A programming tutor for Haskell

Johan Jeuring

Alex Gerdes

Bastiaan Heeren

A programming tutor for Haskell

Johan Jeuring[1,2], Alex Gerdes[1], and Bastiaan Heeren[1]

[1]School of Computer Science, Open Universiteit Nederland
P.O.Box 2960, 6401 DL Heerlen, The Netherlands
{jje, age, bhr}@ou.nl
[2] Department of Information and Computing Sciences, Universiteit Utrecht

Abstract. In these lectures we will introduce an interactive system that supports writing simple functional programs. Using this 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,
  – get suggestions about how to refactor their program.
The system itself is implemented as a functional program, and uses fundamental concepts such as rewriting, parsing, strategies, program transformations and higher-order combinators such as the fold. We will introduce these concepts, and show how they are used in the implementation of the interactive functional programming tutor.

1 Introduction

How do you write a functional program? How can I learn it? Our answer to these questions depends on who is asking. If it is a first-year bachelor computer science student who just finished an introductory object-oriented programming course, we would start with explaining the basic ideas of functional programming, and set many small functional programming exercises for the student to solve. If it is a starting computer science Ph.D. student with a basic knowledge of functional programming, we would take a serious piece of software developed in a functional programming language, analyse it, discuss the advanced concepts used in the implementation, and set a task in which the software is extended or changed. These answers are based on our (and others) experience as teachers: there is no final answer (yet) to the question how programming is learned best, and what makes programming hard [Fincher and Petre, 2004]. We borrow from research that studies how complex cognitive skills are learned, in which the importance of providing worked-out examples [Merriënboer and Paas, 1990], giving hints, and giving immediate feedback on actions of students [Hattie and Timperley, 2007] is emphasised.

These lecture notes address the question 'How do you write a functional program' with the audience of advanced graduate students or starting Ph.D. students in mind. The serious piece of software addresses the same question:

'How do you write a functional program?', but now with a first-year bachelor student computer science in mind. We will introduce an intelligent functional programming tutoring system for Haskell [Peyton Jones et al., 2003], using which a student can:

- develop a program incrementally,
- receive feedback about whether or not she is on the right track,
- ask for a hint when she is stuck,
- can see how a complete program is stepwise constructed,
- get suggestions about how to refactor her program.

As far as we are aware, this is the first intelligent tutoring system for Haskell.

The implementation of the intelligent functional programming tutor uses many advanced functional programming concepts. To support incremental development of programs and refactoring, the tutor uses rewrite and refinement rules. To give feedback about whether or not a student is on the right track the tutor uses strategies to describe the various solutions, and parsing to follow the student's behaviour. To give hints to a student that is stuck, the system uses several analysis functions on strategies, viewing a strategy as a context-free grammar. These notes will introduce all of these concepts.

These notes are organised as follows. Section 2 introduces our intelligent functional programming tutor by means of some example interactions. Section 3 gives the architecture of the software for the tutor. Section 4 discusses rewrite and refinement rules and shows how they are used in the tutor. Section 5 introduces strategies for solving functional programming problems. Section 6 introduces our strategy language. Section 7 shows how we use techniques from parsing to follow student behaviour, and to give hints to a student that is stuck. Section 8 discusses related and future work, and concludes.

2   A programming tutor for Haskell

This section introduces our intelligent functional programming tutoring system by means of some interactions of a hypothetical student with the tutor. The functional programming tutor is an example of an intelligent tutoring system for the domain of functional programming. An intelligent tutoring system is an environment that sets tasks for a student, and offers support to the student when solving these tasks, by means of hints, corrections, and worked-out solutions [VanLehn, 2006]. So the intelligent functional programming tutor sets small functional programming tasks, and gives feedback in interactions with the student.

