A textual feedback service translates the output of a feedback service to a textual feedback message. Texts are specified and configured in so-called *feedback scripts*. These scripts are external text files containing responses for various situations. In case of the *diagnose* service, depending on the diagnosis (e.g., a common mistake was recognised, or the submitted term is correct and accepted by the specified strategy), a feedback message is selected from the script and reported to the student. Figure 5.1 gives a graphical representation of the communication between an ITS, and a domain reasoner that uses a feedback script. A feedback text can be selected based on the current location in the strategy, denoted by a label. Other selection criteria are the name of the rule that is recognised (possibly a buggy rule), or the submitted term being correct or not.

We can generate three levels of feedback messages for the next step to take, which can be categorised as follows (Vanlehn et al., 2005):

- *general:* a general, high-level statement about the next step to take;
- *concrete:* a more detailed explanation of the next step in words;
- *bottom-out:* the exact next step to carry out, possibly accompanied with the rewritten term.

The level of a feedback message for the next step is passed as an argument to a textual feedback service.

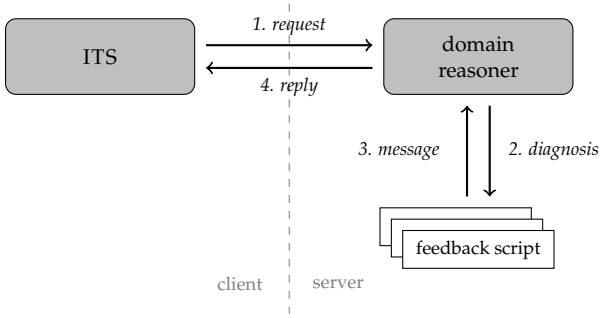


Figure 5.1: Feedback scripts communication

Having only static texts in the feedback scripts (that is, texts that appear verbatim in the script) restricts the expressiveness of the messages that can be reported. We allow a variety of attributes in the textual messages of a script, and these attributes are replaced by dynamic content depending on the context. In this way, messages can include the result from a feedback service, such as the number of steps remaining. Feedback scripts contain some more constructs to facilitate writing feedback messages, such as local string definitions and an import mechanism.

The text for the three levels of feedback messages can be specified in a feedback script as follows:

```
text addTwoFractions  = { Use the procedure for adding fractions:
                          If necessary, rename the fractions so
                          that they have a common denominator. }
hint concrete         = { @expected }
hint bottom-out       = { @expected: this results in @after }
```

The general level uses the text that belongs to the active label. The attribute `expected` is replaced by the (translation of the) rule suggested by the strategy. The attribute `after` represents the term after application of the expected rule. Another example of dynamic feedback generation is the translation of a rule that makes use of argument information. The attribute `arg1` is replaced dynamically by a textual representation of the first argument. For example, the translation for the RENAME rule can be defined as follows:

```
text rename = { multiply the fraction @arg2 with @arg1/@arg1 }
```

The first argument of the RENAME rule is the factor to scale with, and the second argument the fraction to be renamed. Rule translations are, for instance, used in the textual version of the *derivation* feedback service for generating worked-out solutions:

$$\frac{2}{3} + \frac{3}{5}$$

$$\Rightarrow \{\textit{multiply the fraction } \tfrac{2}{3} \textit{ with } \tfrac{5}{5}\}$$

$$\frac{10}{15} + \frac{3}{5}$$

$$\Rightarrow \{\textit{multiply the fraction } \tfrac{3}{5} \textit{ with } \tfrac{3}{3}\}$$

$$\frac{10}{15} + \frac{9}{15}$$

$$\Rightarrow \{\textit{add the fractions by adding their numerators}\}$$

$$\frac{19}{15}$$

$$\Rightarrow \{\textit{simplify the fraction}\}$$

$$1 + \frac{4}{15}$$

An important advantage of external feedback scripts is that they can be changed easily, without recompiling the tutoring software. This approach also allows us to add feedback scripts that support new exercises. A final benefit is that supporting multiple languages (as opposed to only English) comes quite natural. Each language supported has its own feedback script.

Part II

Haskell Tutor

6 A programming tutor for Haskell

Learning by doing through developing programs, and learning through feedback on these programs are essential aspects of learning programming. Introductory functional programming courses often start with distinguishing the various steps a student has to take to write a program. A teacher usually explains by example how to develop a program: give the main function a name; if the function takes an argument, give the argument a name; if the value of the argument determines the action to be taken subsequently, analyse the argument, and depending on the form of the argument, develop the appropriate right-hand sides, etc. Once a student starts developing a program herself, in a lab session or at home, this kind of explanatory help is usually not present. Moreover, giving immediate help to large classes of students is almost always impossible. Especially beginning programmers are often at a loss about how to proceed when developing a program.

We have developed Ask-Elle , a programming tutor

- that targets first year computer science students,
- in which a student incrementally develops a program that is equivalent (modulo syntactic variability) to one of the teacher-specified model solutions for a programming problem,
- that gives feedback and hints on intermediate, incomplete, and possibly buggy programs, based on teacher-specified annotations in model solutions,

- to which teachers can easily add their own programming exercises, and in which teachers can adapt feedback,

- and in which a student can use her preferred step-size in developing a program: from making a minor modification to submitting a complete program in a single step.

Using our programming tutor a student develops a program by making small, incremental, changes to a previous version of the program. Other common scenarios in teaching programming are to give a student an incomplete program, and ask her to complete the program, or to give a student a program, and ask her to change the program at a particular point. In such assignments, a student *refines* or *rewrites* a program. Both rewriting and refining preserve the semantics of a program, and refining possibly makes a program more precise.

The feedback that we offer, such as giving a hint, is derived from a strategy. Strategies play a central role in our approach. We use strategies to capture the procedure of how to solve an exercise. A strategy describes which basic steps have to be taken, and how these steps are combined to arrive at a solution. In case of a functional programming exercise, the strategy describes how to incrementally construct a program. We reuse the strategy EDSL described in detail in Section 2.3 for defining strategies for programming.

In this part of this thesis 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, 2003). We believe, however, that our approach based on programming strategies is also applicable to other programming languages and programming paradigms. Our method is not tied to a particular programming language. The concepts on which our approach is based, such as strategies, refinement rules, and program transformations, are applicable to every programming language.

We start this chapter with explaining the high-level architecture of the programming tutor (Section 6.1), and continue with giving a description of the functional programming domain (Section 6.2). We continue with describing how to test incomplete programs (Section 6.3), which is an important part of the domain description. Section 6.4 shows an example interactive session, which shows how our functional programming tutor supports the incremental development of programs. We conclude this chapter with a description of some conducted experiments. The chapters in the remainder of this part of the thesis concentrate on different aspects of the tutor, such as how to specify exercises and how to construct programming strategies.

6.1 Programming tutor overview

Our programming tutor, called ASK-ELLE , offers exercises in the functional programming language Haskell. We use concepts and techniques from software technology, such as strategies, parsing, and program transformations, to calculate semantically rich feedback. Our tutor can be accessed via a web browser[1]. On the main page, a student selects an exercise to work on (such as *reverse*). While developing a program, a student can check that she is still on a path to a correct solution, ask for a single hint or all possible choices on how to proceed at a particular stage, or ask for a worked-out solution.

6.1.1 Architecture

The programming tutor consists of a front- and back-end. The front-end handles the interaction with the student, such as displaying feedback messages. The back-end takes care of the feedback generation. Figure 6.1 shows a schematic overview of our programming tutor.

The front-end of the tutor is implemented as a web application, which makes uses of AJAX (asynchronous Javascript and XML) technology to make service calls. Each time a student clicks a button such as `Check` or `Hint`, our web application sends a service request to the functional programming domain reasoner (the back-end).

The back-end of our tutor is a domain reasoner for the domain of functional programming exercises in Haskell. The domain reasoner is built using the software framework for specifying domain reasoners, which we introduced in Part I. Our functional programming domain reasoner is also available for external programming environments. The domain reasoner uses the Helium compiler for Haskell to calculate feedback. The Helium compiler has been developed to give better feedback to students on the level of syntax and types (Heeren et al., 2003). We reuse Helium's error messages when a student makes a syntax-error, or develops a wrongly typed program. The domain reasoner offers the services that are listed in Section 5.1. For example, if a student submits a syntax- and type-correct program, we analyse the submitted program using the *diagnose* service.

The domain reasoner is stateless: all information the domain reasoner needs is included in the service request. For example, a request to check a program sends the strategy for solving the exercise together with the previous and new expression of the student to the *diagnose* feedback service. The result of the *diagnose* service is converted to a feedback message using feedback scripts (see Section 5.3).

[1] `http://ideas.cs.uu.nl/ProgTutor/`

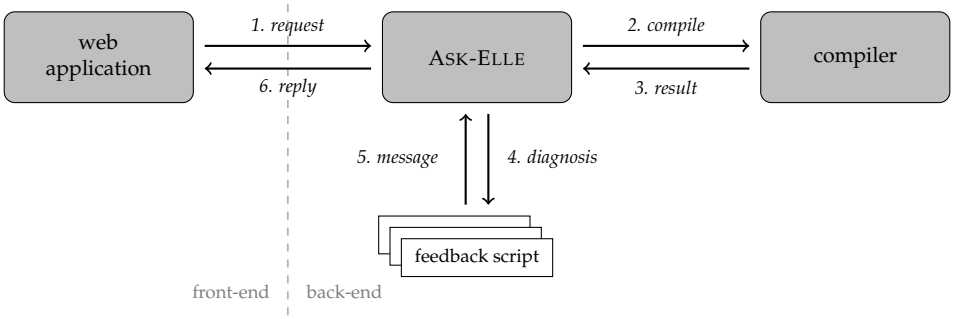


Figure 6.1: ASK-ELLE architecture

6.1.2 Recognising student programs

In a nutshell, we recognise (incomplete) student programs as follows. A teacher specifies her exercises by giving a set of annotated *model solutions* for a programming problem. A model solution is a program that an expert writes, using good programming practices. We derive a *programming strategy* from this set of model solutions. We use this strategy to track the progress of a student, and to generate feedback. Our tutor supports the incremental construction, in a top-down fashion, of a model solution. It recognises incomplete versions of these solutions, together with all kinds of syntactical variants. We support the refinement of programs, but instead of showing that a program ensures a post-condition, we assume a program to be correct if we can determine it to be equal to a model solution. A student applies *refinement rules* to an intermediate program to make it more defined. Before we compare a student (intermediate) program to the programs generated by the strategy, we *normalise* all programs using program transformations. For example, in this normalisation procedure we give all variables a unique name (α-renaming).

6.2 Domain description

We have constructed a domain reasoner for the domain of functional programming in Haskell. As explained in Section 2.1, we need to give instances for the three fundamental components that are necessary for generating feedback: the domain, rules for reasoning about the domain, and strategies for guiding reasoning. Feedback generation is offered to the web application front-end via feedback services. In this section we introduce the different components for the functional programming domain reasoner. The following chapters provide the details of these components.

The first component is a description of the domain of functional programs. The

domain is given by an abstract syntax in the form of a Haskell datatype, a parser that parses concrete syntax to abstract syntax, and a pretty printer that shows abstract syntax trees as programs in concrete syntax. We have constructed a simplified version of the Helium AST. We could have reused the Helium AST, but having our own AST allows us to abstract away from Helium. This makes it possible to more easily switch to another compiler, such as GHC (Glasgow Haskell compiler). We reuse many components from the Helium compiler, such as the parser, pretty printer, and type checker. We have specified a view to go back and forth between the Helium AST and our own.

The second component is the rules with which the terms in the domain, the (intermediate) programs, are manipulated. In contrast to mathematical domains, we use *refinement* rules instead of rewrite rules. Rewriting preserves the semantics of a program; refining makes a program more defined. Refinement rules typically replace an unknown part by some expression. A program refinement rule can introduce one or more new unknown parts. A program is complete if it does not contain any more unknown parts. Every constructor in the AST has a corresponding refinement rule that refines an undefined value to the constructor, possibly applied to undefined values. For example, we have a construct for an expression variable and hence we have a refinement rule that introduces an expression variable. We explain the refinement rules in detail in Chapter 8.

The third component is the strategy for solving a functional programming exercise. The strategy for a particular programming exercise is derived from a set of model solutions. In Chapter 8 we explain the details of this process. The programming strategy allows a student not only to give one of the model solutions, but many variants of it.

Besides these three components we need some additional components, such as the equivalence and similarity relations, before we can specify programming exercises. Chapter 4 contains a complete overview of these components. We list the additional components that are shared between all programming exercises:

equivalence. Checking if two programs have the same input/output behaviour is undecidable. We do want to give feedback about the correctness of submitted programs. If a submitted program follows a strategy, we know that it is equal to a model solution and hence correct. If a student program does not follow a strategy, we still want to be able to give feedback. As an approximation we use *testing* to check if a program behaves the same as one of the model solutions. If we can find a counter example, we know that the student program is wrong, and we report this. If a student program passes our tests, we regard it as equivalent. Strictly speaking, this is not correct and we therefore need to construct our feedback messages with care, because if a student program passes the test, it is not guaranteed to be correct. Section 6.3 describes how

we test (intermediate) programs.

similarity. To determine whether or not two (intermediate) programs are similar to each other we use normalisation. Our normalisation procedure uses many program transformations to rewrite a program to a normal form. After normalising both programs, we compare them syntactically. In Chapter 9 we describe the normalisation procedure.

generator. Strategies for mathematical exercises can be used to solve many tasks (containing different terms). For example, adding the fractions in $\frac{1}{2} + \frac{2}{3}$ and $\frac{2}{3} + \frac{1}{5}$ can be solved using a single strategy. In contrast to mathematical strategies, a programming strategy only solves a single programming exercise. For instance, a programming strategy for writing the *reverse* function cannot be used to solve a programming exercise that asks to define the *length* function. The generator for the functional programming domain is very simple. It only generates a completely undefined program, since every programming exercise starts with such a program.

suitable. It is unnecessary to check whether or not a start term is suitable, because we always start out with the same undefined start program. The *suitable* predicate always returns *True* for programming exercises.

finished. The *finished* predicate checks if a submitted program can be accepted as a final answer. The *finished* check should be independent of the strategy, because we use this predicate (amongst others) to validate strategies. To verify if a student is finished and has solved a programming exercise, we check if the program is fully defined (i.e., the program does not contain holes anymore), and test if it behaves the same as a model solution.

The remaining components of an exercise: the buggy rules set, the additional rules set, and the rule ordering relation are not used in the generation of feedback for programming exercises.

Using the above components we can construct a functional programming exercise in Haskell. To add a new programming exercise a teacher adds a set of model solutions for that exercise. All exercise components are constructed automatically from these model solutions. A teacher can adapt the generated feedback by annotating the model solutions. Chapter 7 shows how model solutions can be annotated.

Our programming tutor offers a number of programming exercises. Figure 6.2 shows the list of exercises currently offered. These exercises are a subset of the Haskell-99 [2] set of programming problems. Solving these programming exercises

[2] `http://www.haskell.org/haskellwiki/99_Haskell_exercises`

supposedly gives a good overview of the programming language. This set of 99 programming problems is also available for other languages, such as Prolog, Lisp, Scala, and Perl.

6.2.1 Additional feedback services

When solving a programming exercise a student can ask for an analysis of the steps taken so far, or ask for a hint. The web application translates a request of the student to a service call to the functional programming domain reasoner. The feedback services described in Section 5.1 are not sufficient for the functional programming domain reasoner. We need two additional feedback services, which may also be useful for other domains.

The additional services are:

deepdiagnose. Similar to the Lisp tutor (Corbett et al., 1988), the refinement rules in our tutor model Haskell at the finest grain size that has functional meaning in Haskell. It is therefore likely that a student wants to take multiple steps at once. We want to offer students the possibility to take multiple steps at once. For example, when defining a case alternative, it is not uncommon that a student defines the entire case alternative in one step.

The original *diagnose* service only recognises the first next step. The *diagnose* feedback service takes the strategy, the previous program, and the current program as arguments. It determines if the current program can be derived from the previous program using any of the rules that are allowed by the strategy, generated by the *allfirsts* service. The *diagnose* service uses the similarity relation to calculate a normal form of both the expected and the submitted programs, and checks that the submitted program appears in the set of expected programs.

The *deepdiagnose* service recognises multiple steps. The deepdiagnose service is defined as follows. We only show the case that is different from the diagnose service:

$$
\begin{array}{l}
\mathit{deepdiagnose}\ (\Gamma, \phi, \sigma)\ \mathit{submitted} \\
\quad \ldots \\
\quad \mid \exists\ (\Gamma', \phi', \sigma')\ \in\ \mathit{dfs}\ (\Gamma, \phi, \sigma)\ .\ \mathit{submitted}\ \approx\ \mathit{unfocus}\ (\phi') \\
\qquad = (\mathit{Expected}, (\Gamma', [\![\mathit{submitted}]\!], \sigma')) \\
\quad \ldots \\
\quad \textbf{where} \\
\qquad \mathit{dfs}\ [\,] \qquad = [\,] \\
\qquad \mathit{dfs}\ (S : SS) = S : \mathit{dfs}\ (\mathit{map}\ \mathit{fst}\ (\mathit{allfirsts}\ S) +\!\!+\ SS)
\end{array}
$$

Where $\approx$ is the similarity relation, and *dfs* a function that performs a depth-first search. In Chapter 10 we show how we can search efficiently for a similar intermediate program.

taskdescription. We can add labels to any (sub)strategy using the *label* combinator (see Section 2.3.5). These labels introduce minor rules that are used to indicate whether or not a label is active at a certain position in the strategy. We can attach a feedback message to these labels using our feedback scripts. The actual label in the strategy acts as an identifier for a feedback message in an accompanying feedback script.

The *taskdescription* service returns the feedback messages for all active labels.

Furthermore, we have added textual versions for the *deepdiagnose* and *allfirsts* services.

6.3 Testing incomplete programs

When a student deviates from a programming strategy, we cannot use the strategy for calculating feedback. In this case, our feedback services use other components of an exercise to generate feedback, such as the equivalence function, see Chapter 4. Checking if two programs are equivalent is very hard, and in general undecidable. For programming exercises we use *testing* as an approximation.

There exist quite a few libraries for testing in Haskell, such as SmallCheck (Runciman et al., 2008) and QuickCheck (Claessen and Hughes, 2000). We use QuickCheck to test programs submitted by students. Using QuickCheck we can specify logical properties that a program (or a function) should satisfy. QuickCheck generates test-cases and tries to falsify the given property. If QuickCheck can falsify a property, it tries to find the smallest counter example. This process is called shrinking.