2.1   Reverse

Elisa just started a course on functional programming, and has attended lectures on how to write simple functional programs on lists. Her teacher has set
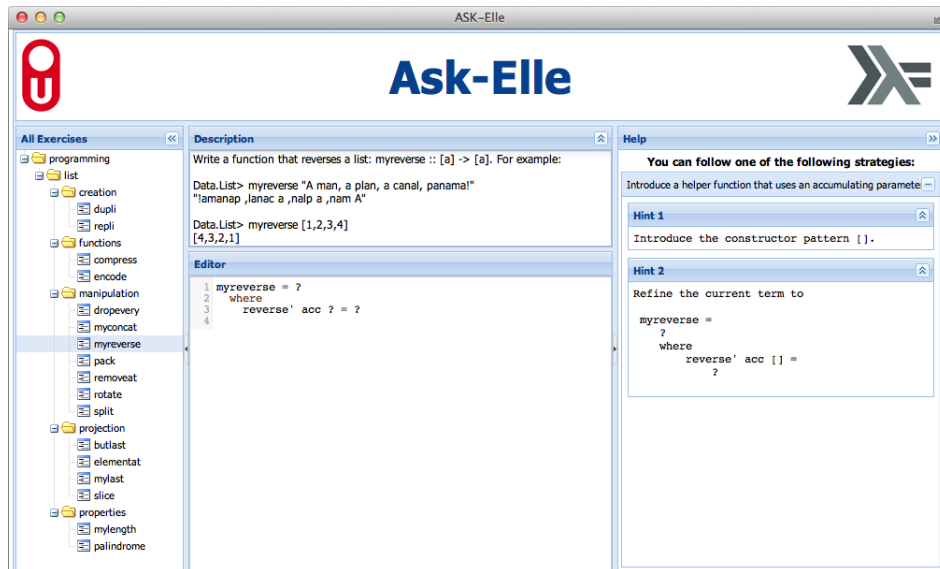
Fig. 1. The web-based functional programming tutor

a couple of exercises from H-99: Ninety-nine Haskell Problems [1], in particular problem 5: reverse a list.

We now show a couple of possible scenarios in which Elisa interacts with the tutor to solve this problem. A screenshot of the tutor is shown in Figure 1. At the start of a tutoring session the tutor gives a problem description: Write a function that reverses a list. For example:

Data.List⟩ reverse `"A man, a plan, a canal, panama!"`
`"!amanap ,lanac a ,nalp a ,nam A"`
Data.List⟩ reverse $[1, 2, 3, 4]$
$[4, 3, 2, 1]$

and displays the name of the function to be defined:

reverse $= \bot$

The task of a student is to refine the incomplete parts, denoted by $\bot$, of the program. The symbol $\bot$ 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 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

_____

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

3

tutor for a hint. Besides holes, a student can also introduce new declarations, function bindings, alternatives, and refine patterns.

Suppose Elisa 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.
>   – Implement reverse using the *foldl* function.
>   – 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 Elisa 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. Elisa 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:
>
> $reverse = reverse' \perp$
>   **where** $reverse'\ acc = \perp$

At this point, Elisa 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 Elisa chooses to implement $reverse'$ by pattern matching on the second argument, which is a list, starting with the empty list case:

> $reverse = reverse'\ [\,]$
>   **where**
> $reverse'\ acc\ [\,] = \perp$

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:

> $reverse = reverse'\ [\,]$
>   **where**
> $reverse'\ acc\ [\,] = [\,]$

The tutor responds with:

Incorrect [] in the right hand side of reverse′ on line 3

Correcting the error, Elisa enters:

$reverse = reverse'\ []$
  where
    $reverse'\ acc\ [] = acc$

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

Define the non-empty list case of reverse′

She continues with

$reverse = reverse'\ []$
  where
    $reverse'\ acc\ [] \qquad = acc$
    $reverse'\ acc\ (x:xs) = \perp$

which is accepted, and then

$reverse = reverse'\ []$
  where
    $reverse'\ acc\ [] \qquad = acc$
    $reverse'\ acc\ (x:xs) = reverse'\ (y:acc)\ \perp$

which gives:

Error: undefined variable y

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

$reverse = reverse'\ []$
  where
    $reverse'\ acc\ [] \qquad = acc$
    $reverse'\ acc\ (x:xs) = reverse'\ (x:acc)\ xs$

Done! You have correctly solved the exercise.