QuickCheck normally only tests fully defined programs, as do other testing libraries for Haskell. However, most of the programs that we want to test in our tutor are not fully defined; we want to test incomplete programs. Consider the following incomplete program for *reverse*:

$$
\begin{array}{l}
\mathit{reverse} = \mathit{reverse}'\ [\,] \\
\quad \textbf{where} \\
\quad\quad \mathit{reverse}'\ \mathit{acc}\ [\,] \quad\quad = [\,] \\
\quad\quad \mathit{reverse}'\ \mathit{acc}\ (x : \mathit{xs}) = \mathit{reverse}'\ \bullet\ \mathit{xs}
\end{array}
$$

Although the definition of *reverse* is incomplete, we can already prove that this program does not meet the specification of *reverse*. The first function binding of $\mathit{reverse}'$, which pattern matches on the empty list, returns the empty list, whereas it should have returned the accumulated list.

We use a modified version of QuickCheck[3], which can test incomplete programs. If this version of QuickCheck evaluates an undefined part of a program, it throws a special exception. When this exception is caught, QuickCheck considers the exception as expected behaviour and continues with the test. We generate a straightforward property to test if the student program behaves the same as a model solution. For example, the property for the *reverse* exercise is defined as follows:

```
prop_EqReverseModel :: [Int] → Bool
prop_EqReverseModel xs = model xs == student xs
  where
    model xs   = Prelude.reverse xs
    student xs = ...
```

Note that it does not matter to which model solution we compare the student program, since all model solutions should have identical behaviour.

Running the modified version of QuickCheck with the given property and the example student definition given earlier, gives the following result:

```
> quickCheck prop_EqReverseModel
*** Failed! Falsifiable (after 4 tests and 1 shrink):
[0]
```

This result can be used as input for the construction of a feedback message, which can be presented to the student.

6.4 An interactive session

This section shows some interactions of a hypothetical student with the tutor. We assume that the student has just started a course on functional programming, and has visited lectures on how to write simple functional programs on lists. Her teacher has set the exercise number 5 from the Haskell-99 list: reverse a list. The exercise is based on the following model solutions:

```
  -- Use the prelude function foldl
reverse = foldl (flip (:)) []
```

and

```
  -- Introduce a helper function that uses an accumulating parameter
reverse = reverse' []
```

[3]The modification is contributed by Koen Claessen.

```
  where
    reverse' acc []     = acc
    reverse' acc (x:xs) = reverse' (x:acc) xs
```

Note the accumulating parameter model solution for *reverse* is in essence the same as the model solution using *foldl*. Our tutor recognises the accumulating parameter solution from the *foldl* model solution. However, if we omit this model solution, we would not be able to guide the student towards this solution.

In addition to these two model solutions, the tutor can also recognise the construction of the naive, quadratic time solution for *reverse*, often implemented by means of an explicit recursive definition:

```
  -- Use explicit recursion
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

If a student implements this version of *reverse*, the tutor can tell the student that this is a correct definition of *reverse*, but that it is a quadratic time algorithm, and that a linear-time algorithm is preferable[4]. Despite the fact that we can also specify suboptimal solutions, we will call these solutions "model solutions".

We now show a couple of possible scenarios in which the student interacts with the tutor to solve this problem. At the start of a tutoring session the tutor gives a problem description:

```
Write a function that reverses a list:  reverse :: [a] -> [a].
For example:

    > reverse "A man, a plan, a canal, panama!"
    "!amanap ,lanac a ,nalp a ,nam A"
    > reverse [1,2,3,4]
    [4,3,2,1]
```

and displays the name of the function to be defined:

```
reverse = •
```

The task of a student is to refine the incomplete parts, denoted by • , of the program. The symbol • is used as a placeholder for a hole in a program that needs to be refined to a complete program. A student can use such holes to defer the refinement

[4] We have not yet implemented this feature. However, it is relatively easy to detect which model solution was implemented.

of parts of the program. After each refinement, a student can ask the tutor whether or not the refinement is bringing him or her closer to a correct solution, or, if the student doesn't know how to proceed, ask the tutor for a hint. Besides holes, a student can also introduce new declarations, function bindings, alternatives, and refine patterns.

Suppose the student has no idea where to start and asks the tutor for help. The tutor offers several ways to help a student. For example, it can list all possible ways to proceed solving an exercise. In this case, the tutor would respond with:

```
There are several ways you can proceed:
```

- `Introduce a helper function that uses an accumulating parameter.`
- `Use the prelude function` *foldl*`.`
- `Use explicit recursion.`

We assume here that a student has some means to obtain information about concepts such as accumulated parameters that are mentioned in the feedback texts given by the tutor. This information might be obtained via lectures, an assistant, or lecture notes, or might even be included in the tutor at some later stage. Among the different possibilities, the tutor can make a choice, so if the student doesn't want to choose, but just wants a single hint to proceed, she gets:

```
Introduce a helper function that uses an accumulating
parameter.
```

Here we assume that the teacher has set up the tutor to prefer the solution that uses a helper function with an accumulating parameter. The student can ask for more detailed information at this point, and the tutor responds with increasing detail:

```
Define function reverse in terms of a function reverse',
which takes an extra parameter in which the reversed
list is accumulated.
```

with the final bottom-out hint:

```
Define:
```

$$\begin{array}{l} \mathit{reverse} = \mathit{reverse}'\ \bullet \\ \quad \textbf{where}\ \mathit{reverse}'\ \mathit{acc}\ \bullet = \bullet \end{array}$$

At this point, the student can refine the function at multiple positions. In this exercise we do not impose an order on the sequence of refinements. However, the

tutor offers a teacher the possibility to enforce a particular order of refinements. Suppose that the student chooses to implement *reverse′* by pattern matching on the second argument, which is a list, starting with the empty list case:

$$
\begin{array}{l}
\mathit{reverse} = \mathit{reverse}'\ [\,] \\
\quad \textbf{where} \\
\quad\quad \mathit{reverse}'\ \mathit{acc}\ [\,] = \bullet
\end{array}
$$

Note that this step consists of two smaller steps: the argument to *reverse′* has been instantiated to $[\,]$, and the definition of *reverse′* got an extra argument. She continues with:

$$
\begin{array}{l}
\mathit{reverse} = \mathit{reverse}'\ [\,] \\
\quad \textbf{where} \\
\quad\quad \mathit{reverse}'\ \mathit{acc}\ [\,] = \mathit{acc}
\end{array}
$$

which is accepted by the tutor. If the student now asks for a hint, the tutor responds with:

```
Define the non-empty list case of reverse'
```

She continues with

$$
\begin{array}{l}
\mathit{reverse} = \mathit{reverse}'\ [\,] \\
\quad \textbf{where} \\
\quad\quad \mathit{reverse}'\ \mathit{acc}\ [\,]\ \ \ \ \ \ = \mathit{acc} \\
\quad\quad \mathit{reverse}'\ \mathit{acc}\ (x : \mathit{xs}) = \bullet
\end{array}
$$

which is accepted, and then

$$
\begin{array}{l}
\mathit{reverse} = \mathit{reverse}'\ [\,] \\
\quad \textbf{where} \\
\quad\quad \mathit{reverse}'\ \mathit{acc}\ [\,]\ \ \ \ \ \ = \mathit{acc} \\
\quad\quad \mathit{reverse}'\ \mathit{acc}\ (x : \mathit{xs}) = \mathit{reverse}'\ \ \bullet\ \ \mathit{ys}
\end{array}
$$

which gives:

```
Error:  undefined variable ys
```

This is an error message generated by the compiler for the programming language. The student continues with:

$$
\begin{array}{l}
\mathit{reverse} = \mathit{reverse}'\ [\,] \\
\quad \textbf{where}
\end{array}
$$

```
reverse' acc []     = acc
reverse' acc (x:xs) = reverse' • xs
```

Thinking of a possible optimisation, the student rewrites the program as follows:

```
reverse = reverse' []
  where
    reverse' acc []     = []
    reverse' acc (x:xs) = reverse' • xs
```

The tutor responds with:

```
This program is incorrect.
Counterexample:  [0].
```

The student undoes the last change, and continues with:

```
reverse = reverse' []
  where
    reverse' acc []     = acc
    reverse' acc (x:xs) = reverse' (x:acc) xs
```

The tutor ends with the message:

```
Done!  You have solved the exercise.
```

6.4.1 Integers within a range

The next example we show is problem 22 from the Haskell-99 questions: Write a function which enumerates all numbers contained in a given range. For example:

```
> range 4 9
[4,5,6,7,8,9]
```

The Haskell 99 page mentions six solutions to this problem; here is one:

```
range x y = unfoldr (λi → if i == succ y then Nothing else Just (i, succ i)) x
```

This solution uses the *unfoldr* function defined by:

```
unfoldr       :: (b → Maybe (a, b)) → b → [a]
unfoldr f b = case f b of
                Just (a, new_b) → a : unfoldr f new_b
                Nothing         → []
```

Our system prefers the solution using *unfoldr*. The following shows the tutor's response when asked for a derivation:

$$range = \bullet$$

$\Rightarrow$ { Introduce parameters }

$$range\ x\ y = \bullet$$

$\Rightarrow$ { Use *unfoldr* }

$$range\ x\ y = unfoldr\ \bullet\ \bullet$$

$\Rightarrow$ { Start at x }

$$range\ x\ y = unfoldr\ \bullet\ x$$

$\Rightarrow$ { Introduce a lambda-abstraction }

$$range\ x\ y = unfoldr\ (\lambda i \rightarrow \bullet\ )\ x$$

$\Rightarrow$ { Introduce an **if-then-else** to specify a stop criterion }

$$range\ x\ y = unfoldr\ (\lambda i \rightarrow \textbf{if}\ \bullet\ \textbf{then}\ \bullet\ \textbf{else}\ \bullet\ )\ x$$

$\Rightarrow$ { Introduce the stop criterion }

$$range\ x\ y = unfoldr\ (\lambda i \rightarrow \textbf{if}\ i == succ\ y\ \textbf{then}\ \bullet\ \textbf{else}\ \bullet\ )\ x$$

$\Rightarrow$ { Return *Nothing* for the stop criterion }

$$range\ x\ y = unfoldr\ (\lambda i \rightarrow \textbf{if}\ i == succ\ y\ \textbf{then}\ Nothing\ \textbf{else}\ \bullet\ )\ x$$

$\Rightarrow$ { Give the output value and the value for the next iteration }

$$range\ x\ y = unfoldr\ (\lambda i \rightarrow \textbf{if}\ i == succ\ y\ \textbf{then}\ Nothing\ \textbf{else}\ Just\ (i, succ\ i))\ x$$

These interactions show that our tutor can

- give hints about which step to take next, in various levels of detail,
- list all possible ways in which to proceed,
- point out that an error has been made, and where the error appears to be,
- show a complete worked-out example.

In addition to the web application front-end, we also have a command-line version of our programming tutor. We mainly use the command-line version to quickly test new functionality and new exercises. We do not intend to use this front-end on a large scale.

```
[alex@edoras ~/Documents/Research/FPTutor]$ bin/fptutor.cgi --interactive

       >>               >=>             >========>  >=>  >=>
      >>=>              >=>             >=>         >=>  >=>
     >> >=>      >===>  >=>  >=>        >=>         >=>  >=>   >==>
    >=>  >=>    >=>     >=> >=>  >====> >======>    >=>  >=> >>   >=>
   >=====>>=>     >==>  >=>=>           >=>         >=>  >=> >>===>>=>
  >=>      >=>      >=> >=> >=>         >=>         >=>  >=> >>
 >=>        >=> >=> >=> >=>  >=>        >========>  >==> >==> >====>

Please choose one of the following exercises:

1:   last
2:   butlast
...
8:   reverse
...
22:  range
[1 .. 18] > 8

  Write a function that reverses a list: reverse :: [a] -> [a].
  For example:

  > reverse "A man, a plan, a canal, panama!"
  "!amanap ,lanac a ,nalp a ,nam A"

  > reverse [1,2,3,4]
  [4,3,2,1]

Ask-Elle > diagnose
Type a new term, you can use multiple lines and stop with a single . on a line.
| reverse = foldl (flip (:)) []
| .

Feedback: Correct.

Ask-Elle> :q
```

6.5 Experiments

We have used our functional programming tutor in a course on functional programming for bachelor students at Utrecht University in September 2011. The course attracted more than 200 students. Around a hundred of these students have used our tutor in two sessions in the second week of the course after three lectures. 40 students filled out a questionnaire about the tutor, and we collected remarks at the lab session in which the students used the tutor. Table 6.1 shows the questions and the average of the answers on a Likert scale from 1 to 5. The first seven questions

#	Question	Score
1	The tutor helped me to understand how to write simple functional programs	3,15
2	I found the high-level hints about how to solve a programming problem useful	3,43
3	I found the hints about the next step to take useful	3,05
4	The step-size of the tutor corresponded to my intuition	2,85
5	I found the possibility to see the complete solution useful	4,25
6	The worked-out solutions helped me to understand how to construct programs	3,55
7	The feedback texts are easy to understand	3,25
8	The kind of exercises offered are suitable for a first functional programming course	3,90

Table 6.1: Questionnaire: questions and scores.

are related and indicate how satisfied a student is with the tutor. The last question addresses how students value the difficulty of the offered exercises.

The goal of the experiment is to analyse if students appreciate our approach, such as giving feedback on intermediate answers. The experiment does not check whether or not the tutor is more effective or efficient from a learning point of view. We hope to study this in the future.

Reflection on the scores The scoring shows that the students particularly like the worked-out solution feedback. However, we don't know whether the students like the stepwise construction of a program, or if they are just interested in the final answer. The kind of exercises are as expected by the students. The results also show that the step-size used by the tutor does not correspond to the intuition of the student. We noticed this already during the experiment. The students often took larger steps than the tutor was able to handle.

The average of the first seven question gives an overall score of the tutor of 3,4 out of 5. This is maybe sufficient, but there clearly is room for improvement.

6.5.1 Evaluation

In addition to questions about the usage of the tutor, the questionnaire contained a number of general questions, such as

1. We offer the feedback services: such as giving a hint and diagnosing the submitted program. Do you think we should offer more or different feedback services?

2. Do you have any other remarks, concerns, or ideas about our programming tutor?

The answers from the students to the first question indicate that the current services are adequate. We received some interesting suggestions on how to improve our tutor in response to the second open question. The remarks that appear most are:

- Some solutions are not recognised by the tutor
- The response of the tutor is sometimes too slow

The first remark might indicate that a student thinks his or her own solution is correct, but our tutor doesn't accept it because it is incorrect or contains imperfections, such as being inefficient. During the labs several students blamed the tutor for not accepting their incorrect solutions. Nevertheless, these remarks address the fact that we cannot give a judgement when a student deviates from a path towards one of the model solutions. There are three possible reasons why our tutor thinks that a student deviates from a path towards a model solution. If a student program is incorrect, we should be able to detect and report this by giving a counter example[5]. If a student solution is correct, and uses desirable programming techniques, the set of model solutions should be extended. In the third case the tutor receives a functionally correct student program that contains some, possibly minor, imperfections, for example, a clumsy way of calculating the length of a list xs: $length\ (x : xs) - 1$. The tutor cannot conclude that a student program contains imperfections when it passes the tests but deviates from the strategy, so it cannot give a definitive judgement. However, after using an exercise in the tutor for a while, and updating the tutor whenever we find an improvement, it is likely that the set of model solutions is complete, and therefore unlikely that a student comes up with a new model solution. Therefore, in this particular case we can give feedback that a student program probably has some undesired properties. We have used our approach for assessment of functional programming exercises (see Chapter 11), in which we could recognise almost 90% of the correct solutions based on only five model solutions. All of the other 10% of the correct solutions had some imperfections.

The second remark is related to the step-size supported by the tutor. When a student takes a large step, the tutor has to check many possibilities, due to the flexibility that our tutor offers. We solved this problem by introducing a special

[5] At the time of the experiment our programming tutor was not able to use testing.

search mode when recognising large steps. In Chapter 10 we explain how we have implemented this search mode.

In addition to the above experiment, we also questioned a number of functional programming experts from the IFIP WG 2.1 group[6] and student participants of the Central European Functional Programming (CEFP 2011) summer school. We asked for input about some of the design choices we made in our tutor, such as giving hints in three levels of increasing specificity. Both the experts as well as the students support most of the choices we made. The main suggestion we got for adding extra services/functionality was to give concrete counterexamples using testing for semantically incorrect solutions. This suggestion corresponds to our own interpretation of the results from the experiment.

[6] `http://www.cs.uu.nl/wiki/bin/view/IFIP21/WebHome`

H-99 #	Name	Description
1	last	Write a function that selects the last element of a list.
2	butlast	Write a function that selects the but last element of a list (with a minimal length of 2).
3	elementat	Write a function that finds the *n*-th element of a list. The first element in the list is number 1.
4	length	Write a function that calculates the number of elements a list contains.
5	reverse	Write a function that reverses a list.
6	palindrome	Write a function that finds out whether a list is a palindrome. A palindrome can be read forward or backward (e.g. xamax).
7	concat	Write a function that flattens a list.
8	compress	Write a function that eliminates consecutive duplicates of list elements.
9	pack	Write a function that groups consecutive duplicates of list elements into sublists. If a list contains repeated elements they should be placed in separate sublists.
10	encode	Write a function that calculates the so-called 'run-length encoding' of a list.
14	dupli	Duplicate the elements of a list.
15	repli	Replicate the elements of a list a given number of times.
16	dropevery	Drop every *n*-th element from a list.
17	split	Split a list into two parts; the length of the first part is given.
18	slice	Extract a slice from a list. Given two indices, *i* and *k*, the slice is the list containing the elements between the *i*-th and *k*-th element of the original list (both limits included). Start counting the elements with 1.
19	rotate	Rotate a list n places to the left.
20	removeat	Remove the *n*-th element from a list.
22	range	Write a function which enumerates all numbers contained in a given range.

Figure 6.2: Offered programming exercises.

7 SPECIFYING PROGRAMMING EXERCISES

Anderson et al. (1995) claim that one of the reasons why programming tutors are not used much, is the of lack of teacher adaptability. It is often quite hard for teachers to adapt or add programming exercises to a tutor, and to adapt the feedback given by a tutor. An important goal of ASK-ELLE is to allow as much flexibility as possible for both teachers and students. Adding an exercise to our tutor is relatively easy. A teacher can specify her own exercises by giving a set of model solutions for a problem. Based on these model solutions our tutor generates feedback. A teacher can adapt feedback by *annotating* model solutions. A student may use her own names for functions and variables, and may use different, but equivalent, language constructs. In this chapter we show how we use annotated model solutions from a teacher to give feedback to a student in ASK-ELLE. This requires translating annotated model solutions to a form that we can use to track intermediate student steps.

7.1 Configuration

By annotating a model solution, a teacher adds feedback to the solution. This feedback only applies to the model solution in which it is specified. If a student implements a program based on that particular model solution, she receives that feedback when asking for support. Some feedback, however, concerns the exercise as a whole instead of a single model solution. For example, a description of the

exercise or an example result of an application of the program to be implemented. In addition to annotated model solutions, the teacher should supply a *configuration file* with meta-data about the programming exercise. A configuration file is specified using XML (extensible markup language) notation, and should adhere to the following DTD (document type definition):