The third model solution described at an abstract level in the hint at the start of the exercise is the naive, quadratic time solution for *reverse*, implemented by means of an explicit recursive definition:

$reverse\ [] \qquad = []$
$reverse\ (x:xs) = reverse\ xs \mathbin{+\!\!+} [x]$

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.

5

2.2 Integers within a range

The next example we show is problem 22 from the Haskell 99 questions: Create a list containing all integers within a given range. For example:

> *Data.List⟩ range* 4 9
> $[4,5,6,7,8,9]$

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

> *range x y = unfoldr* $(\lambda i \rightarrow$ **if** *i* == *succ y* **then** *Nothing* **else** *Just* $(i, succ\ i))\ x$

This solution uses the *unfoldr* function defined by:

> *unfoldr* :: $(b \rightarrow Maybe\ (a, b)) \rightarrow b \rightarrow [a]$
> *unfoldr f b* = **case** *f b* **of**
>   *Just* $(a, new\_b) \rightarrow a : unfoldr\ f\ new\_b$
>   *Nothing* $\rightarrow []$

Our system prefers the solution using *unfoldr*. If a student asks for a worked-out solution, the system would respond with the derivation given in Figure 2.

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, such as a parse error, a dependency error (such as using an unbound variable), or a type error has been made, and where the error appears to be,
- show a complete worked-out example.

3 The architecture of the tutor

Our tutor can be accessed via a browser[2]. On the main page, a student selects an exercise to work on (such as *reverse*). The tutor provides a starting point ($\bot$), and the student can then start refining the $\bot$ step-wise to a complete program that implements reverse. While developing the program, a student can check that (s)he 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.

The feedback that we offer, such as giving a hint, is derived from a *strategy*. Strategies have 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 outlines how to incrementally construct a program. We have developed an embedded

---

[2] http://ideas.cs.uu.nl/ProgTutor/

$$range = \bot$$
$\Rightarrow$ { Introduce parameters }
$$range\ x\ y = \bot$$
$\Rightarrow$ { Use *unfoldr* }
$$range\ x\ y = unfoldr \bot \bot$$
$\Rightarrow$ { Start at *x* }
$$range\ x\ y = unfoldr \bot x$$
$\Rightarrow$ { Introduce a lambda-abstraction }
$$range\ x\ y = unfoldr\ (\lambda i \rightarrow \bot)\ x$$
$\Rightarrow$ { Introduce an **if-then-else** to specify a stop criterion }
$$range\ x\ y = unfoldr\ (\lambda i \rightarrow \textbf{if } \bot \textbf{ then } \bot \textbf{ else } \bot)\ x$$
$\Rightarrow$ { Introduce the stop criterion }
$$range\ x\ y = unfoldr\ (\lambda i \rightarrow \textbf{if } i \text{ == } succ\ y \textbf{ then } \bot \textbf{ else } \bot)\ x$$
$\Rightarrow$ { Return *Nothing* for the stop criterion }
$$range\ x\ y = unfoldr\ (\lambda i \rightarrow \textbf{if } i \text{ == } succ\ y \textbf{ then } Nothing \textbf{ else } \bot)\ x$$
$\Rightarrow$ { Give the output value and the value for the next iteration }
$$range\ x\ y = unfoldr\ (\lambda i \rightarrow \textbf{if } i \text{ == } succ\ y \textbf{ then } Nothing \textbf{ else } Just\ (i, succ\ i))\ x$$

Fig. 2. Derivation of the definition of *range*

domain-specific language for defining such strategies. Our strategy language is described in detail in Section 6.

The feedback functionality, which is based on strategies, is provided to external environments as a web-service. Each time a student clicks a button such as Check or Hint, our programming environment (the front-end) sends a service request [Gerdes et al., 2008] to our functional programming domain reasoner (the back-end). 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 (the strategy for *reverse*), and the previous and new expression of the student to the *diagnose* feedback-service. The following table describes the most relevant feedback services:

allfirsts. The *allfirsts* service returns all next steps that are allowed by a strategy.

onefirst. The *onefirst* service returns a single possible next step that follows a strategy. The functional programming domain reasoner offers the possibility to specify an order on steps, to select a single step among multiple possible steps.