```
<!DOCTYPE exercise [
  <!ELEMENT exercise (description, classes?)>
  <!ATTLIST exercise id       CDATA #REQUIRED
                     function CDATA #REQUIRED
                     type     CDATA #REQUIRED>
  <!ELEMENT description (#PCDATA)>
  <!ELEMENT classes     (class+)>
  <!ELEMENT class       (#PCDATA)>
]>
```

Here is an example of a configuration file[1]:

```
<?xml version="1.0"?>
<!DOCTYPE exercise SYSTEM "exercise.dtd">
<exercise id="reverseID" function="reverse" type="[a] -> [a]">
  <description>
    Write a function that reverses a list: reverse :: [a] -> [a].
    For example:

      Data.List> reverse "A man, a plan, a canal, panama!"
      "!amanap ,lanac a ,nalp a ,nam A"

      Data.List> reverse [1,2,3,4]
      [4,3,2,1]
  </description>
  <classes>
    <class>list.manipulation</class>
    <class>CS.FP.2012</class>
  </classes>
</exercise>
```

Most of the elements, and attributes are self-explanatory. The `function` attribute is used to identify the main function of the programming exercise, which should be of the type defined using the `type` attribute. The `description` element is displayed when a student selects a programming exercise. Furthermore, to support managing exercises, they can be arranged in *classes*. Using a class a teacher groups together exercises, for example for practising list problems, collecting exercises of the same difficulty, or exercises from a particular textbook.

[1] Note that the ampersand character (&) and the left angle bracket (<) must not appear in their literal form, and need to be escaped.

7.2 Feedback scripts for programming exercises

Section 5.3 introduces feedback scripts. These scripts are used to translate the results of a feedback analysis to a textual feedback message. The programming tutor also uses these feedback scripts. Every refinement rule has a corresponding feedback message. For example, the refinement rule that introduces the constructor *Nothing* has the following entry in a feedback script:

```
text con.nothing.20 = Introduce the constructor Nothing
```

Our domain reasoners require that the name of a rule is unique. A refinement rule is suffixed with a number to make it unique. In Section 8.1 we explain how refinement rules are constructed. A feedback script for a programming exercise is automatically generated. The resulting scripts can be changed by a teacher. Another example is the introduction of a variable:

```
text var.x.23 = Introduce the variable @name_x
```

The `var.x.23` refinement rule introduces a variable name. The name of the variable in the feedback text declaration is a reference `@name_x` to another feedback text. The value of `@name_x` is inserted at run-time, so that we can give feedback using the names introduced by a student. For example, the following code may be injected at runtime:

```
text name_x = y
```

Furthermore, every library function has its own feedback text. For example:

```
string unfoldrText = Use the unfoldr function
```

These texts can also be adjusted by a teacher.

7.3 Annotating model solutions

Section 6.4 shows an example interactive session with the tutor, in which a student develops a program for calculating the reverse of a list. This section shows what a teacher has to do to get this behaviour from the tutor, along with more feedback examples. A teacher adds a programming exercise to the tutor by specifying model solutions. For the example of reversing a list, we have specified three model solutions. The first is defined in terms of the higher-order function *foldl*:

$\{-\#$ `DESC` *Use the prelude function foldl.* $\#-\}$
reverse $=$

```
{−# FEEDBACK foldl takes an operator and a base value as argument. #−}
  (foldl {−# FEEDBACK Use flip and (:). #−}(flip (:)) [])
```

The second model solution uses an accumulating parameter:

```
{−# DESC Introduce a helper function that uses an accumulating parameter. #−}
reverse = reverse' []
  where
    reverse' acc []     = acc
    reverse' acc (x:xs) = reverse' (x:acc) xs
```

The third model solution is a explicit recursive definition of *reverse*:

```
{−# DESC Use explicit recursion. #−}
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Since {-... -} is used for multi-line comments in Haskell, annotated solutions are valid Haskell programs.

A teacher may annotate solutions to fine-tune the generated feedback. We distinguish two types of annotations: location specific and global annotations. Location specific annotations, such as the FEEDBACK annotation, target a particular expression in the model solution. To obtain this location information, and the attached feedback, we have extended the Helium compiler that we use for compiling the source code. The lexer, parser, and abstract syntax have been extended to incorporate feedback annotations. The global annotations are always placed in the header of the model solution source file. These annotations concern the entire model solution, such as the DESC annotation. Except for parsing these annotations, it is not necessary to further adapt the compiler.

The three solutions above are annotated with a high-level description of the approach used in the solution, using the following construction:

```
{−# DESC <description of model solution> #−}
```

The first hint in Section 6.4 gives the descriptions for the three model solutions for the reverse exercise.

In addition to the description annotation, the teacher has attached a specific feedback message to the right-hand side of the *reverse* definition, and to the operator argument of *foldl* in the first model solution, using {−# FEEDBACK ... #−}. When introducing the operator is one of the steps a student can take, and the student asks for help, the tutor will display the message specified. These feedback messages are organised in a hierarchy based on the AST of the model solution. This enables the teacher to configure the tutor to give feedback messages in an increasing level of

detail. This type of feedback is accessible via the *taskdescription* feedback service. Suppose a student chooses to implement the *reverse* function using the higher-order function *foldl*, and has started in the following way:

reverse = *foldl* • •

Here the student can refine the program at two locations: she can introduce an operator argument, or a base argument. If the student asks for help, the tutor responds with:

`foldl takes an operator and a base value as argument.`

A feedback message is 'active' for an element if it is an ancestor of that part of the AST. The above feedback message is given because the corresponding feedback annotation is active for a hole that can be refined by the student. There may be multiple feedback messages active for a particular hole. Since refining the operator argument is one of the possible next steps, the annotation on the operator is also active. The tutor can use this annotation to display a feedback message with more detail:

`Use` *flip* `and (:).`

Suppose the student asks for an even more detailed hint at this point. Because the most detailed feedback annotation has been given, the tutor displays the feedback message belonging to the rule that introduces the *flip* function:

`Introduce the function` *flip*`.`

If the student remains stuck, we give a bottom-out hint:

`Refine the program to:`

reverse = *foldl* (*flip* •) •

In addition to giving detailed feedback about a single step, the tutor can list the different steps that are allowed. In the given example, refining the hole for the base case is another step that the student can take. When asked for a hint about another step the tutor responds with:

`Introduce the empty list constructor []`.

Another way to adapt the feedback is by specifying an *alternative* implementation for a prelude function. For example, the specification below shows how to give an alternative implementation for the *map* prelude function:

$$\{-\#\ \texttt{ALT}\ map\ f\ xs = [\ f\ x\ \mid\ x \leftarrow xs\ ]\ \#-\}$$

Using this annotation we not only recognise the prelude definition, but also the alternative implementation given here. By adding an alternative a teacher expands the number of accepted solutions and therefore changes the way in which the tutor gives feedback. Alternatives give the teacher partial control over which program variants are allowed.

Besides adding alternatives to expand the number of accepted solutions, a teacher may want to emphasise one particular implementation method. For example, a teacher may want to enforce the use of higher-order functions and prohibit their explicit recursive definitions. Using the `MUSTUSE` annotation a teacher disables the recognition of the definition of a prelude function:

$$reverse = \{-\#\ \texttt{MUSTUSE}\ \#-\}foldl\ (flip\ (:))\ [\,]$$

A feedback annotation binds stronger than all other language constructs, i.e., it applies to the smallest possible expression. In the above example the student is not allowed to use the explicit recursive definition of *foldl* due to the `MUSTUSE` annotation. The student is allowed to use the definition of *flip*, because the scope of the `MUSTUSE` annotation is limited to the expression *foldl*. If a teacher wants to prohibit the use of definitions of library functions altogether, she can expand the scope of the annotation by placing parentheses:

$$reverse = \{-\#\ \texttt{MUSTUSE}\ \#-\}(foldl\ (flip\ (:))\ [\,])$$

Using the presented annotations a teacher can easily adapt which solutions are accepted, and fine-tune the generated feedback to her needs. We believe that making programming exercise adaptable by means of annotated model solutions lowers the threshold of using our functional programming tutor.

8 Constructing programming strategies

This chapter introduces strategies for functional programming, and shows how to construct such strategies. In Chapter 2 we introduce an embedded domain-specific language for specifying strategies. This strategy language is used for mathematical domains based upon rewrite rules. We also use this strategy language to specify strategies for functional programming exercises. The main difference with mathematical domains is that we use *refinement rules* instead of rewrite rules.

A strategy for defining a program consists of multiple steps. For example, developing a function implementing *reverse*:

$$
\begin{array}{l}
\mathit{reverse} = \mathit{reverse}'\ [\,] \\
\quad \textbf{where} \\
\quad\quad \mathit{reverse}'\ \mathit{acc}\ [\,] \quad\quad = \mathit{acc} \\
\quad\quad \mathit{reverse}'\ \mathit{acc}\ (x : xs) = \mathit{reverse}'\ (x : \mathit{acc})\ xs
\end{array}
$$

requires developing all components of the program, which in the case of an explicit recursive definition, such as for $\mathit{reverse}'$, consist of a case distinction between the empty list and the non-empty list, and a recursive call in the non-empty list case, amongst others. A programming strategy may also contain a choice between different (sequences of) steps. For example, we can choose to either use the higher-order function *foldl*, or a helper function with an accumulating parameter for *reverse*. Sometimes, the order in which the steps are performed is not relevant, as long as they are performed at some point. For example, the arguments of $\mathit{reverse}'$ can be refined in any order.

A programming strategy captures the multiple steps that a student needs to take to define a program. Based on these programming strategies we support the incremental development of a program. 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 yet to be defined; these undefined parts are called *holes*. Replacing these holes by 'more defined' parts are the steps a student takes when solving a programming problem. In our programming tutor the student replaces holes by typing in the part of the desired program at that point. An alternative is to let the student apply refinement rules that are offered by the tutor. The exact input method is immaterial for our approach.

A strategy for a functional program describes how a student may construct a functional program for a particular problem. Some well-known approaches to constructing correct programs are:

- specify a problem by means of pre- and post-conditions, and then calculate a program from the specification, or provide an implementation and prove that the implementation satisfies the specification (Hoare, 1969; Dijkstra, 1975),
- refine a program by means of refinement rules until an executable program is obtained (Back, 1987; Morgan, 1990),
- specify a problem by means of a simple but possibly very inefficient program, and transform it to an efficient program using semantics-preserving transformation rules (Bird, 1987; Meertens, 1986).

If we would use one of the first two approaches in a programming tutor that can give hints to students on how to proceed, we would have to automatically construct correctness proofs, a problem that is known to be hard. The last approach has been studied extensively, and several program transformation systems have been developed. However, our main goal is to refine instead of transform programs, since this better reflects the activities of beginning programmers. In our approach the set of accepted solutions is limited. Compared to the approaches mentioned above, our approach is more restrictive.

We use programming strategies to track the progress of a student solving a programming problem. We support the incremental construction, in a top-down fashion, of *model solutions*. We can detect deviations from the strategy, supply hints about what to do next, and analyse a step taken by a student.

The next section (Section 8.1) introduces refinement rules. The refinement rules need to be applied to a hole at a particular location in the incomplete program. In Section 8.2 we explain how we focus a refinement rule on a hole. We continue with showing an example functional programming strategy in Section 8.3 In the last section, Section 8.4, we describe how we derive a programming strategy from a model solution.

8.1 Refinement rules

The basic steps for constructing a solution for a programming task are program refinement rules. These rules typically replace an unknown part (•) by some expression. A 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. As in the Lisp tutor (Corbett et al., 1988), the refinement rules in our tutor model Haskell at the finest grain size that has functional meaning in Haskell.

The incremental development of the *reverse* function in the interactive session in Section 6.4 contains several program refinement rules: introduce a helper function, pattern match to determine whether or not a list is empty, and make a recursive call to the function. These rules are the basic elements in programming strategies. They are reusable in other programming exercises, and not specific for the *reverse* exercise.

We offer a number of refinement rules to students. For example:

•	⇒	$\lambda \bullet \rightarrow \bullet$	Introduce a lambda abstraction
•	⇒	**if** • **then** • **else** •	Introduce an **if-then-else**
•	⇒	v	Introduce the variable v

A hole represents a value, and such values may have different types. For example, a hole may represent an expression, as in all of the above examples, or a declaration, as in

•	⇒	$f \ \bullet = \bullet$	Introduce a function binding

A refinement rule replaces a hole with a value of its type, which possibly contains holes again. Internally, such a value is represented by a value of the datatype representing the abstract syntax of a part of a program. For example, the abstract syntax for expressions would typically contain the following constructors:

```
data Expr = Lambda Pattern Expr
          | If Expr Expr Expr
          | App Expr Expr
          | Var String
          | Case Expr Alts
          | ...
          | Hole
```

For declarations we have the following datatype:

```
data Decl = DFunBinds FunBinds
          | DPatBind Pat Rhs
```

```
  | ...
  | DHole
```

Every constructor has a corresponding rule function that takes the same number of arguments as the constructor, and returns a refinement rule for that constructor. So the rule function that produces a refinement rule for introducing an **if-then-else** expression takes three expression arguments. The arguments may be holes or terms containing holes. As another example, the rule function for building a refinement rule that introduces a lambda abstraction takes a pattern, and an expression (the body of the lambda expression) as arguments. As a final example, to make a refinement rule that introduces a variable, we use a rule function that takes the name of that variable (such as v in Figure 8.1) as an argument. The resulting refinement rule returns an expression that does not contain a hole anymore. These rule functions that build refinement rules encapsulate a constructor. For example, the rule function for a **case** is specified as follows:

```
caseRule :: Expr → [Alt] → Rule Expr
caseRule e alts = describe "Introduce case" $ ruleList "case" f
  where
    f _ = [Case e alts]
```

where *Case* is a constructor of the datatype *Expr*. The rule will replace any hole of type *Expr* with a **case** expression. The helper function f performs the replacement. Since any hole is replaced, the function f does not take its argument into consideration. The function *ruleList* translates a function, which returns a list of results, into a rule. The function *describe* attaches a description to the rule. We list some rule functions that are often used in Figure 8.1.

A refinement rule should be as 'small' as possible, in the sense that if we would further split such a rule, we cannot represent the corresponding program anymore, since we cannot build an abstract syntax tree for a program that is halfway completing an abstract-syntax tree construction. For example, the **if-then-else** expression cannot be split into an **if-then** and an **else** part in Haskell. In other words, a refinement rule refines a program on the level of the context-free syntax, and not on the level of tokens.

Holes are the central concept in our refinement rules. Where can they appear? Refinement rules refine:

- expressions, such as the $\bullet$ in $\lambda i \to \bullet$,
- declarations (the second $\bullet$ in $reverse = reverse' \bullet$ **where** $\bullet$),
- function bindings ($f\ [\,] = 0 \Rightarrow f\ [\,] = 0; f\ \bullet = \bullet$),
- alternatives (**case** xs **of** $[\,] \to 0 \Rightarrow$ **case** xs **of** $[\,] \to 0;\ \bullet \to \bullet$),

- patterns (**case** *xs* **of** $\bullet \rightarrow \bullet \Rightarrow$ **case** *xs* **of** $[\,] \rightarrow \bullet$).

We do not introduce refinement rules for other syntactic categories such as modules or classes, because these concepts hardly show up in our beginners' programs. Of course, this might change when the range of applications of the tutor is extended.

How do we come up with a set of refinement rules? A simple solution would be to take the context-free description of Haskell, and turn all productions into refinement rules. However, this general approach leads to all kinds of unnecessary and undesirable rules. For example, deriving a literal integer 42 using the context-free grammar for Haskell takes many steps, but a student would only see $\bullet \Rightarrow 42$. Our leading argument is that a refinement rule should be useful to a student, in the sense that it changes the way a program looks. Furthermore, the set of refinement rules should completely cover the programming language constructs we want the students to use, so that any program can be constructed using refinement rules. Complete coverage of a set of rewrite rules is verified by checking that for every datatype containing holes in the abstract syntax of programs (datatypes for expressions, declarations, function bindings, alternatives, and patterns, in our case), there exist refinement rules from a hole to any other constructor of the datatype.

Some refinement rules are performed silently, and are combined with one or more other refinement rules when generating steps, for example for a derivation or a hint. For instance, introducing an application in a Haskell program amounts to typing a space. We expect that few beginning students will view an application introduction as a step on its own, but instead always supply either a function or an argument name, or both. Our domain reasoner offers the possibility to annotate a rule that should be performed silently, by declaring it as a *minor* rule. We use these minor rules to increase the step size, and avoid showing steps like $\bullet \Rightarrow \bullet\ \bullet$. If application is declared to be a minor rule, a user can refine a hole to an application of a particular function, such as *unfoldr*, to one or more as yet unknown arguments.

At the moment, our tutor mainly supports the incremental construction of a program by means of refinement. However, it can also be used to rewrite a program, preserving its semantics, but changing some other aspects. For example, we might want to ask a student to change her program from using an explicit recursive definition of *reverse* to a definition using *foldl*, as in

$$
\begin{array}{l}
\quad \mathit{reverse} = \mathit{reverse}'\ [\,] \\
\quad\quad \textbf{where} \\
\quad\quad\quad \mathit{reverse}'\ \mathit{acc}\ [\,] \quad\quad = \mathit{acc} \\
\quad\quad\quad \mathit{reverse}'\ \mathit{acc}\ (x : \mathit{xs}) = \mathit{reverse}'\ (x : \mathit{acc})\ \mathit{xs} \\
\Rightarrow \quad \{\ \text{Definition of } \mathit{flip}\ \} \\
\quad \mathit{reverse} = \mathit{reverse}'\ [\,] \\
\quad\quad \textbf{where}
\end{array}
$$

Declarations

patBindRule $\bullet_1$ $\bullet_2$:	$\bullet$	$\Rightarrow$	$\bullet_1 = \bullet_2$
funBindsRule [$\bullet_1$, $\bullet_2$]:	$\bullet$	$\Rightarrow$	$\bullet_1$ $\bullet_2$

Function bindings

funBindRule f $\bullet_1$ $\bullet_2$:	$\bullet$	$\Rightarrow$	f $\bullet_1 = \bullet_2$

Expressions

varRule v:	$\bullet$	$\Rightarrow$	v
litRule l:	$\bullet$	$\Rightarrow$	l
appRule $\bullet_1$ $\bullet_2$:	$\bullet$	$\Rightarrow$	$\bullet_1$ $\bullet_2$
lambdaRule $\bullet_1$ $\bullet_2$:	$\bullet$	$\Rightarrow$	λ $\bullet_1 \rightarrow \bullet_2$
caseRule $\bullet_1$ $\bullet_2$:	$\bullet$	$\Rightarrow$	**case** $\bullet_1$ **of** $\bullet_2$

Alternatives

altRule $\bullet_1$ $\bullet_2$:	$\bullet$	$\Rightarrow$	$\bullet_1 \rightarrow \bullet_2$

Patterns

pVarRule v:	$\bullet$	$\Rightarrow$	v
pWildcardRule:	$\bullet$	$\Rightarrow$	_

Figure 8.1: Some refinement rules for functional programming in Haskell

$$
\begin{aligned}
&\mathit{reverse}'\ \mathit{acc}\ [\,] &&= \mathit{acc}\\
&\mathit{reverse}'\ \mathit{acc}\ (x : \mathit{xs}) &&= \mathit{reverse}'\ (\mathit{flip}\ (:)\ \mathit{acc}\ x)\ \mathit{xs}
\end{aligned}
$$

$\Rightarrow$ { Definition of *foldl* }

$$
\begin{aligned}
&\mathit{reverse} = \mathit{reverse}'\ [\,]\\
&\quad \textbf{where}\\
&\qquad \mathit{reverse}'\ \mathit{acc} = \mathit{foldl}\ (\mathit{flip}\ (:))\ \mathit{acc}
\end{aligned}
$$

$\Rightarrow$ { Inline and β-reduce }

$$\mathit{reverse} = \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,]$$

8.2 Focusing refinement rules

In our tutor, a program is constructed incrementally, in a top-down fashion. When starting the construction of a program there usually is a single hole. During the

development, refinement rules introduce and refine many holes. For example, the *app* refinement rule introduces two new holes: one for an expression that is of a function type, and one for an expression that is the argument of that function. When used in a strategy for developing a particular program, a refinement rule always targets a particular location in the program. For example, the refinement rule that introduces the second argument expression in a *foldl* application cannot be applied to an arbitrary expression hole, but should be applied at exactly the location where the argument, which introduces the base value, is needed in the program. In the next example this is the second expression hole (counted from left to right):

$$foldl\ (flip\ \bullet\ )\ \bullet \Rightarrow foldl\ (flip\ \bullet\ )\ base\_value$$

A refinement rule needs to know which particular location to target in the program. That is, a refinement rule needs to focus on the right hole. A rewrite rule, on the other hand, may be applicable to more than one location in the AST.

When defining a strategy for developing a functional program, we need to relate the new holes that are introduced by the refinement rules to the rules (or strategies) that are going to refine them. For example, the rules that refine the holes introduced by the *app* refinement rule need to be targeted to the location of those holes. Recall that our refinement rules just encapsulate a constructor of an abstract syntax datatype in a rule. For instance, the *app* rule encapsulates the *App* constructor from the *Expr* datatype in an expression refinement rule:

```
appRule :: Expr → Expr → Rule Expr
appRule f x = describe "Introduce application" $ ruleList "app" g
  where
    g _ = [App f x]
```

The *appRule* refinement rule applies *App* to two holes of type *Expr*. The first might for example be refined by the *var* :: *String* → *Expr* refinement rule that introduces a prelude function, as in *var* `"length"`. The *var* refinement rule and the first hole should be connected to each other. We achieve this connection by giving a hole an *identifier* and focusing a refinement rule to be only applicable to a hole with that particular identifier. We extend the *Hole* constructors of the various abstract syntax datatypes with an identifier field. For example, the *Hole* constructor of the *Expr* data type is extended as follows:

```
type HoleID = Int
data Expr   = Hole HoleID | ...
```

Recall that rules can operate on a pair of a zipper and an environment, see Section 2.3.9. We target a refinement rule to be only applicable to a particular hole by

using the focus of the zipper. The combination of a zipper and an environment is called a *context*. We can lift rules to work on a context. For example the rule that introduces a variable:

```
varRule :: Name → Rule Expr
varRule n = describe ("Introduce the variable " ++ show n) $ ruleList "var" f
  where
    f _ = [Var n]
```

can be lifted as follows:

```
varRule' :: Name → Rule (Context Expr)
varRule' = liftToContext ∘ varRule
```

The function *liftToContext* lifts a rule so that it is applied to the expression in focus.

We put the focus on the to be refined hole, before we apply the corresponding refinement rule. For example, before applying the refinement rule that introduces the right-hand side of the *reverse'* helper function, we put the focus on the hole at the right-hand side:

```
reverse = •2
  where
    reverse' acc [] = ⟦ •1 ⟧
```

We use the navigation rules to put the focus on a particular hole. So, given a hole and a refinement rule we can focus the rule as follows:

```
focusRule :: Eq a ⇒ a → Rule (Context a) → Strategy (Context a)
focusRule hole rule = ⟨focusOn hole <⋆> rule⟩
```

Where *focusOn* puts the focus on the given hole, using navigation rules. We omit the definition of this function. The placement of the focus is put in sequence with the refinement rule. The sequence of putting the focus right and applying the rule is put in an atomic block, because interleaving with other steps may change the focus. For example, the following code makes the rule *varRule'* only applicable to a hole with identifier 1:

```
var_acc :: Strategy (Context Expr)
var_acc = focusRule •1 (varRule' "acc")
```

This strategy can be used to refine the right-hand side of the helper function *reverse'* given earlier. Applying this strategy results in:

```
reverse = •2
  where
    reverse' acc [] = ⟦ •1 ⟧
```

$\Rightarrow$ { Introduce the variable *acc* }

$$
\begin{aligned}
&\mathit{reverse} = \bullet_2 \\
&\quad \textbf{where} \\
&\qquad \mathit{reverse}'\ \mathit{acc}\ [\,] = [\![\mathit{acc}]\!]
\end{aligned}
$$

The approach of finding holes by means of their identifier requires that these identifiers are unique. The numbering of holes takes place in a state monad.

A refinement rule is only applicable when the holes it refines are present in the AST. For instance, in the programming strategy for *reverse*′, the refinement rule that introduces the right-hand side of *reverse*′ is applied *before* the *varRule*′ `"acc"` rule. We use the sequence combinator to enforce the order in which the refinements have to take place. When sequencing two programming substrategies, we ensure that the first substrategy refines to a term that can be refined by the second substrategy. Determining the order of refinement rules is defined during the derivation of a programming strategy.

Relating holes and refinement rules using holes with identifiers has some consequences for the implementation of our functional programming domain reasoner. For the other domains we have developed, the domain reasoners operate on the term that has been submitted by the student. In the functional programming domain reasoner, however, we get an AST with holes *without* identifiers when we parse a student submission, because the concrete syntax does not contain hole identifiers. To compare a student program to the expected programs by the strategy, we need to repair the holes in the student submission so that they have the correct identifiers. To do so, we *reconstruct* the student term. Recall that we maintain information about what part of the strategy has already been solved by the student, since it is communicated back and forth between the front- and back-end, see 6.1.1. We reconstruct the AST of the student term using this part of the strategy, by applying it to the start term.

8.3 Strategies in functional programming

For any programming problem, there are many solutions. Some of these solutions are syntactical variants of each other, but other solutions implement different ideas to solve a problem. We specify a strategy for solving a functional programming problem by means of model solutions for that problem. We can automatically *derive* a strategy from a model solution. The strategies for the various model solutions are then combined into a single strategy using the choice combinator. So, for the *reverse* model solutions from Section 6.4 we would get a single strategy combining the three strategies for the model solutions

We derive a programming strategy from a model solution by inspecting the abstract syntax tree of a model solution, and matching the refinement rules with the AST. This is a tree matching algorithm, which yields a strategy. In Section 8.4 we describe the process of deriving programming strategies. For example, here is a strategy that is derived from the definition of *reverse* in terms of *foldl*:

```
    patBind
<*> pVar "reverse"
<*> app <*> var "foldl"
        <*> (   (paren <*> app <*> var "flip"
                                   <*> infixApp <*> con "(:)"
                )
          <%> con "[]"
            )
```

There are several things to note about this strategy. The ordering of the rules by means of the sequence combinator <*> indicates that this strategy for defining *reverse* recognises the top-down construction of *reverse*. Since we use the interleave combinator <%> to separate the arguments to *foldl*, a student can develop the arguments to *foldl* in any order. This strategy uses three rules we did not introduce in the previous section, namely *infixApp*, which introduces an infix application, *con*, which introduces a constructor of a datatype, and *paren*. The rule *paren* ensures that the first argument of *foldl* is in between parentheses. The hole introduced by this rule is filled by means of the strategy that introduces *flip* (:). The rule *paren* is minor, so we don't require a student to explicitly introduce parentheses in a single step, but recognise it together with the introduction of the function *flip*. Since rules correspond to abstract syntax tree constructors, this shows that our abstract syntax also contains constructors that represent parts of the program that correspond to concrete syntax, such as parentheses. This way we can also guide a student in the concrete syntax of a program. However, we might also leave concrete syntax guidance to the parsing and type-checking phase of Helium.

If the above strategy would be the complete strategy for defining *reverse*, then a student would only be allowed to construct exactly this definition. This would almost always be too restrictive. Therefore, we would typically use a strategy that combines a set of model solutions. However, our approach necessarily limits the solutions accepted by the tutor: a solution that uses an approach fundamentally different from the specified model solutions will not be recognised by the tutor. Depending on the model solutions provided, this might be a severe restriction. However, in experiments with lab exercises in a first-year functional programming course (Gerdes et al., 2012b), we found that our tutor recognises almost 90% of the correct student programs by means of a limited set of model solutions. The remaining 10% of correct solutions were solutions 'with a smell': correct, but using

constructs we would never use in a model solution. We expect that restricting the possible solutions to a programming problem is feasible for beginning programmers. It is rather uncommon that a beginning programmer develops a new model solution for a beginners' problem.

8.4 Deriving programming strategies

When solving a programming exercise in our tutor, a student needs to define a program that solves a programming problem. A programming strategy describes sequences of refinement steps: applying all the steps of such a sequence to a start term results in a solution for the programming problem. We could specify all allowed sequences that solve a programming task by hand. This is, however, a labourious and error prone process. For example, when specifying a programming strategy by hand, it is necessary to manually connect the refinement rules to their corresponding hole (by supplying the correct identifiers). It is less labour intensive to automatically derive a programming strategy from a model solution. 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 she 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. We combine multiple model solutions with the strategy combinator for choice.

During the derivation of a programming strategy we inspect the AST of a model solution, and map each language construct to a refinement rule using its corresponding rule function. For example, when deriving an application, we use the corresponding rule function *appRule*. The refinement rule returned by *appRule* is combined with the derived strategies of the components of an application, i.e., a function and its arguments. When deriving a programming strategy, we specialise all rules to be applicable to a particular hole. For example, for the derivation of an application we first generate new holes that are passed as argument to the *appRule* rule function. Then, the rule functions for the arguments of the application are specialised to these holes.

By using the interleave combinator in the strategy for certain combinations of language constructs, we gain some flexibility in the sequences of refinement steps that we accept. For example, the two definitions for the two function bindings for *reverse'* can appear in any order since this does not change the meaning of the function.

In Chapter 7 we show that a teacher can annotate model solutions to fine-tune the feedback generation. During the derivation of a programming strategy we have to take these annotations into account. Our abstract syntax, and the Helium compiler, have been extended to accommodate these annotations:

```
data Expr = App Expr Expr
          | Hole HoleID
          | ...
          | Feedback String Expr
          | MustUse Expr
          | Alt Decl
```

These annotations are translated to the right strategy constructs. For example, the *Feedback* annotation is translated to a *label* that contains the feedback message. Consider the following contrived example:

```
($) f x = {-# FEEDBACK Apply f to x #-}(f x)
```

This program would be translated to the following strategy:

```
    funBinds
<*> funBind "$"
<*> (pVar "f" <%> pVar "x")
<*> label "Apply f to x" (app <*> (var "f" <%> var "x"))
```

In this strategy a label has been placed on the substrategy for recognising the right-hand side of the function. If a student asks for help when defining the right-hand side, the tutor will give a feedback message based on this label.

8.4.1 Strategies for library functions

To recognise as many syntactic variants as possible of (a part of) a solution to a programming problem, we derive special strategies for recognising occurrences of library functions in programs. When deriving a library function we not only derive the usage of that function, but also its definition. For example, the strategy derived for *flip* not only recognises *flip* itself, but also its definition, which can be considered an inlined and β-reduced version of *flip*. For example, the following function:

```
snoc = flip (:)
```

is translated into:

```
    patBind
<*> pVar "snoc"
<*> app     <*> var "flip" <*> con ":"
<|> lambda <*> pVar "x" <*> pVar "y"
           <*> infixApp <*> con ":" <*> (var "y" <%> var "x")
```

The variable names x and y, used in the lambda-abstraction, need to be fresh and should not appear free in the argument of *flip*, in order to avoid variable capturing. The above strategy recognises both *flip* $(:)$ itself, and the β-reduced, infix constructor, form $\lambda xs\ x \to x : xs$.

It is important to specify model solutions for exercises using abstractions available in Haskell's prelude like *foldl*, *foldr*, *flip*, etc, if applicable. If a student would use these abstractions in a solution, where a model solution wouldn't, then the student's program wouldn't be accepted. For any function in the prelude, a student may either use the function name itself in her program, such as for example $(\circ)$, or its implementation, such as $\lambda f\ g\ x \to f\ (g\ x)$.

In addition to recognising the definition of a library function, a teacher can specify *alternative* definitions of a library function, see Section 7.3. This increases the number of accepted variants even further. A student program that uses this alternative definition will also be recognised. For example,

```
{−# ALT foldl op e == foldr (flip op) e . reverse #−}
reverse = foldl (flip (:)) []
```

So, if a function that is specified by means of a *foldl*, using this alternative we also accept an implementation in terms of a *foldr* together with *reverse*.

Many programming tasks involving lists use higher-order functions, such as *foldr*. 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 using *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. In this case, a teacher may want to prohibit the use of the definition of higher-order functions. In Section 7.3 we have introduced the *MUSTUSE* annotation, which prevents the recognition of the definition of a library function. If the derivation process encounters a *MUSTUSE* annotation, it no longer derives strategies for library functions, which recognise the definition of a library function as well as its usage.

A strategy cannot capture all variants of a program that a student introduces. For example, the fact that a student uses different names for variables is hard, if not impossible, to express in a strategy. However, we do want to give a student the possibility to use her own variable names. We use *normalisation* to handle such kinds of variations. Chapter 9 describes the normalisation process.

9 A CANONICAL FORM FOR HASKELL PROGRAMS

One of the advantages of using strategies for recognising student programs is that accepted programs are guaranteed to be equivalent to a model solution. However, most strategies are rather strict, and might reject programs that are equivalent but have some differences. For example, consider the following two programs:

$$f\ x = x - 1$$

and

$$f\ y = (-)\ y\ 1$$

Although the two programs are different in appearance, they are equivalent.

Not all differences can or should be captured in a strategy, because they are standard transformations of a program, independent of a particular strategy. For example, sometimes a student doesn't explicitly specify all arguments to a function, and for that purpose we use η-reduction when analysing a student program:

$$\lambda x \to f\ x \Rightarrow f$$

Using normalisation, we want to recognise as many syntactical variants of Haskell programs as possible. We are not so much interested in the exact canonical form, as long as it can be used to efficiently compare two terms for equality. To verify that a program submitted by a student follows a strategy, we apply all rules allowed by

the strategy to the previous submission of the student, normalise the programs thus obtained, and compare each of these programs against the normalised submitted student program.

Normalisation uses various program transformations to reach a canonical form of a Haskell program. The program transformations used in the normalisation process are based on the *lambda calculus*. The lambda calculus is at the core of Haskell, and its reduction rules form the heart of the evaluation machinery. We use amongst others α-renaming, β- and η-reduction, inlining, and desugaring program transformations. In general, comparing two lambda terms for equality is undecidable. However, we can decide equivalence for many terms using these program transformations. Our equivalence checker may reject equivalent programs and hence we may have false-negatives. Until now we have not found this to be a problem in practice. If programs are found to be equivalent, they are semantically equivalent, so we do not obtain false-positives.

Our normalisation procedure starts with α-renaming, which gives all bound variables a fresh name. Then it desugars the program, restricting the syntax to a (core) subset of the full abstract syntax. The next step removes constant arguments and inlines local definitions, which makes some β-reductions possible. Finally, normalisation performs β- and η-reductions in applicative order (leftmost-innermost) and normalises a program to β-normal form. For example, consider the following model solution for *reverse*:

```
reverse = rev []
  where
    rev acc []       = acc
    rev acc (x : xs) = rev (x : acc) xs
```

The next code shows the *reverse* model solution after normalisation:

```
reverse =
  let b = λy1 → λy2 → case (y1, y2) of
            (c, []) → c
            (d, e : f) → b ((:) e d) f
  in  b []
```

Although the programs are the same, they are syntactically quite different.

9.1 Program transformations

In this section we show some of the program transformations and discuss the limitations of our normalisation.