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

finished. The *finished* service checks whether or not the program is accepted as a final answer.

stepsremaining. The *stepsremaining* service returns the number of steps that remain to be done according to the strategy. This is achieved by calculating the length of the derivation returned by the *derivation* service.

diagnose. The *diagnose* service diagnoses a program submitted by a student.

The diagnose feedback-service (and all our other feedback-services) 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-mistake, or develops a wrongly typed program. If a student submits a syntax- and type-correct program, we analyse the submitted program using the *diagnose*-service.

The diagnose-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. The diagnose service is flexible in the sense that a student may use different names for locally defined entities, and different syntactic constructs for the same expression (**let** versus **where**, and many other equivalences). The diagnose-service calculates a normal form of both the expected and the submitted programs, and checks that the submitted program appears in the set of expected programs. If the submitted program appears in the set of expected programs, the tutor accepts the step, and responds positively. If it doesn't, the tutor checks if the program can be recognised by any of the known wrong approaches (typical erroneous solutions that we have encountered in student solutions), and if it can reports this to student. Finally, if the student program cannot be recognised the student is asked to try again[3].

4 Rewriting and refining

As the examples in the Section 2 show, 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 him to complete the program, or to give a student a program, and ask him to change the program at a particular point. In such assignments, a student *refines* or *rewrites* a program. Rewriting preserves the semantics of a program; refining possibly makes a program more precise. This section discusses how students can refine and rewrite functional programs.

We offer a number of refinement rules to students. Section 2 already gives some examples:

$$\bot \Rightarrow \lambda \bot \to \bot \qquad \text{Introduce lambda abstraction}$$
$$\bot \Rightarrow \textbf{if } \bot \textbf{ then } \bot \textbf{ else } \bot \qquad \text{Introduce \textbf{if-then-else}}$$
$$\bot \Rightarrow v \qquad \text{Introduce variable } v$$

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

---

[3] In Section 8 on future work we explain how we intend to relax this restriction in the future

declaration, as in

$$\bot \Rightarrow f \bot = \bot \qquad \text{Introduce function binding}$$

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 type. 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*
        | *Hole*

and more. A refinement rule takes the same number of arguments as its abstract syntax constructor. So the refinement rule introducing an **if-then-else** expression takes three expression arguments. The arguments may be holes or terms containing holes. As another example, the 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, the refinement rule that introduces a variable takes the name of that variable (such as v in Figure 3) as an argument, and returns an expression that does not contain a hole anymore. Note that the variable introduced might be bound or unbound; refinement rules are unaware of the binding structure of the language. The refinement rules are kept simple and basically encapsulate a constructor.

A refinement rule refines a program on the level of the context-free syntax, and not on the level of tokens, so, for example, we don't have a rule that says *the*$\bot \Rightarrow$ **then**.

Holes are the central concept in our refinement rules. Where can they appear? Refinement rules refine:

– expressions, such as the $\bot$ in $\lambda i \rightarrow \bot$,
– declarations (the second $\bot$ in *reverse* = *reverse'* $\bot$ **where** $\bot$),
– function bindings ($f\ [\ ] = 0 \Rightarrow f\ [\ ] = 0; f\bot = \bot$),
– alternatives (**case** *xs* **of** $[\ ] \rightarrow 0 \Rightarrow$ **case** *xs* **of** $[\ ] \rightarrow 0; \bot \rightarrow \bot$),
– patterns (**case** *xs* **of** $\bot \rightarrow \bot \Rightarrow$ **case** *xs* **of** $[\ ] \rightarrow \bot$).