9.1.1 Desugaring

Desugaring removes syntactic sugar from a program. Syntactic sugar is usually introduced to conveniently write programs, such as writing $\lambda x\ y \rightarrow ....$ for $\lambda x \rightarrow \lambda y \rightarrow ....$ Syntactic sugar does not change the semantics of a program. However, if we want to compare a (possibly partially complete) student program syntactically against model solutions we want to ignore syntactic sugar. Desugaring consists of several program transformations such as removing superfluous parentheses[1], rewriting a **where** expression to a **let** expression whenever possible, moving the arguments of a function binding to a lambda abstraction (e.g., $f\ x = y \Rightarrow f = \lambda x \rightarrow y$), and rewriting infix operators to (prefix) functions. The following derivation shows how a somewhat contrived example is desugared:

$$\mathit{reverse} = \mathit{foldl}\ f\ [\,]\ \textbf{where}\ f\ x\ y = y : x$$

$\Rightarrow$ { **where** to **let** }

$$\mathit{reverse} = \textbf{let}\ f\ x\ y = y : x\ \textbf{in}\ \mathit{foldl}\ f\ [\,]$$

$\Rightarrow$ { Infix operators to (prefix) functions }

$$\mathit{reverse} = \textbf{let}\ f\ x\ y = (:)\ y\ x\ \textbf{in}\ \mathit{foldl}\ f\ [\,]$$

$\Rightarrow$ { Function bindings to lambda abstractions }

$$\mathit{reverse} = \textbf{let}\ f = \lambda x \rightarrow y \rightarrow (:)\ y\ x\ \textbf{in}\ \mathit{foldl}\ f\ [\,]$$

In the following paragraph on inlining we will see how the declaration of f is inlined in the *foldl*-expression.

9.1.2 Inlining

Inlining replaces a call to a user-defined function by its body. We perform inlining to make β-reduction possible. Inlining is performed together with dead-code elimination, because the inlining procedure may introduce dead-code. Dead-code elimination is a program transformation that removes code that will never be evaluated. For example:

$$\mathit{reverse} = \textbf{let}\ f = \lambda x \rightarrow \lambda y \rightarrow (:)\ y\ x\ \textbf{in}\ \mathit{foldl}\ f\ [\,]$$

$\Rightarrow$ { Inline }

$$\mathit{reverse} = \textbf{let}\ f = \lambda x \rightarrow \lambda y \rightarrow (:)\ y\ x\ \textbf{in}\ \mathit{foldl}\ (\lambda x \rightarrow \lambda y \rightarrow (:)\ y\ x)\ [\,]$$

[1] Helium has AST constructors for parentheses.

$\Rightarrow$ { Dead-code elimination }

$$reverse = foldl\ (\lambda x \rightarrow \lambda y \rightarrow (:)\ y\ x)\ [\,]$$

Note that we do not perform these two transformations on incomplete programs, e.g., programs that contain holes. The reason for this limitation is that dead-code elimination may remove refinements that are introduced by a student. Consider the following intermediate program:

$$reverse = \bullet$$

Suppose a student submits the following intermediate program:

$$\begin{array}{l} reverse = \bullet \\ \quad \textbf{where} \\ \quad\quad reverse'\ \bullet\ \bullet = \bullet \end{array}$$

Applying the dead-code elimination transformation to the above intermediate program results in the same program as started with. As a consequence, we would not be able to detect the student step.

We do not inline recursive functions. Recursive functions are rewritten in terms of *fix*, which does not get β-reduced to avoid non-termination.

9.1.3 Constant arguments

An argument is constant if it is passed unchanged to *all* recursive function calls. Consider the following naive implementation of the higher-order function *foldr*:

$$\begin{array}{l} foldr\ op\ b\ [\,] \quad\quad = b \\ foldr\ op\ b\ (x : xs) = x\ `op`\ foldr\ op\ b\ xs \end{array}$$

This implementation has two constant arguments: *op* and *b*. A better implementation is:

$$\begin{array}{l} foldr\ op\ b = f \\ \quad \textbf{where} \quad f\ [\,] \quad\quad = b \\ \quad\quad\quad\quad\ \ f\ (x : xs) = x\ `op`\ f\ xs \end{array}$$

The above definition is the definition of *foldr* in the Haskell prelude. Compilers often optimise such constant arguments away, to save space and increase speed. Our goal with this transformation is not to optimise programs, but instead to increase the number of possibilities to apply β-reduction. Note that we do not inline recursive functions. However, the constant arguments of a recursive function

can be β-reduced. The optimisation of a recursive function with constant arguments, such as the naive *foldr* function, separates the recursive (f in the example) from the non-recursive part of a function. Therefore, only after optimising constant arguments away does it help to inline the function. The optimised version of *foldr* will be inlined, but the recursive helper function f will not be inlined.

9.1.4 Lambda calculus reductions

We use α-conversion to rename bound variables. To check that a program is syntactically equivalent to a model solution, we α-convert both the submitted student program as well as the model solution. α-conversion ensures that all variable names are unique. This simplifies the implementation of other program transformation steps, such as β-reduction, because substitutions become capture avoiding.

η-reduction reduces a program to its η-short form, trying to remove as many lambda abstractions as possible. η-reduction replaces $\lambda x \to f\ x$ by f if x does not appear free in f.

Finally, we apply β-reduction. β-reduction takes the application of a lambda abstraction to an argument, and substitutes the argument for the lambda-abstracted variable: $(\lambda x \to expr)\ y \Rightarrow_\beta expr[x := y]$. The substitution $[x := y]$ replaces all free occurrences of the variable x by the expression y. For example, using β-reduction we get:

$$(\lambda f\ x\ y \to f\ y\ x)\ (:) \Rightarrow \lambda x\ y \to (:)\ y\ x$$

9.2 Discussion

When recognising a student program we want to be as flexible as possible and ignore some differences between the student program and a corresponding model solution. This flexibility is implemented at two places: in the programming strategy, and the normalisation procedure. Why do we implement this flexibility in two places? Why do we need strategies at all? Does it suffice to only use normalisation to recognise student programs based on a set of model solutions? For example, instead of deriving strategies for library functions (as described in Section 8.4.1), which recognise also the definition of that function, we could add a program transformation to our normalisation procedure that rewrites the usage of a library function to its definition. The justification for the chosen implementation is that we cannot generate feedback based on the normalisation procedure. We can only generate feedback based on strategies. So, if we want to use a rewrite or refinement step for generating feedback, then it needs to be in the programming strategy.

Correctness of a normalisation procedure depends on several aspects (Filinski and Korsholm Rohde, 2004). A normalisation procedure is

- *sound* if the output term, if any, is β-equivalent to the input term,
- *standardising* if equivalent terms are mapped to the same result,
- *complete* if normalisation is defined for all terms that have normal forms.

We claim that our normalisation procedure is sound and complete but not standardising, but we have yet to prove this. The main reason for our normalisation procedure to be non-standardising is that we do not inline and β-reduce recursive functions. For example, while the terms *take* $3\ [1\,..]$ and $[1,2,3]$ are equivalent, the first will not be reduced by normalisation. Therefore, these terms have different normalisation results. We do not incorporate β-reduction of recursive function because this might lead to non-terminating normalisations.

Normalisation by evaluation (NBE) (Berger et al., 1998) is an alternative approach to normalisation. NBE evaluates a λ-term to its (denotational) semantics and then reifies the semantics to a λ-term in β-normal and η-long form. The difference with our, more traditional, approach to normalisation is that NBE appeals to the semantics (by evaluation) of a term to obtain a normal form. The main goal of NBE is to efficiently normalise a term. We are not so much interested in efficiency, but it may well be that NBE improves standardisation of normalisation.

10 A PROGRAMMING STRATEGY RECOGNISER

An important aspect of a programming tutor is that it offers sufficient freedom to students: a student should be able to use her own names, to use her own favourite programming style, her own refinement step-size, etc. We try to be as flexible as possible towards students while guiding the student towards a correct and elegant solution. We use special strategies to recognise definitions and alternatives of library functions. To ignore insignificant differences, such as different variable names, we use program normalisation. Another feature of our programming tutor is that we allow an arbitrary step size. The refinement rules in our tutor model Haskell at the finest grain size that has functional meaning in Haskell. Forcing a student to take small steps, by allowing only one refinement step at a time, would be a severe limitation of our programming tutor. We want to offer students the possibility to make larger steps than these small steps. This is challenging in the context of teacher annotated model solutions to program exercises.

Our tutor constructs a programming strategy from a set of model solutions. The strategy is interpreted as a recogniser that recognises program refinement steps of students. Chapter 3 describes the implementation of the recogniser for our strategy language. This chapter discusses how we adapt this recogniser, such that it can efficiently recognise multiple refinement steps.

Strategy recogniser. We interpret a strategy as a context-free grammar. The language generated by a strategy can be used to determine whether or not a sequence

of rules applied by a student follows a strategy. The sequence of rules should be a sentence in the language, or a prefix of a sentence, since we solve exercises incrementally. A recogniser for a context-free grammar recognises refinement steps that are applied to some initial term, usually the empty program. The current location within the strategy, the remaining strategy, at which the student has applied a refinement rule is maintained in a state. The recogniser uses this state information to give precise feedback. Using the information about the progress of student, we can calculate which steps are allowed next, and check whether or not a student deviates from a path towards a model solution.

10.1 Parallel top-down recogniser

The recogniser accepts intermediate (incomplete) solutions because it recognises prefixes. It cannot use backtracking, since this would imply that it suggests steps that do not lead to the intended solution, and hence guides students into the wrong direction. It follows that the recogniser needs to choose between the various model solutions on the basis of a single refinement step. This is problematic when multiple model solutions share a first step, i.e., when we encounter a left-factor in the strategies generated for the model solutions. Note that combining model solutions almost always leads to left-factors. The introduction of a declaration, and a function name is very often shared between the different model solutions. Consider the following, somewhat contrived, strategy:

$$\begin{aligned} \mathit{leftFactor} \;=\; & \mathit{label}\ \ell_1\ (\mathit{app} \mathbin{<\!\star\!>} \mathit{var}\ \texttt{"f"} \mathbin{<\!\star\!>} \mathit{var}\ \texttt{"x"}) \\ \mathbin{<|>}\ & \mathit{label}\ \ell_2\ (\mathit{app} \mathbin{<\!\star\!>} \mathit{var}\ \texttt{"g"} \mathbin{<\!\star\!>} \mathit{var}\ \texttt{"y"}) \end{aligned}$$

The two substrategies labelled ℓ_1 and ℓ_2 share a left-factor: the rewrite rule *app*. We should decide which substrategy to follow *after* recognising the application of *app*, but the requirement to choose based on a single refinement step does not allow for this. Committing to a choice after recognising that *app* has been applied is unfortunate, since it will force the student to follow the same substrategy. For example, if the recogniser chooses *var* `"f"` after the application of *app* and commits to the substrategy with label ℓ_1, and a student subsequently performs the *var* `"g"` step, we erroneously report that that step does not follow the strategy. The standard method of dealing with this problem is to apply left-factoring. Left-factoring is a grammar transformation that is useful when two productions for the same nonterminal start with the same sequence of terminal and/or nonterminal symbols. This transformation factors out the common part, called the left-factor, of such productions. In a strategy, the equivalent transformation factors out common sequences of rewrite rules from substrategies separated by the choice combinator. However, the presence of labels makes it impossible to use left-factoring, since

moving or merging labels leads to scrambling annotations of models solution, making it very hard if not impossible to give the intended hints. It is clear how to left-factor (major) rewrite rules, but how should we deal with labels, or minor rules in general? Pushing labels inside the choice combinator,

$$\begin{aligned} \mathit{leftFactor} = \mathit{app} \mathbin{<\!\star\!>} \quad & (\ \ \mathit{label}\ \ell_1\ (\mathit{var}\ \texttt{"f"} \mathbin{<\!\star\!>} \mathit{var}\ \texttt{"x"}) \\ & \mathbin{<|>}\ \mathit{label}\ \ell_2\ (\mathit{var}\ \texttt{"g"} \mathbin{<\!\star\!>} \mathit{var}\ \texttt{"y"})) \end{aligned}$$

or making a choice between the two labels breaks the relation between the label and the strategy. Labels are used to mark positions in a strategy, and have corresponding feedback texts, which very likely become inaccurate if labels are moved automatically. We need to defer committing to a particular path in the strategy.

To deal with left-factors, we conceptually fork the recogniser whenever we run into a left-factor. If any of these recognisers fails to recognise the student solution, we discard it. Thus we obtain a top-down variant of a parallel recogniser. Using a top-down parallel recogniser we allow a teacher to specify model solutions that have common components. In the implementation we do not start another recogniser process, but we extend the state, which the recogniser uses, to maintain a list of remaining strategies, instead of a single strategy. The relevant feedback services, such as *onefirst* and *diagnose* are adapted to so that they operate on a list of states.

To parse an ambiguous grammar we need a parallel parser. Tomita (1985) introduced parallel parsing for bottom-up parsers. Similar to Tomita, we parse in parallel. Different from Tomita, we perform parallel parsing top-down.

The strategy language has been used to describe how to solve exercises in many mathematical domains, such as solving quadratic equations, and differentiating functions. The strategies in these domains are less likely to be ambiguous, and a top-down recogniser for LL(1) grammars that are not left-recursive supports solving such exercises well.

10.2 Search space reduction

We have performed several experiments with ASK-ELLE, and asked students to evaluate the programming tutor, see Section 6.5. Students are generally positive about using the tutor; their main comment is that the tutor is of no help when performing many refinement steps in a single step. Some students even paste complete solutions in the tutor (which we might consider undesirable behaviour, but which we don't want to disallow). At the time of the experiments, the tutor could not recognise this. Recognising multiple steps is difficult. In an expression such as

$$
\begin{array}{l}
\mathit{reverse}\ \mathit{list} = \mathit{reverse}'\ \mathit{list}\ [\,] \\
\quad \textbf{where}\ \mathit{reverse}'\ \bullet \qquad\ \bullet\ = \bullet \\
\qquad\qquad \mathit{reverse}'\ (x : xs)\ \bullet\ = \mathit{reverse}'\ xs\ \bullet
\end{array}
$$

a student may refine any of the five holes, in any order. The derived strategy for this solution allows to interleave the refinement of the five holes. The number of interleavings grows factorially. That means that even for relatively small introductory programs the number of intermediate solutions may be huge. The use of standard strategies not only increases the number of accepted solutions, but also the number of possible interleavings. The experiment showed that it is not an option to check, by means of multiple calls to the *firsts* function, if a student submission is an element of the set of all possible intermediate solutions.

To efficiently check if a student submission is accepted by a strategy, we make some assumptions about how a student refines a program. For example, we assume that it is more likely that a student first finishes a particular part of the program, such as a function binding, then refining at arbitrary places. Consider the following incomplete definition of *map*′:

$$
\begin{array}{l}
\mathit{map}'\ \ \bullet_1\ \ \bullet_2 \qquad = \bullet_3 \\
\mathit{map}'\ f\ (\ \bullet_4\ :\ \bullet_5\ ) = \bullet_6
\end{array}
$$

The student has started to refine the second function binding, which pattern matches on a non-empty list. Instead of refining holes in the first function binding, she probably completes the second function binding first. Therefore, we use depth-first search to find matching solutions. We search the abstract syntax tree in a depth-first manner for a hole that can be refined. For example, if a refinement rule introduces two function bindings, then we take the first and try to refine it as far as possible before we start on the second function binding.

10.2.1 Pruning

We constrain the search space of intermediate answers to determine whether or not a student submission follows a strategy. First, we observe that the first steps of the different strategies for model solutions may be the same, but they diverge after a number of steps. For example, if a student submits

$$\mathit{reverse} = \mathit{foldl}\ (\mathit{flip}\ \bullet\ )\ \bullet$$

she follows the strategy of the model solution using *foldl*, and we do not generate the intermediate answers of the other model solutions. Since we use refinement rules, a student can no longer refine her program towards those model solutions. This reduces the number of interleavings significantly. We filter out these intermediate

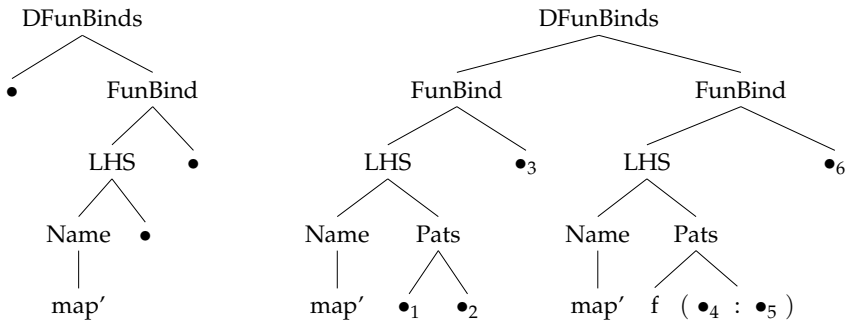


Figure 10.1: Example ASTs that overlap.

answers by determining whether or not the normalised abstract syntax trees of the model solution and the student submission overlap, where a hole (•) overlaps with any tree. Figure 10.1 shows an example of an intermediate answer to the *map′* function, which overlaps with the student submission.

10.2.2 A search mode for the interleave combinator

Even with pruning, the search space remains too large, due to the amount of possible interleavings. To reduce the number of interleavings, we observe that when recognising multiple steps, the *order* of refinements of holes that may be interleaved is irrelevant. Consider the *reverse* example from the previous subsection. For recognition it does not matter whether we first introduce the cons operator followed by the empty list constructor, or vice versa. Interleaving causes many duplicates in the set of intermediate answers. For example:

$$
\begin{aligned}
& \mathit{reverse} = \mathit{foldl}\ (\mathit{flip}\ \bullet\ )\ \bullet \\
\Rightarrow\ & \mathit{reverse} = \mathit{foldl}\ (\mathit{flip}\ (:))\ \bullet \\
\Rightarrow\ & \mathit{reverse} = \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,]
\end{aligned}
$$

or:

$$
\begin{aligned}
& \mathit{reverse} = \mathit{foldl}\ (\mathit{flip}\ \bullet\ )\ \bullet \\
\Rightarrow\ & \mathit{reverse} = \mathit{foldl}\ (\mathit{flip}\ \bullet\ )\ [\,] \\
\Rightarrow\ & \mathit{reverse} = \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,]
\end{aligned}
$$

We use the irrelevance of refinement order when recognising multiple steps by introducing a search mode for the interleave combinator. The *deepdiagnose* feedback service uses the search mode when diagnosing a student program, to check whether or not it can be accepted by the programming strategy.