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 4 using the context-free grammar for Haskell takes many steps, but a student would only

Declarations
patBind:	$\perp$	$\Rightarrow$	$\perp = \perp$
funBinds:	$\perp$	$\Rightarrow$	$\perp$
			$\perp$

Function bindings
funBind:	$\perp$	$\Rightarrow$	$f\perp = \perp$

Expressions
var:	$\perp$	$\Rightarrow$	v
lit:	$\perp$	$\Rightarrow$	l
app:	$\perp$	$\Rightarrow$	$\perp\,\perp$
lambda:	$\perp$	$\Rightarrow$	$\lambda\perp \rightarrow \perp$
*case_*:	$\perp$	$\Rightarrow$	**case** $\perp$ **of** $\perp$

Alternatives
alt:	$\perp$	$\Rightarrow$	$\perp \rightarrow \perp$

Patterns
pVar:	$\perp$	$\Rightarrow$	v
pWildcard:	$\perp$	$\Rightarrow$	$\_$

Fig. 3. Some refinement rules for functional programming in Haskell

see $\perp \Rightarrow 4$. 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. These refinement rules 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. Preferably, the refinement rules are derived from looking at interactions of students in an editor, but lacking a tutor, we use our experience as programmers and teachers as a first approximation of the set of desirable refinement rules. We list some refinement rules that are often used in Figure 3.

Some refinement steps are performed silently, and are combined with one or more other refinement steps in a hint. For example, 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. Our domain reasoner offers the possibility to annotate a rule that is performed silently, by declaring

it as a *minor* rule. We use these minor rules to increase the step size, and avoid showing steps like $\perp \Rightarrow \perp \perp$. 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. Minor rules are not only used for increasing the step size, to avoid showing steps that make no sense to students, but also to perform administrative tasks, such as modifying an environment that we maintain next to the program.

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

$$reverse = reverse' \; [\,]$$
$$\quad \textbf{where}$$
$$\quad\quad reverse' \; acc \; [\,] \quad\quad = acc$$
$$\quad\quad reverse' \; acc \; (x:xs) = reverse' \; (x:acc) \; xs$$
$$\Rightarrow \quad \{ \text{ Definition of } flip \}$$
$$reverse = reverse' \; [\,]$$
$$\quad \textbf{where}$$
$$\quad\quad reverse' \; acc \; [\,] \quad\quad = acc$$
$$\quad\quad reverse' \; acc \; (x:xs) = reverse' \; (flip \; (:) \; acc \; x) \; xs$$
$$\Rightarrow \quad \{ \text{ Definition of } foldl \}$$
$$reverse = reverse' \; [\,]$$
$$\quad \textbf{where}$$
$$\quad\quad reverse' \; acc = foldl \; (flip \; (:)) \; acc$$
$$\Rightarrow \quad \{ \text{ Inline and } \beta\text{-reduce} \}$$
$$reverse = foldl \; (flip \; (:)) \; [\,]$$

To support such an exercise, each of our rewrite rules works both ways: we can remove *flip* (:) by applying the definition of *flip*, or we can introduce it as in the above derivation.

In the code for the tutor[4], rules are specified in the file `Domain/FP/Rules.hs`. The rules are specified as functions taking terms, which may contain holes, as arguments. The rule *introCase* looks as follows:

$$case_- :: Expr \rightarrow [Alt] \rightarrow Rule \; Expr$$
$$case_- \; e =$$
$$\quad toRefinement \; \texttt{"Introduce case"} \; \texttt{"case"} \circ Case \; e$$

where *Case* is a constructor of the datatype *Expr*, and *toRefinement* turns a description (`"Introduce case"`), an identifier (`"case"`), and a value of some type into a *Rule* of that type. The precise definitions of the *Rule*, *Expr* and *Alt* datatypes are not important for these notes.

---

[4] See `http://ideas.cs.uu.nl/trac/wiki/Download`.

5 Strategies in functional programming

The basic steps for constructing a solution for a programming task are program refinement rules introduced in the previous section. These rules typically replace an unknown part, a hole, by some term. A program refinement rule can introduce one or more new unknown parts. We are finished with an exercise as soon as all unknown parts have been completed. How do we guide a student in making progress to a complete solution?

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.

Our tutor is built for the domain of functional programming. It supports reasoning about functional programs by means of refinement and rewrite rules. The knowledge about how to guide this 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 have chosen to use this term.

A strategy for a functional program describes how a student should 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 specification 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. To support program refinement in a tutor, we limit the solutions that are accepted by the tutor.

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, 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.

This section introduces strategies, and shows how we formulate strategies for functional programming.