The semantics of the original interleave combinator chooses between the left-interleave of both substrategies:

$$x \mathbin{<\%>} y \qquad = (x \mathbin{\%>} y) \mathbin{<|>} (y \mathbin{\%>} x)$$
$$(\langle a \rangle \mathbin{<\star>} x) \mathbin{\%>} y = a \mathbin{<\star>} (x \mathbin{<\%>} y)$$

The search mode for interleave changes the semantics of $<\%>$. It chooses between the left-interleave of the substrategies and the right substrategy:

$$x \mathbin{<\%>} y \qquad = (x \mathbin{\%>} y) \mathbin{<|>} y$$
$$(\langle a \rangle \mathbin{<\star>} x) \mathbin{\%>} y = a \mathbin{<\star>} (x \mathbin{<\%>} y)$$

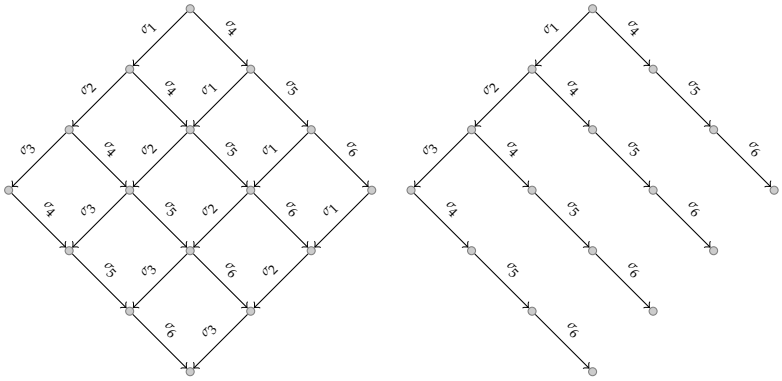
The right-hand side of the choice ignores steps from x. We recognise intermediate answers containing steps from x with the left-interleave of x with y. Because of the left-interleave, these steps are recognised before steps from y. This is safe because the order of refinement steps does not matter. Using the search mode for interleave, all sequences of refinement steps leading to the same intermediate program are replaced by a single sequence, drastically reducing the search space. The search mode is used in the *deepdiagnose* service (see Section 6.2.1). Note that we still need the normal behaviour of the interleave combinator for generating hints. The feedback services other than *deepdiagnose*, such as *allfirsts*, use the original semantics of the interleave combinator.

Figure 10.2 shows an example of all possible derivations for the normal and the alternative semantics of the interleave combinator, for the following strategy:

$$\sigma = (\sigma_1 \mathbin{<\star>} \sigma_2 \mathbin{<\star>} \sigma_3) \mathbin{<\%>} (\sigma_4 \mathbin{<\star>} \sigma_5 \mathbin{<\star>} \sigma_6)$$

When using the alternative semantics, all intermediate answers (depicted by the grey circles) are recognised by a single derivation. The search mode improves the efficiency of recognising (intermediate) programs substantially.

Our approach is similar to partial-order reduction in model checking (Alur et al., 2001). It can be applied in the functional programming domain because we use refinement rules. If we would also use rewrite rules, we would need to prove that the rewriting system is Church-Rosser to use the alternative semantics of interleave.



(a) Normal mode (b) Search mode

Figure 10.2

11 Assessing Haskell programs

Besides using programming strategies and programming transformations to generate semantically rich feedback, we also use these techniques for *assessing* functional programming exercises in Haskell.

Traditionally, a teacher or an assistant assesses a student's abilities and progress. However, providing timely feedback is not always possible with large class sizes. Repeatedly assessing student exercises is tedious, time consuming, and error prone. It is difficult to keep judgements consistent and fair. To assist teachers in assessing programming assignments, many assessment tools have been developed. We have developed a tool for assessing student programs in Haskell based on programming strategies. Using programming strategies we can guarantee that a student program is equivalent to a model solution, and we can report which solution strategy has been used to solve a programming problem.

Many programming exercise assessment tools are based on some form of testing (Ala-Mutka, 2005). Test-based assessment tools try to determine correctness by comparing the output of a student program to the expected results on test data. Using testing for assessment has a number of problems. First, an inherent problem of testing is coverage: how do you know you have tested enough? Testing does not ensure that the student program is correct. Second, assessing design features, such as the use of good programming techniques or the absence of imperfections, is hard if not impossible with testing. This is unfortunate, because teachers want students to adopt good programming techniques. Consider the following func-

tion that solves the problem of converting a list of binary numbers to its decimal representation:

$$
\begin{aligned}
&\mathit{fromBin} :: [\mathit{Int}] \to \mathit{Int}\\
&\mathit{fromBin} = \mathit{fromBin}'\ 2\\
&\mathit{fromBin}'\ n\ [\,] \qquad = 0\\
&\mathit{fromBin}'\ n\ (x : xs) = x \cdot n\hat{}(\mathit{length}\ (x : xs) - 1)\\
&\qquad\qquad\qquad\quad + \mathit{fromBin}'\ n\ xs
\end{aligned}
$$

This function returns correct results, hence test-based assessment tools will most likely accept this as a good solution. However, this implementation contains at least one imperfection: the length calculation is inefficient (an element is added to the list and then the length of the list is subtracted by one). We found this imperfection frequently in a set of student solutions. Third, testing cannot reveal which algorithm has been used. For instance, when asked to implement quicksort, it is difficult to discriminate between bubblesort and quicksort. Fourth, testing is a dynamic process and is therefore vulnerable to bugs, and even malicious features, that may be present in solutions.

We use programming strategies, derived from teacher annotated model solutions, and our normalisation procedure to assess functional programming exercises in Haskell. Our approach is rather different from testing: we can guarantee that the submitted student program is equivalent to a model program. We can recognise many different equivalent solutions from a model solution. For example, the following student solution:

$$
\begin{aligned}
&\mathit{fromBin} = \mathit{fromBaseN}\ 2\\
&\mathit{fromBaseN}\ b\ n = \mathit{fromBaseN}'\ b\ (\mathit{reverse}\ n)\\
&\quad \textbf{where}\\
&\qquad \mathit{fromBaseN}'\ \_\ [\,] \qquad = 0\\
&\qquad \mathit{fromBaseN}'\ b'\ (c : cs) = c + b' \cdot (\mathit{fromBaseN}'\ b'\ cs)
\end{aligned}
$$

is recognised from this model solution:

$$\mathit{fromBin} = \mathit{foldl}\ ((+) \circ (2\cdot))\ 0$$

Despite the fact that this solution appears very different from the model solution, it will be recognised as equivalent. The two solutions are essentially the same. The *foldl* function can be defined as a *foldr*:

$$\mathit{foldl}\ \mathit{op}\ b = \mathit{foldr}\ (\mathit{flip}\ \mathit{op})\ b \circ \mathit{reverse}$$

The student solution makes use of this equivalence. It uses, however, the explicit recursive definition of *foldr*. We specify this equivalence using the `ALT` annotation, see Section 7.3.

In this chapter we show how programming strategies and program transformations can be used to assess functional programming exercises. Using strategies for assessing student programs solves the four problems of using testing for assessment described above:

1. if a program is determined to be equivalent, it is guaranteed to be correct
2. we can recognise and report imperfections
3. we can determine which algorithm has been implemented
4. strategy-based assessment is carried out *statically*.

In contrast with our approach, test-based assessment tools can give a judgement of all programs including incorrect ones. Test-based assessment tools can prove a program to be incorrect by providing a counter-example. We could add testing, using QuickCheck, to the assessment tool to get around this disadvantage.

Strategy based assessment. The most important features we want to assess in a student program are:

- Correctness: does the program implement the requirements?
- Design: has the program been implemented following good programming practices?

We use programming strategies as a foundation for assessment of programming exercises. To use our assessment tool, a teacher only needs to specify one or more annotated model solutions. A programming strategy is automatically derived from these model solutions. Using a programming strategy we generate a set of solutions equivalent to the model solutions. A student solution is correct if it is an element of the generated set. In general, strategies do not generate all solutions that are equivalent to the model solution described. We normalise the generated set of model solutions and the student program using meaning preserving program transformations, which are described in Section 9.1. The assessment tool only assesses fully defined programs, which means that we can safely include the inlining and dead-code elimination program transformations in the normalisation procedure. After normalising we syntactically check if a program is an element of the set of normalised model solutions.

11.1 Using our assessment tool

We have applied our assessment tool to student solutions that were obtained from a lab assignment in a first-year functional programming course at Utrecht University

(2008). We were not involved in any aspect of the assignment, and received the solutions after they had been graded ('by hand') by the teaching assistants. In total we received 94 student solutions.

The students had to implement the *fromBin* function introduced earlier. This function should convert a list of bits to a decimal number. For example, applying *fromBin* to $[1,0,1,0,1,0]$ should return 42. This is a small typical beginners exercise in Haskell. The *fromBin* exercise can be solved in various ways, using different kinds of higher-order functions. There are a number of model solutions, which differ quite a bit from one another. All of them use recommended programming techniques.

The first of our model solutions uses a *foldl*.

```
{-# ALT foldl op b = foldr (flip op) b . reverse #-}
fromBin = foldl ((+) ∘ (2·)) 0
```

The second model solution uses tupling. Tupling is a well-known programming technique that groups intermediate results in a tuple, which is passed around in a recursive function. The tuple is passed in the form of multiple function arguments.

```
fromBin xs = fromBin' (length xs − 1) xs
  where
    fromBin' _ []     = 0
    fromBin' l (x:xs) = x · 2^l + fromBin' (l − 1) xs
```

The third solution reverses the input list, and then computes the inner product of this list and a list of powers of two.

```
fromBin = sum ∘ zipWith (·) (iterate (·2) 1) ∘ reverse
```

All of the above model solutions are elegant and efficient. The fourth (and last) model solution is simple, but inefficient:

```
fromBin []     = 0
fromBin (x:xs) = x · 2^length xs + fromBin xs
```

Because the length of the list is calculated in each recursive call, this definition takes time quadratic in the size of the input list to calculate its result. The other model solutions are all linear. It is up to the teacher to decide to either accept or reject solutions based on this model. This flexibility is one of the advantages of our approach.

11.1.1 Classification of student solutions

We have partitioned the set of student programs into four categories by hand:

Good. A good program is a proper solution with respect to the features we assess (correctness and design). It should ideally be equivalent to one of the model solutions.

Good with modifications. Some students have augmented their solution with sanity checks. For example, they check that the input is a list of zeroes and ones. Since the exercise assumes the input has the correct form, we have not incorporated such checks in the model solutions. The transformation machinery cannot yet remove such checks, we have removed them by hand.

Imperfect. We reject programs containing imperfections. The solution to *fromBin* given at the beginning of this chapter is an example of an imperfect solution. Another common imperfection we found is the use of a superfluous case:

$$\begin{aligned} &\mathit{fromBin}\ [\,] &&= 0 \\ &\mathit{fromBin}\ (x : [\,]) &&= x \\ &\mathit{fromBin}\ (x : xs) &&= (x \cdot 2\hat{}\mathit{length}\ xs) + \mathit{fromBin}\ xs \end{aligned}$$

In this example, the second case is unnecessary.

Incorrect. A few student programs were incorrect. They all contained the same error: no definition of *fromBin* on the empty list.

11.1.2 Results

From the 94 student programs, 64 programs fall into the good category and 8 fall into the good with modifications category. From these, our assessment tool recognises 64 programs (89%). Another, and perhaps better, way of looking at these figures is that 64 student solutions are accepted based on just four model solutions. All of the incorrect and imperfect programs were rejected in the test. Some of these incorrect programs were not noticed by the teaching assistants that corrected these programs.

Using our tool a teacher only needs to assess the remaining student solutions. Our tool cannot tell if these remaining student solutions are correct or incorrect. We could use testing for these cases.

It may happen that a correct student solution does not correspond to a model solution. If such a solution is elegant and efficient, a teacher could add it to the set of model solutions. In the case it does not meet the requirements for a model solution, it is up to the teacher to take a decision. For example, the following student solution uses the tupling technique:

$$
\begin{aligned}
&\mathit{fromBin}\ [\,] &&= 0\\
&\mathit{fromBin}\ [x] &&= x\\
&\mathit{fromBin}\ (x : y : \mathit{rest}) &&= \mathit{fromBin}\ ((2 \cdot x + y) : \mathit{rest})
\end{aligned}
$$

Instead of using a tuple or an extra argument, this solution 'misuses' the head of the list to store the result, which rules it out for being considered as a model solution. The teacher needs to decide whether or not this 'misuse' is an imperfection or not.

By checking all model solutions independently, we can tell which model solution, or strategy, a student has used to solve the exercise. Our test showed that 18 students used the *foldl* model solution, 2 used tupling, 2 the inner product solution, and 40 solutions were based on the last model solution with explicit recursion.

It is unlikely that a solution is accepted by more than one model solution. In our test all solutions were accepted by a single model solution. If model solutions are very similar, it might be possible to use the ALT annotation to recognise both from a single model solution.

11.2 Related work on assessment

The survey of automated programming assessment by Ala-Mutka (2005) shows that many assessment tools are based on dynamic testing. In contrast, our assessment tool statically checks for correctness. The survey provides many pointers to related work. We describe the three closest approaches.

The PASS system, developed by Thorburn and Rowe (1997), assesses C programs by evaluating whether a student program conforms to a predefined solution plan. A drawback of the system is that it needs testing for this evaluation. Moreover, a solution plan is much more strict compared to a strategy. For example, the system considers the definition of any helper-function incorrect. Our approach allows a higher degree of freedom by means of standard strategies and program transformations.

The approach of Truong et al. (2004) is also based on model solutions and abstract syntax tree inspections. However, their primary use is to assess software quality and not so much correctness. In addition to similarity checks, their system also calculates software metrics, which are used to give feedback to a student. A drawback of their approach is that it does not take the different syntactic forms of a model solution into account. Moreover, the similarity check considers only the outline of a solution and not its details.

Xu and Chee (2003) show how to diagnose Smalltalk programs using program transformations. Our work is quite similar to theirs. For a functional programming language the set of transformations is much smaller and simpler. We would like to implement their advanced method for locating errors in student programs.

Program verification tools are used to prove programs correct with respect to some specification (Mol et al., 2002). Automatic program verification tools provide as much support as possible in constructing this proof. However, users always need to give hints or proof steps to complete proofs for non-trivial programs, such as *fromBin*.

12 Epilogue and future work

In this thesis we have introduced a strategy language for specifying exercises, and an intelligent tutor that supports the stepwise development of simple functional programs. The tutoring system targets students at the starting university, or possibly end high-school, level. Using our programming tutor a student is allowed to develop a program in many different ways. Teachers can add programming exercises to the programming tutor by means of annotated model solutions. Teachers determine which solutions are accepted and/or suggested to students, and which solutions are not allowed. Our tutor automatically calculates hints and feedback at intermediate development steps from the model solutions for a problem. This reduces the work required for using the tutor, and allows a teacher to use her favourite exercises. The generated feedback is calculated from programming strategies, which are derived from the annotated model solutions. Using programming strategies, in combination with program transformations, the tutor recognises many different student programs from a limited set of model solutions.

The distinguishing characteristics of our functional programming tutor are:

- it supports the incremental development of programs: students can submit incomplete programs and receive feedback and/or hints.
- it calculates feedback automatically based on model solutions to exercises. A teacher does not have to specify feedback the feedback by hand.
- correctness is based on provable equivalence to a model solution, using

a normal form for functional programs. If we cannot determine whether or not a program is equivalent to a model solution, we use testing as an approximation.

- it recognises arbitrary many student steps on the way to a solution.

We have conducted an experiment in which around a hundred students worked with our functional programming tutor. Furthermore, we asked functional programming experts about the design choices we made. The main conclusions of these two investigations are:

- Students appreciate worked-out solutions, and are moderately positive about the tutor.

- We need to judge student programs even when a student deviates from the model solutions.

We have extended our programming tutor with testing capabilities to address the latter judgement problem.

We use programming strategies not only for generating feedback, but also for assessing programming exercises. We differ from test based assessment tools in that we can guarantee a student solution to be equivalent to a model solution. In a test we performed on almost 100 student programs we managed to recognise and characterise 89% of the correct solutions, and we found several programs that had been incorrectly graded as correct by student assistants.

We have introduced a strategy language with which we can specify strategies for programming exercises, and exercises in many other domains. A strategy is defined as a context-free grammar. The formulation of a strategy as a context-free grammar allows us to automatically calculate several kinds of feedback. Languages for modelling procedures or strategies for exercises have been developed before. Our language has a similar expressive power and structure. Our main contribution is that we make strategies explicit, and that we can automatically calculate advanced feedback. This is achieved by separating the strategy language into a context-free language, the strategy combinators, and a non-context-free language, the embedding as a domain-specific language.

Furthermore, we have presented a formal and precise definition of the main concepts that we use to construct semantically rich feedback for ITSs, such as the strategy language and the feedback services. We have defined several relations on strategies that give the semantics of the strategy language, such as the big-step relation. Feedback services are an interface to our feedback functionality that can be used by learning environments. These services are expressed in terms of the big-step relation. The formalisation gives us more confidence in the correctness of

our approach. Furthermore, the formalisation allows us to state properties that the concepts used should have. We use these properties to validate our implementation.

We have also presented the implementation of a recogniser for strategies. Although it is tempting to reuse existing parsing tools and libraries, a closer look at the problem reveals subtle differences that make existing tools unsuitable for recognising incomplete beginners' programs. Some design choices were discussed, in particular for how to deal with recursion, and how to mark positions in a strategy. We have shown how our implementation of a recogniser can be used to automatically calculate several kinds of feedback.

12.1 Future work

We conclude with a list of possible directions for further research.

Functionality

Refactoring exercises. For now our programming tutor only offers exercises that can be solved by refining an intermediate program to a solution. In addition to this kind of exercises, we would like to offer refactoring exercises. When solving a refactoring exercise a student rewrites (instead of refines) a program into a semantically equivalent program, which is possibly more efficient or elegant. The *HaRe* (Li et al., 2005) and *hlint* [1] projects are good sources for an initial set of rewrite rules for the domain of functional programming.

Other programming languages/paradigms. The concepts that we use to generate feedback and hints for programming exercises are not specific for Haskell. We can use the approach described in this thesis to develop similar programming tutors for other functional programming languages, such as Lisp or OCaml. Our approach is not bound to functional programming: we could use the same approach to develop tutoring systems for other programming languages or paradigms. We believe that we have not made assumptions that exclude imperative programming languages and think that our programming tutor is *language generic*, but we would have to further investigate this. We want to investigate the possibilities for automatically generating large parts of a programming tutor, based on a (probably annotated) grammatical description (Klint et al., 2005), and using generic programming techniques (Backhouse et al., 1999).