5.1 Strategies for procedural skills

A procedure often consists of multiple steps. For example, developing a function for *reverse* requires developing the complete program, which in the case of an explicit recursive definition consists 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 procedure may also contain a choice between different (sequences of) steps. For example, we can choose to either use *foldl*, or an explicit recursive definition for *reverse*. 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, biology, and programming. The basic elements of the strategy language are rewrite rules or refinement rules. We use rewrite rules for exercises in mathematics, logic, and biology. Refinement rules are rewrite rules that may contain holes, and are used in the programming domain. The strategy language supports combining strategies by means of *strategy combinators*. For example, if s and t are strategies, then so are:

$s <\vert> t$	choice: do either s or t
$s <\!*\!> t$	sequence: do s before t
$s <\%> t$	interleave: steps from s and t are performed in some order

Furthermore, we have a strategy *fail*, which always fails (the unit of choice), and a strategy *succeed*, which always succeeds (the unit of sequence). Section 6 gives a complete description of our strategy language and combinators. The contents of this section is not necessary for understanding the contents of Section 6, and readers interested in the strategy language only can skip this section.

5.2 Strategies for functional programs

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. A model solution is compiled into a programming strategy by inspecting its abstract syntax tree (AST), and matching the refinement rules with the AST. This is a standard tree matching algorithm, which yields a strategy that can later be adapted by a teacher

13

for his own purposes. The strategies for the various model solutions are then combined into a single strategy using the choice combinator. So, for the *reverse* exercise from Section 2 we would get a single strategy combining the three strategies for the model solutions. For example, here is a strategy that is compiled 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 when we give a hint for this part of the program, we don't just introduce parentheses, but also the function *flip* (and the invisible application operator, which is also introduced by a minor rule). 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., 2010], 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 programming problems is feasible for beginning programmers. It is rather uncommon that a beginning programmer develops a new model solution for a beginners' problem.

We discuss how to recognise as many variants of a solution as possible in the next two subsections. Subsection 5.3 describes strategies for Haskell's prelude, and Subsection 5.4 discusses a canonical form of Haskell programs.

5.3 Strategies for Haskell's prelude

To recognise as many syntactic variants as possible of (a part of) a solution to a programming problem, we describe a strategy for functional programming at a high abstraction level. For example, we define special strategies *foldlS* and *flipS* for recognising occurrences of *foldl* and *flip* in programs. The strategy *flipS* not only recognises *flip* itself, but also its definition, which can be considered an inlined and β-reduced version of *flip*. The strategy *flipS* takes a strategy as argument, which recognises the argument of *flip*.

$$
\begin{aligned}
\textit{flipS fS} \;=\; & \textit{app} \mathbin{<*>} \textit{var } \texttt{"flip"} \mathbin{<*>} \textit{fS} \\
& \mathbin{<|>} \textit{lambda} \mathbin{<*>} \textit{pVar x} \mathbin{<*>} \textit{pVar y} \\
& \qquad\qquad \mathbin{<*>} \textit{app} \mathbin{<*>} \textit{fS} \mathbin{<*>} (\textit{var y} \mathbin{<\%>} \textit{var x})
\end{aligned}
$$

The variable names x and y, used in the lambda-abstraction, are fresh and do not appear free in *fS*, in order to avoid variable capturing. The *flipS* (*con* `"(:)"`) strategy recognises both *flip* (:) itself, and the β-reduced, infix constructor, form $\lambda xs\ x \rightarrow x : xs$. The *flipS* strategy is used in a strategy *reverseS* for a model solution for *reverse* as follows:

$$
\begin{aligned}
&\textit{reverseS} = \textit{foldlS } (\textit{paren} \mathbin{<*>} \textit{flipS consS})\ \textit{nilS} \\
&\quad \textbf{where} \\
&\qquad \textit{consS} = \textit{infixApp} \mathbin{<*>} \textit{con } \texttt{"(:)"} \\
&\qquad \textit{nilS}\ \ = \textit{con } \texttt{"[]"}
\end{aligned}
$$