Support for larger programs. Our programming tutor offers introductory programming exercises, which are rather small. We have not investigated if the

[1] `http://community.haskell.org/~ndm/hlint/`

domain reasoner for functional programming can handle large programs. We might not be able to recognise all steps from beginning to end for large programs. However, we are probably able to recognise the first steps. We think 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. We would like to investigate whether or not our approach scales, and check if it is able to deal with larger programs.

Bug location. When a student deviates from a strategy, we use testing to check if the student submission has the same behaviour as a model solution. In case we find a counterexample, we report to the student that the program is incorrect and display the counterexample. It would be an improvement if we could give the student the location of the error. If we are able to determine the location of the bug, we could display, for example, a red line underneath the part of the program that causes the erroneous behaviour. Brain-storming sessions with Koen Claessen, Bastiaan Heeren, Johan Jeuring, and Ulf Norell led to the following idea. If we find a counterexample in a program, we gradually undefine the program (i.e., make a program less defined, by replacing a defined part with a hole) and test if it still behaves incorrectly. When we can no longer falsify the program, we know what part of the program is incorrect. The following derivation exemplifies this process for the *reverse* function:

$$
\begin{aligned}
& \mathit{reverse}\ [\,] \quad\ = \bullet \\
& \mathit{reverse}\ (x : xs) = x : \bullet \\
\Rightarrow\ & \{\mathit{test\ fails}\ ;\ \mathit{remove\ first\ function\ binding}\} \\
& \mathit{reverse}\ (x : xs) = x : \bullet \\
\Rightarrow\ & \{\mathit{test\ still\ fails}\ ;\ \mathit{remove\ right-hand\ side}\} \\
& \mathit{reverse}\ (x : xs) = \bullet \\
\Rightarrow\ & \{\mathit{test\ succeeds}\ ;\ \mathit{undefine}\ (:)\} \quad \text{-- go as deep as possible} \\
& \mathit{reverse}\ (x : xs) = x\ \text{`}\bullet\text{`}\ \bullet \\
\Rightarrow\ & \{\mathit{test\ succeeds}\ ;\ \mathit{undefine}\ x\} \\
& \mathit{reverse}\ (x : xs) = \bullet\ :\ \bullet \\
\Rightarrow\ & \{\mathit{test\ succeeds}\ ;\ \mathit{max\ depth\ reached}\}
\end{aligned}
$$

From the above example we can conclude that the error is introduced by the x in the right-hand side of the second function binding. Using this information we could give the student a detailed feedback message, in which an indication of the location of the error is underlined in red.

The described process is similar to techniques that are used to construct type

error messages (Lerner et al., 2007). The approach also bears resemblance to the shrinking process in QuickCheck (Claessen and Hughes, 2000). We would like to investigate this idea, and experiment with it.

Program design exercises. Developing a function is an important part of functional programming. But so are testing a function, describing its properties, abstracting from recurring patterns, etc. (Felleisen et al., 2002). We want to investigate how much of the program design process can be usefully integrated in an intelligent tutoring system for functional programming.

Technology

Inlining/dead-code elimination. When recognising intermediate program solutions we do not use inlining and dead-code removal. If we inline an incomplete program submitted by a student, we might undo the step that a student has taken. If the student introduces a helper function that is not used anywhere in the program, then inlining and dead-code removal would undo this step. As a consequence, the tutor is unable to detect if the student has taken a step. For example, if the student submits the next program:

$$f = \bullet \ \bullet$$

followed by

$$f = g \ \bullet$$
$$\textbf{where}$$
$$g = \bullet$$

Applying the inlining/dead-code removal program transformation would result in the previous program submitted by the student.

However, inlining is needed to recognise some intermediate programs. For instance, if a student uses a helper function whereas the corresponding model solution does not, we need inlining/dead-code removal to recognise the student solution. We would like to investigate when it is possible to perform the inlining/dead-code removal program transformation.

Compiler. We use the Helium compiler because it generates excellent error messages. Unfortunately, Helium does not fully support the Haskell98 standard, let alone the newer Haskell2010 standard. For example, there is no support for type classes. GHC does support both Haskell standards, and has many more features. We have implemented our programming tutor in a modular way, and it is should not take too much effort to replace the current compiler. However, we have extended and adapted some parts of Helium, such as the parser and the abstract syntax. We want to investigate if it is possible to use GHC instead of Helium for compiling student programs.

Usability

Experiments with teachers. We have not yet performed experiments with teachers, excluding ourselves, using our system. We want to perform experiments in which teachers need to add programming exercises, and fine-tune the generation of feedback. We want to test the usability of our tutor.

Learning effect. We also want to study the learning effect of our tutor together with researchers from the domain of learning sciences.

SAMENVATTING

Computers zijn niet meer weg te denken uit onze samenleving. Dagelijks komen veel mensen op verschillende manieren in aanraking met computers. Is het niet met een normale desktop computer, dan wel met een mobiele telefoon, die een kleine computer onder de motorkap heeft. Ook televisies, witgoed-apparaten en auto's, om slechts enkele te noemen, zijn tegenwoordig steeds meer voorzien van een computer. Bovendien hebben we ook vaak indirect met computers van doen. Bijvoorbeeld, iemand die wel eens contact heeft gehad met een helpdesk, weet maar al te goed dat veel zaken door computers worden afgehandeld[2].

Een computer kan berekeningen voor ons uitvoeren; dit gebeurt via *computerprogramma's*[3], of programma's in het kort. Deze programma's zijn geïnstalleerd op een computer en sturen deze aan. Bijvoorbeeld, een navigatie-programma stuurt de navigatie-computer in een auto aan, door deze een route te laten berekenen naar een opgegeven bestemming en door aanwijzigingen weer te geven op een scherm. Vanzelfsprekend is het belangrijk dat de programma's en computers functioneren zoals de gebruiker dat voor ogen heeft.

Computerprogramma's worden geschreven in een *programmeertaal*. In een programmeertaal kun je uitdrukken wat een computer moet doen. Er bestaan vele programmeertalen zoals: Haskell, Erlang, Clean, Java, C en C++. Deze programmeertalen zijn onder te verdelen in twee groepen (paradigma's), namelijk het imperatieve en het functionele paradigma. Een programma geschreven in een imperatieve programmeertaal bestaat uit een sequentie van instructies die een computer achtereenvolgens uitvoert. Functionele talen hebben een andere aan-

[2] Wat niet per definitie inhoudt dat de zaken dan beter zijn geregeld.
[3] Een andere term voor een computerprogramma is een *applicatie*.

pak; in een functionele programmeertaal definieert een programmeur een aantal (wiskundige) functies die beschrijven wat een computer moet doen.

Het schrijven van een computerprogramma is geen eenvoudige aangelegenheid en vergt kennis van een programmeertaal. Het leren van een programmeertaal is eveneens geen sinecure. Evenals bij andere vakken is het maken van opgaven een belangrijk onderdeel van het leren. Door feedback (terugkoppeling) te geven op uitgewerkte opgaven kan een student zich verbeteren. Onderzoek heeft aangetoond dat het geven van feedback het meest effectief is, wanneer dit tijdens het maken van een opgave gebeurt. Normaliter wordt deze directe feedback door een leraar gegeven. Het is natuurlijk erg moeilijk voor een leraar om een grote groep studenten directe feedback te geven. Om een leraar daarbij te ondersteunen zijn er vele elektronische leeromgevingen ontwikkeld, waarin studenten opgaven kunnen maken en daarbij directe feedback aangeboden krijgen.

Er zijn leeromgevingen ontwikkeld voor een aantal programmeertalen, waaronder Java, Lisp, Prolog en Haskell. Een leeromgeving biedt meer voordelen dan enkel de mogelijkheid voor het geven van directe feedback. Zo is het bijvoorbeeld altijd beschikbaar en kunnen grote groepen studenten er tegelijkertijd gebruik van maken. Ondanks de voordelen van leeromgevingen voor programmeren, worden ze niet veel gebruikt. Naast de voordelen kleven er ook een aantal nadelen aan het gebruik van een leeromgeving voor programmeren. Een veel voorkomend nadeel is dat het moeilijk is voor een leraar om de aangeboden opgaven aan te passen. Het toevoegen van opgaven is vaak veel werk, waarbij de te geven feedback soms met de hand moet worden gespecificeerd.

In dit proefschrift laten we onder andere zien hoe we deze nadelen kunnen aanpakken. Wij hebben een programmeerleeromgeving ontwikkeld, genaamd ASK-ELLE, waarin feedback automatisch wordt gegenereerd, en het gemakkelijk is om opgaven toe te voegen of aan te passen. ASK-ELLE is een leeromgeving voor de functionele programmeertaal Haskell. Haskell heeft een aantal karakteristieke eigenschappen: het heeft een 'lazy' evaluatie mechanisme, het is sterk getypeerd en het staat niet zonder meer neveneffecten toe. Mede door deze eigenschappen is Haskell een geschikte programmeertaal voor het gebruik in het onderwijs.

In ASK-ELLE kunnen studenten o.a.: stapsgewijs programma's ontwikkelen, nagaan of een gemaakte stap goed is of niet, om een hint vragen als ze vast zitten en volledig stapsgewijs uitgewerkte programma's bekijken. Deze feedback wordt automatisch gegenereerd op basis van *modeloplossingen* voor een programmeeropgave. Een modeloplossing is een uitwerking voor een opgave die een expert zou schrijven en waarin gebruik wordt gemaakt van goede programmeertechnieken. Een leraar kan een opgave aan ASK-ELLE toevoegen door één of meerdere modeloplossingen te definiëren voor een programmeeropgave. De genereerde feedback kan worden aangepast door de leraar. Daartoe kan de leraar de modeloplossingen voorzien van aanwijzingen hoe de feedback moet worden gegenereerd. Tevens kan

een leraar de feedback teksten bewerken en is er ondersteuning voor feedback in meerdere talen.

Het proefschrift is in twee delen gesplitst. Het eerste deel behandelt 'domain reasoners'. Een 'domain reasoner' is verantwoordelijk voor de generatie van feedback voor opgaven uit een bepaald domein. Voorbeelden van zulke domeinen zijn: propositie logica, lineaire algebra, rekenen met breuken en functioneel programmeren. Onze 'domain reasoners' bieden de generatie van feedback aan via *webservices*. Webservices maken het mogelijk om onze 'domain reasoners' op afstand via het Internet te gebruiken. Bijvoorbeeld, het minimum aantal stappen dat een student nog moet nemen om een opgave op te lossen, wordt als webservice aangeboden. Naast onze eigen leeromgevingen maken ook externe leeromgevingen gebruik van onze 'domain reasoners'.

Een 'domain reasoner' bevat een aantal opgaven voor het gekozen domain. De opgaven worden beschreven binnen de context van een 'domain reasoner'. De beschrijving van een opgave bevat onder andere: een omschrijving van de syntax van het domein, een set van regels waarmee expressies in het domain kunnen worden herschreven, de *strategie* waarmee de opgave kan worden opgelost en semantische functies (zoals de gelijkheidsfunctie). Deze onderdelen heeft de 'domain reasoner' nodig om feedback te genereren.

Een prominent onderdeel van een opgave is de strategie; deze heeft verreweg het meeste invloed op de generatie van feedback. Een strategie beschrijft welke stappen (toepassingen van regels) een student kan nemen om een opgave op te lossen. Wij hebben een *strategietaal* ontwikkeld waar strategieën in kunnen worden uitgedrukt. Deze strategietaal biedt een aantal combinatoren aan waarmee strategieën kunnen worden samengesteld. Bijvoorbeeld, de strategietaal heeft een combinator waarmee twee (sub)strategieën in sequentie kunnen worden gezet. Dat wil zeggen dat eerst de ene strategie moet worden uitgevoerd voordat aan de andere mag worden begonnen. Een strategie wordt geïnterpreteerd als een contextvrije grammatica. We hebben een aantal strategiefuncties ontwikkeld, die veel lijken op de *firsts* en *empty* functies voor contextvrije grammatica's. De *firsts* strategiefunctie bepaalt de set van toegestane stappen volgens de strategie. De *empty* functie gaat na of we volgens de strategie klaar zijn. Deze twee functies worden veelvuldig gebruikt bij het genereren van feedback. In dit proefschrift presenteren we een formalisatie van de strategietaal. Dit geeft ons de mogelijkheid om eigenschappen af te leiden, die de strategietaal zou moeten hebben. Deze eigenschappen kunnen we gebruiken om onze implementatie te testen.

Het tweede deel van dit proefschrift gaat in op de details van ASK-ELLE. We laten een voorbeeld interactieve sessie zien hoe een student een opgave oplost met behulp van ASK-ELLE. Een student begint met een totaal ongedefinieerd programma (een gat) en maakt stapsgewijs het programma meer compleet. Hiervoor gebruikt de student *verfijningsregels* voor het functionele programmeerdomein. Het

toepassen van een verfijningsregel op een incompleet programma vervangt een gat in het programma door een programma-constructie (dat mogelijk gaten bevat), en maakt daarmee het programma meer gedefinieerd. Dit wijkt af van de andere (wiskundige) domeinen, waar herschrijfregels worden gebruikt in plaats van verfijningsregels. Een student is klaar met een opgave als het programma volledig gedefinieerd is en geen gaten meer bevat.

Welke verfijningsregels een student kan toepassen, wordt bepaald door een (programmeer)strategie. Deze strategie wordt afgeleid uit een set van modeloplossingen voor een programmeeropgave. Elke constructie in een modeloplossing wordt vertaald naar een bijbehorende verfijningsregel. Bijvoorbeeld, als een modeloplossing een **if** − **then** − **else**-constructie bevat, wordt deze tijdens het afleiden van de strategie naar een verfijningsregel vertaald die een gat vervangt door deze constructie. Indien er in de modeloplossing gebruik wordt gemaakt van een bibliotheekfunctie (een voorgedefinieerde standaardfunctie), dan wordt de strategie zo uitgebreid dat we het gebruik van een dergelijke functie ook herkennen. Hiermee kan de programmeertutor meerdere equivalente vormen van hetzelfde programma herkennen en is daarmee flexibeler geworden. De student heeft dan de vrijheid om de modeloplossing te volgen of de definitie van een bibliotheekfunctie te gebruiken. Om de tutor nog meer equivalente vormen van een modeloplossing te laten herkennen, maken we gebruik van *normalisatie*. Normalisatie zorgt ervoor dat we niet-significante verschillen negeren. Bijvoorbeeld, het feit dat een student een andere naam gebruikt voor een variabele in vergelijking met de modeloplossing willen we graag negeren. Normalisatie maakt gebruik van *programmatransformaties* gebaseerd op de *λ-calculus*. Indien een student afwijkt van de strategie kunnen we geen oordeel geven of het programma van de student correct is of niet. In de wiskundige domeinen kunnen we terugvallen op een equivalentie functie. Deze is voor het functionele programmeerdomein echter niet beschikbaar. In een dergelijk geval maken we gebruik van *testen* om na te gaan of een programma van een student zich hetzelfde gedraagt als een modeloplossing. De programmeertutor kan zelfs incomplete programma's testen.

We hebben een aantal experimenten met de programmeertutor uitgevoerd. We hebben ongeveer 200 eerstejaars studenten informatica van de Universiteit Utrecht met de tutor laten werken. De studenten waren gematigd positief over de tutor. Uit een enquête die de studenten hebben ingevuld, kwamen twee verbeterpunten naar voren. De tutor dient een oordeel te geven indien er wordt afgeweken van de strategie en de tutor moet programma's herkennen die tot stand zijn gekomen met het toepassen van meerdere verfijningsregels in één keer. Het eerste punt hebben we aangepakt door het testen van incomplete programma's toe te voegen aan de tutor (dit was op het moment van de experiment nog niet geïmplementeerd). Het tweede punt hebben we opgelost door de manier van herkennen van (incomplete) programma's aan te passen. Doordat de strategie heel veel verschillende vormen

van een modeloplossing herkent, is het geen optie om deze allemaal te genereren en vervolgens na te gaan of het studentprogramma een element is van deze set. We geven de strategietaal een andere semantiek wanneer we een studentprogramma proberen te herkennen. We maken gebruik van het feit dat tijdens het herkennen van een studentprogramma de volgorde van verfijningsregels niet relevant is. Door de volgorde te fixeren, wordt de zoekruimte enorm gereduceerd.

We gebruiken de technieken die aan onze programmeertutor ten grondslag liggen ook voor een ander doel. We hebben een applicatie ontwikkeld waarmee we geautomatiseerd functionele programmeeropgaven kunnen nakijken. Met deze applicatie hebben we eveneens een experiment uitgevoerd. De resultaten zijn veelbelovend: op basis van een vijftal modeloplossing konden we bijna 90% van alle studentuitwerkingen van een oordeel voorzien.

Curriculum vitae

Alex Gerdes

10 juli 1978	Geboren te Emmen
1990 – 1995	HAVO aan het Esdal College te Emmen
1995 – 1996	HTS - Elektrotechniek aan de Hogeschool Drenthe (propedeuse)
1996 – 2000	HTS - Technische Informatica aan de Hogeschool Drenthe
1999 – 2001	Software design engineer bij Ericsson te Emmen
2001 – 2007	Studie Technische Informatica aan de Open Universiteit
	Afstudeeronderzoek op het gebied van generiek programmeren, uitgevoerd aan de Universiteit Utrecht.
2001 – 2007	Signal processing/software engineer bij ASTRON te Dwingeloo
2007 – 2012	Promovendus aan de faculteit Informatica van de Open Universiteit
Q2 – 2012	Software developer bij de Universiteit Utrecht
2012 – heden	Software developer bij QuviQ te Göteborg in Zweden

BIBLIOGRAPHY

Kirsti Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005.

R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-Order Reduction in Symbolic State-Space Exploration. *Formal Methods in System Design*, 18:97–116, 2001.

John R. Anderson. *Rules of the Mind*. Lawrence Erlbaum Associates, 1993.

John R. Anderson, Frederick G. Conrad, and Albert T. Corbett. Skill Acquisition and the LISP tutor. *Cognitive Science*, 13:467–505, 1986.

John R. Anderson, Albert T. Corbett, Kenneth R. Koedinger, and Ray Pelletier. Cognitive tutors: lessons learned. *The Journal of the learning sciences*, 4(2):167–207, 1995.

David Aspinall, Ewen Denney, and Christoph Lüth. A Tactic Language for Hiproofs. In *MKM 2008: Proceedings of the 7th international conference on Intelligent Computer Mathematics*, volume 5144 of *LNCS*, pages 339–354. Springer Verlag, 2008.

Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1997.