It is important to specify model solutions for exercises using abstractions available in Haskell's prelude like *foldl*, *foldr*, *flip*, etc, if applicable. In the *reverseS* example we have for example used both *foldlS* and *flipS*. 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. A large part of the Haskell prelude is available in our strategy language. 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 $\lambda f\ g\ x \rightarrow f\ (g\ x)$. The strategies for functions in the prelude also contain some conversions between abstractions, such as

$$
\textit{foldl op e} \mathrel{\texttt{==}} \textit{foldr } (\textit{flip op})\ e \circ \textit{reverse}
$$

So, if a function that is specified by means of a *foldl* is implemented by means of a *foldr* together with *reverse*, this is also accepted. Of course, students can introduce their own abstractions.

A strategy cannot capture all variations of a program that a student can introduce. 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

15

a student the possibility to use her own variable names. We use *normalisation* to handle such kinds of variations. If a student introduces her own *foldr*, possibly using a different name, normalisation will successfully compare this against a model solution using the prelude's *foldr*.

5.4 A canonical form for Haskell programs

The diagnose service checks whether or not a student submissions follows a strategy. 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 (which is also passed to the diagnose service), normalise the programs thus obtained, and compare each of these programs against the normalised submitted student program. Using normalisation, which returns a canonical form of a program, 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:

$$\lambda x \rightarrow f\ x \Rightarrow f$$

Normalisation uses various program transformations to reach a canonical form of a Haskell program. We use amongst others inlining, α-renaming, β- and η-reduction, and desugaring program transformations. 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 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.

In the remainder of this section we show some of the program transformations and discuss the limitations of our normalisation.

Desugaring. Desugaring removes syntactic sugar from a program. Syntactic sugar is usually introduced to conveniently write some kind of programs, such as writing $\lambda x\ y \rightarrow \ldots$ for $\lambda x \rightarrow \lambda y \rightarrow \ldots$ Syntactic sugar does not change the semantics of a program. However, if we want to compare a student program syntactically against (possibly partially complete) model solutions we want to ignore syntactic sugar. Desugaring consists of several program transformations such as removing superfluous parentheses, rewriting a **where** expression to a **let** expression, 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:

$$reverse = foldl\ f\ [\ ]\ \textbf{where}\ f\ x\ y = y : x$$
$$\Rightarrow \quad \{\ \textbf{where}\ \text{to}\ \textbf{let}\ \}$$

$reverse = \textbf{let}\ f\ x\ y = y : x\ \textbf{in}\ foldl\ f\ [\,]$
$\Rightarrow\quad \{\ \text{Infix operators to (prefix) functions}\ \}$
$reverse = \textbf{let}\ f\ x\ y = (:)\ y\ x\ \textbf{in}\ foldl\ f\ [\,]$
$\Rightarrow\quad \{\ \text{Function bindings to lambda abstractions}\ \}$
$reverse = \textbf{let}\ f = \lambda x \to y \to (:)\ y\ x\ \textbf{in}\ foldl\ f\ [\,]$

In the following paragraph on inlining we will see how the declaration of *f* is inlined in the *foldl*-expression.

Inlining. Inlining replaces a call to a user-defined function by its body. We perform inlining to make β-reduction possible. For example,

$reverse = \textbf{let}\ f = \lambda x \to \lambda y \to (:)\ y\ x\ \textbf{in}\ foldl\ f\ [\,]$
$\Rightarrow\quad \{\ \text{Inline}\ \}$
$reverse = foldl\ (\lambda x \to \lambda y \to (:)\ y\ x)\ [\,]$

Constant arguments. An argument is constant if it is passed unchanged to *all* recursive function calls. Compilers often optimise such constant arguments away, to save space and increase speed. Consider the following naive implementation of the higher-order function *foldr*:

$foldr\ op\ b\ [\,]\quad\ \ = b$
$foldr\ op\ b\ (x : xs) = x\ `op`\ foldr\ op\ b\ xs$

This implementation has two constant arguments: *op* and *b*. A better implementation is:

$foldr\ op\ b = f$
$\quad\textbf{where}\ f\ [\,]\quad\ \ = b$
$\qquad\qquad\ f\ (x : xs) = x\ `op`\ f\ xs$

The above definition is the standard definition for *foldr* from the Haskell prelude. 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. Recursive functions are rewritten in terms of *fix*, which does not get β-reduced. 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 help function *f* will not be inlined.