Ralph-Johan Back. A Calculus of Refinements for Program Derivations. Reports on Computer Science and Mathematics 54, Åbo Akademi, 1987.

Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic Programming — An Introduction. In *AFP 1999: 3rd International Summer School on Advanced Functional Programming*, volume 1608 of *LNCS*, pages 28–115. Springer Verlag, 1999.

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

Michael J. Beeson. Design Principles of Mathpert: Software to support education in algebra and calculus. In N. Kajler, editor, *Computer-Human Interaction in Symbolic Computation*, pages 89–115. Springer Verlag, 1998.

Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by Evaluation. In B. Möller and J. Tucker, editors, *Prospects for Hardware Foundations*, volume 1546 of *LNCS*, pages 624–624. Springer Verlag, 1998.

Jan A. Bergstra and Jan Willem Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

John Biggs and Catherine Tang. *Teaching for Quality Learning at University*. Open University Press, 2007.

Richard Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume F36 of *NATO ASI Series*, pages 5–42. Springer Verlag, 1987.

Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998.

Christian Bokhove and Paul Drijvers. Digital Tools for Algebra Education: Criteria and Evaluation. *International Journal of Computers for Mathematical Learning*, 15(1): 45–62, 2010.

Peter Boon and Paul Drijvers. Algebra en applets, leren en onderwijzen (algebra and applets, learning and teaching, in Dutch). `http://www.fi.uu.nl/publicaties/literatuur/6571.pdf`, 2005.

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

Eric Bouwers. Improving automated feedback – a generic rule-feedback generator. Master's thesis, Utrecht University, department of Information and Computing Sciences, 2007.

John Seely Brown and Richard R. Burton. Diagnostic Models for Procedural Bugs in Basic Mathematical Skills. *Cognitive Science*, 2:155–192, 1978.

Alan Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press, 1983.

Alan Bundy. The use of explicit plans to guide inductive proofs. In *Conference on Automated Deduction*, pages 111–120, 1988.

Alan Bundy. A critique of proof planning. In *Computational Logic (Kowalski Festschrift)*, volume 2408 of *LNAI*, 2002.

Hamid Chaachoua, Jean-François Nicaud, Alain Bronner, and Denis Bouhineau. Aplusix, a learning environment for algebra, actual use and benefits. In *ICME 2004: 10th International Congress on Mathematical Education*, 2004.

Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP 2000: Proceedings of the 5th ACM SIGPLAN international conference on Functional Programming*, pages 268–279. ACM Press, 2000.

Arjeh Cohen, Hans Cuypers, Ernesto Reinaldo Barreiro, and Hans Sterk. Interactive mathematical documents on the web. In *Algebra, Geometry and Software Systems*, pages 289–306. Springer Verlag, 2003.

Albert T. Corbett, John R. Anderson, and Eric J. Patterson. Problem compilation and tutoring flexibility in the Lisp tutor. In *ITS 1988: Proceedings of the 1st international conference on Intelligent Tutoring SystemsProceedings of the conference on Intelligent Tutoring Systems*, pages 423–429, 1988.

Alcino Cunha and Joost Visser. Strongly typed rewriting for coupled software transformation. *Electronic Notes in Theoretical Computer Science*, 174(1):17–34, 2007.

Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.

Hans van Ditmarsch. Logic software and logic education. `http://www.cs.otago.ac.nz/staffpriv/hans/logiccourseware.html`. These pages contain a comprehensive, alphabetically ordered list of educational logic software, 2009.

Ido Erev, Adi Luria, and Anan Erev. On the effect of immediate feedback, 2006. `http://goo.gl/eodze`.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to design programs: an introduction to programming and computing*. The MIT Press, 2002.

Andrzej Filinski and Henning Korsholm Rohde. A denotational account of untyped normalization by evaluation. In I. Walukiewicz, editor, *Foundations of Software Science and Computation Structures*, volume 2987 of *LNCS*, pages 167–181. Springer Verlag, 2004.

Sally Fincher and Marian Petre, editors. *Computer Science Education Research*, 2004. Routledge Falmer.

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: a programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.

Jeroen Fokker. Functional parsers. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995.

Wan Fokkink. *Introduction to Process Algebra*. Springer Verlag, 2000.

Freudenthal Institute. Digital Math Environment. `http://www.fi.uu.nl/dwo`, 2004.

Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and Sylvia Stuurman. Feedback services for exercise assistants. In D. Remenyi, editor, *The Proceedings of the 7th European Conference on e-Learning*, pages 402–410. Academic Publishing Limited, 2008.

Alex Gerdes, Bastiaan Heeren, and Johan Jeuring. Constructing Strategies for Programming. In J. Cordeiro, B. Shishkov, A. Verbraeck, and M. Helfert, editors, *Proceedings of the First International Conference on Computer Supported Education*, pages 65–72. INSTICC Press, 2009.

Alex Gerdes, Bastiaan Heeren, and Johan Jeuring. Properties of Exercise Strategies. *Electronic Notes in Theoretical Computer Science*, 44:21–34, 2010a.

Alex Gerdes, Johan Jeuring, and Bastiaan Heeren. Using Strategies for Assessment of Programming Exercises. In *SIGCSE 2010: Proceedings of the 41st ACM SIGPLAN technical symposium on Computer Science Education*, pages 441–445, 2010b.

Alex Gerdes, Johan Jeuring, and Bastiaan Heeren. Teachers and students in charge. In *EC-TEL 2012: the 7th European Conference on Technology Enhanced Learning*. Springer Verlag, 2012a. To appear. An extended version is available as Technical report Utrecht University UU-CS-2012-007.

Alex Gerdes, Johan Jeuring, and Bastiaan Heeren. An interactive functional programming tutor. In *ITICSE 2012: Proceedings of the 17th Annual Conference on Innovation and Technology in Computer Science Education*. ACM Press, 2012b.

Giorgi Goguadze, Alberto González Palomo, and Erica Melis. Interactivity of exercises in ActiveMath. In *ICCE 2005: International Conference on Computers in Education*. IOS Press, 2005.

Mark Guzdial. Programming environments for novices. In Sally Fincher and Marian Petre, editors, *Computer Science Education Research*. Routledge Falmer, 2004.

John Hattie and Helen Timperley. The power of feedback. *Review of Educational Research*, 77(1):81–112, 2007.

Bastiaan Heeren and Johan Jeuring. Recognizing Strategies. In Aart Middeldorp, editor, *WRS 2008: Reduction Strategies in Rewriting and Programming, 8th International Workshop*, 2008.

Bastiaan Heeren and Johan Jeuring. Canonical Forms in Interactive Exercise Assistants. In *MKM 2009: Proceedings of the 8th international conference on Intelligent Computer Mathematics*, volume 5625 of *LNCS*, pages 325–340. Springer Verlag, 2009.

Bastiaan Heeren and Johan Jeuring. Adapting mathematical domain reasoners. In *MKM 2010: Proceedings of the 9th international conference on Intelligent Computer Mathematics*, volume 6167 of *LNCS*, pages 315–330. Springer Verlag, 2010.

Bastiaan Heeren and Johan Jeuring. Interleaving strategies. In *MKM 2011: Proceedings of the 10th international conference on Intelligent Computer Mathematics*, volume 6824 of *LNCS*, pages 196–211. Springer Verlag, 2011.

Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning Haskell. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 62 – 71. ACM Press, 2003.

Bastiaan Heeren, Johan Jeuring, Arthur Leeuwen, and Alex Gerdes. Specifying Strategies for Exercises. In *Proceedings of the 9th AISC international conference, the 15th Calculemas symposium, and the 7th international MKM conference on Intelligent Computer Mathematics*, pages 430–445. Springer Verlag, 2008.

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

Martin Hennecke. *Online Diagnose in intelligenten mathematischen Lehr-Lern-Systemen (in German)*. PhD thesis, Hildesheim University, 1999.

Ralf Hinze and Johan Jeuring. Generic Haskell: Applications. In R. Backhouse and J. Gibbons, editors, *Generic Programming*, volume 2793 of *LNCS*, pages 57–96. Springer Verlag, 2003.

Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing approaches to generic programming in Haskell. In R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors, *Datatype-Generic Programming*, volume 4719 of *LNCS*, pages 72–149. Springer Verlag, 2007.

Tony Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

Tony Hoare. *Communicating sequential processes*. Prentice Hall, 1985.

Jay Holland, Tanja Mitrovic, and Brent Martin. J-Latte: a constraint-based tutor for Java. In *ICCE 2009: Proceedings of the 17th International on Conference Computers in Education*, pages 142–146, 2009.

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

Helmut Horacek and Magdalena Wolska. Handling errors in mathematical formulas. In M. Ikeda, K. Ashley, and T.-W. Chan, editors, *ITS 2006: Proceedings of the 8th international conference on Intelligent Tutoring Systems*, volume 4053 of *LNCS*, pages 339–348. Springer Verlag, 2006.

Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28A(4), 1996.

Paul Hudak. *The Haskell school of expression: learning functional programming through multimedia*. Cambridge University Press, 2000.

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

John Hughes. The Design of a Pretty-printing Library. In *AFP 1995: 1st International Spring School on Advanced Functional Programming*, volume 925 of *LNCS*, pages 53–96. Springer Verlag, 1995.

John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37: 67 – 111, 2000.

Graham Hutton. Higher-order Functions for Parsing. *Journal of Functional Programming*, 2(3):323–343, 1992.

Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

Marina Issakova. *Solving of linear equations, linear inequalities and systems of linear equations in interactive learning environment*. PhD thesis, University of Tartu, 2007.

Johan Jeuring, Alex Gerdes, and Bastiaan Heeren. A programming tutor for Haskell. In V. Zsók, Z. Horváth, and R. Plasmeijer, editors, *CEFP 2011: Central European Functional Programming School*, volume 7241 of *LNCS*, pages 1–45. Springer Verlag, 2012.

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

Henry A. Kautz and James F. Allen. Generalized plan recognition. In *National Conference on Artificial Intelligence*, pages 32–37, 1986.

Caitlin Kelleher and Randy Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys*, 37(2):83–137, 2005.

Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology*, 14(3): 331–380, 2005.

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

Amruth N. Kumar. The effect of using problem-solving software tutors on the self-confidence of female students. In *SIGCSE 2008: Proceedings of the 39th ACM SIGPLAN technical symposium on Computer Science Education*, pages 523–527. ACM Press, 2008.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, 2003.

Ralf Lämmel, Eelco Visser, and Joost Visser. The Essence of Strategic Programming. Available at `http://homepages.cwi.nl/~ralf/eosp/`, 2002.

Daan Leijen and Erik Meijer. Parsec: A practical parser library. *Electronic Notes in Theoretical Computer Science*, 41(1):1–20, 2001.

Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. Searching for type-error messages. In *PLDI 2007: Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 425–434. ACM Press, 2007.

Huiqing Li, Simon Thompson, and Claus Reinke. The Haskell Refactorer, HaRe, and its API. *Electronic Notes in Theoretical Computer Science*, 141(4):29–34, 2005.

Josje Lodder, Johan Jeuring, and Harrie 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.

Josje Lodder, Harrie Passier, and Sylvia Stuurman. Using IDEAS in Teaching Logic, Lessons Learned. In *CSSE 2008: International Conference on Computer Science and Software Engineering*, pages 553–556, 2008.

Natalia López, Manuel Núñez, Ismael Rodríguez, and Fernando Rubio. 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, 2002.

Susan Lowes. Online teaching and classroom change: The impact of virtual high school on its teachers and their schools. Technical report, Columbia University, Institute for Learning Technologies, 2007.

Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. In *Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 125–180. ACM Press, 2001.

Lambert Meertens. Algorithmics — towards programming as a mathematical activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 289–334. North–Holland, 1986.

Erica Melis, Eric Andrès, George Goguadze, Paul Libbrecht, Martin Pollet, and Carsten Ullrich. Activemath: a generic and adaptive web-based learning environment. *International Journal of Artificial Intelligence in Education*, 12:385–407, 2001.

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

Maarten de Mol, Marko van Eekelen, and Rinus Plasmeijer. Theorem proving for functional programmers - Sparkle: a functional theorem prover. In *IFL 2001: Proceedings of the 13th International Workshop on Implementation of Functional Languages*, volume 2312 of *LNCS*, pages 55–72. Springer Verlag, 2002.

Carroll Morgan. *Programming from specifications*. Prentice Hall, 1990.

Edna H. Mory. Feedback research revisited. In D.H. Jonassen, editor, *Handbook of research for educational communications and technology*, 2003.

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

Dereck C. Oppen. Pretty printing. *ACM Transactions Programming Languages and Systems*, 2(4):465–483, 1980.

Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 2008.

Claus Pahl. Managing evolution and change in web-based teaching and learning environments. *Computers & Education*, 40(2):99–114, 2003.

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

Ross Paterson. Arrows and computation. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, pages 201–222. Palgrave, 2003.

Larry C. Paulson. *ML for the Working Programmer, 2nd Edition*. Cambridge University Press, 1996.

Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. A survey of literature on the teaching of introductory programming. In *Working group reports on ITiCSE on Innovation and technology in computer science education*, pages 204–223. ACM Press, 2007.

Simon Peyton Jones, editor. *Haskell 98, Language and Libraries. The Revised Report*. Cambridge University Press, 2003.

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

Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: executable specifications of interactive work flow systems for the web. In *ICFP 2007: Proceedings of the 12th ACM SIGPLAN international conference on Functional Programming*, pages 141–152. ACM Press, 2007.

Viera K. Proulx. Programming patterns and design patterns in the introductory computer science course. In *SIGCSE 2000: Proceedings of the 31st ACM SIGPLAN technical symposium on Computer Science Education*, pages 80–84. ACM Press, 2000.

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 37–48. ACM Press, 2008.

John A. Self. Formal approaches to student modelling. Technical report, Lancaster University, 1991.

The OpenMath Society. The OpenMath Standard. `http://www.openmath.org/standard/index.html`, 2006.

Elliot Soloway. 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, 1985.

Elliot M. Soloway, Beverly Woolf, Eric Rubin, and Paul Barth. Meno-II: an intelligent tutoring system for novice programmers. In *IJCAI 1981: Proceedings of the 7th international joint conference on Artificial intelligence*, volume 2, pages 975–977. Morgan Kaufmann Publishers Inc., 1981.

J. Sweller, J.J.G. van Merriënboer, and F. Paas. Cognitive architecture and instructional design. *Educational Psychology Review*, 10:251–295, 1998.

Doaitse Swierstra and Luc Duponcheel. Deterministic, Error-Correcting Combinator Parsers. In *AFP 1996: 2nd International Summer School on Advanced Functional Programming*, volume 1129 of *LNCS*, pages 184–207. Springer Verlag, 1996.

Edward R. Sykes and Franya Franek. A Prototype for an Intelligent Tutoring System for Students Learning to Program in Java. *Advanced Technology for Learning*, 1(1), 2004.

Simon Thompson. *The Craft of Functional Programming*. Addison-Wesley, 1999.

Gareth Thorburn and Glenn Rowe. Pass: an automated system for program assessment. *Computers & Education*, 29(4):195–206, 1997.

Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.

Nghi Truong, Paul Roe, and Peter Bancroft. Static analysis of students' Java programs. In *ACE 2004: Proceedings of the sixth conference on Australasian computing education*, pages 317–325. Australian Computer Society, Inc., 2004.

Ian Utting, Stephen Cooper, Michael Kölling, John Maloney, and Mitchel Resnick. Alice, greenfoot, and scratch – a discussion. *ACM Transactions on Computing Education*, 10(4):17:1–17:11, 2010.

Jeroen J. G. Van Merriënboer and Paul A. Kirschner. *Ten Steps to Complex Learning: A Systematic Approach to Four-component Instructional Design*. Lawrence Erlbaum Associates, 2007.

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(3-4):375–413, 2010.

Kurt VanLehn. *Mind Bugs – The Origins of Procedural Misconceptions*. The MIT Press, 1990.

Kurt VanLehn. The behavior of tutoring systems. *International Journal on Artificial Intelligence in Education*, 16(3):227–265, 2006.

Kurt Vanlehn, Collin Lynch, Kay Schulze, Joel A. Shapiro, Robert Shelby, Linwood Taylor, Don Treacy, Anders Weinstein, and Mary Wintersgill. The Andes Physics Tutoring System: Lessons Learned. *International Journal on Artificial Intelligence in Education*, 15:147–204, 2005.

Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building Program Optimizers with Rewriting Strategies. In *ICFP 2008: Proceedings of the 13th ACM SIGPLAN international conference on Functional Programming*, pages 13–26, 1998.

Philip Wadler. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *POPL 1987: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 307–313, 1987.

L. Xu and A. Sarrafzadeh. Haskell-Tutor: An Intelligent Tutoring System for Haskell Programming language. *Postgraduate Conference of the Institute of Information and Mathematical Sciences*, 2004.

Songwen Xu and Yam San Chee. Transformation-Based Diagnosis of Student Programs for Programming Tutoring Systems. *IEEE Transansactions on Software Engineering*, 29(4):360–384, 2003.

Claus Zinn. Supporting tutorial feedback to student help requests and errors in symbolic differentiation. In M. Ikeda, K. Ashley, and T.-W. Chan, editors, *ITS 2006: Proceedings of the 8th international conference on Intelligent Tutoring Systems*, volume 4053 of *LNCS*, pages 349–359. Springer Verlag, 2006.

List of acronyms

ACP	algebra of communicating processes. 25, 27
AJAX	asynchronous Javascript and XML. 91
AST	abstract syntax tree. 29, 30, 93, 112, 113, 121, 123–125, 138, 139
CAS	computer algebra system. 73
CFG	context-free grammar. 19, 22–24, 36
CGI	common gateway interface. 82, 83
CMU	Carnegie Mellon university. 43
DME	digital math environment. 15, 18
DTD	document type definition. 110
EBNF	extended Backus-Naur form. 28, 29
EDSL	embedded domain-specific language. 16, 18, 19, 29, 35, 44, 90, 115
GHC	Glasgow Haskell compiler. 93, 155
HTTP	hypertext transfer protocol. 82

ITS	intelligent tutoring system. 2, 5, 57, 68, 73–76, 81–84, 152
JSON	Javascript object notation. 81, 82
LCD	least common denominator. 21, 23, 32, 64, 78
NBE	normalisation by evaluation. 134
RPC	remote procedure call. 81, 82
TRS	term rewrite system. 60
UML	unified modelling language. 35
URL	uniform resource locator. 82, 83
XML	extensible markup language. 81, 91, 110, 177

INDEX