Lambda calculus reductions. At the heart of our normalisation are program transformations based on the λ-calculus.

17

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, due to the fact that 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:

$$(\lambda f\ x\ y \to f\ y\ x)\ (:) \Rightarrow \lambda x\ y \to (:)\ y\ x$$

Although we don't expect a student to write a program containing a β-redex, this happens in practice.

Discussion. 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* $3\ [1 \mathinner{.\,.}]$ 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.

We could extend our normalisation procedure with several other transformations, such as permuting function arguments, or swapping components of pairs, but haven't done so yet.

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.

5.5 Relating strategies to locations in programs

A program is constructed incrementally, in a top-down fashion. When starting the construction of a program there is usually 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 base argument expression in an *foldl* application cannot be applied to an arbitrary expression hole, but should be applied at exactly the location where the argument is needed in the program. In the next example this is the second expression hole (counted from left to right):

$$foldl\ (flip\ \bot)\ \bot \Rightarrow foldl\ (flip\ \bot)\ some\_argument$$

To target a particular location in a program, every refinement rule is extended with information about the location of the hole it refines. 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 holes that appear in the refinement rules to the strategies that are used to refine these holes. For example, the holes introduced by the *app* refinement rule need to be connected to rules that refine them. 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:

$$app :: Expr \rightarrow Expr \rightarrow Rule\ Expr$$
$$app\ f\ x = toRefinement\ \texttt{"Introduce application"}\ \texttt{"app"}\ (App\ f\ x)$$

The *app* refinement rule applies *App* to two expression holes. These holes should be connected to the rules that are going to refine them. The first might for example be a *var* :: *String* → *Expr* refinement rule that introduces a prelude function, as in *var* `"length"`. The hole expression and the *var* rule have to be connected. We achieve this connection by giving a hole an *identifier* and specialising a rule only to be 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:

$$\textbf{type}\ HoleID = Int$$
$$\textbf{data}\ Expr\quad = Hole\ HoleID\ |\ \ldots$$

When combining refinement rules in a strategy, we do not only specify the refinement rule, but also the identifier of the hole it is going to refine. We define a datatype that combines a term containing one or more holes and a strategy that refines the holes in that term:

$$\textbf{data}\ Refinement\ a = Ref\ \{\,term :: a,\ strat :: Strategy\ a\,\}$$

Here is an example value of this datatype for a strategy that introduces the application of the prelude function *length* to the empty list $[\,]$:

$$Ref\ (App\ (Hole\ 1)\ (Hole\ 2))\ (var_1\ \texttt{"length"}\ <\!\%\!>\ con_2\ \texttt{"[]"})$$

The refinement rule is annotated with the identifier of the hole it should refine. So, the refinement rule var_1 is only applicable to a hole with identifier 1. The actual numbering of holes takes place in a state monad:

$$\textbf{type}\ RefinementS\ a = State\ Int\ (Refinement\ a)$$

After evaluating the state monad, every hole has a unique number. We define a function *bindRefinement* to bind a rule to a hole:

$$bindRefinement :: Rule\ a \rightarrow RefinementS\ a$$

This function takes a rule and returns, in the state monad, a term together with the strategy consisting of the argument rule. The *bindRefinement* function ensures that the refinement rule is applied to the right hole. We use generic programming techniques to locate a particular hole in an AST, but we omit the details.

Since strategy combinators combine rules, the combinators have to be aware of the relations between refinement rules and holes, and adapt them appropriately whenever rules are combined. For example, when combining two programming strategies by means of the choice combinator, both substrategies should refine the *same* hole. Since the concepts of refinement rules and holes are special for the programming domain, and do not appear in most of the other domains we have studied, we define lifted versions of the combinators introduced in Subsection 5.1 that deal with the relations between refinement rules and holes. For example, the lifted version of the choice combinator uses a 'plain' choice combinator to combine the substrategies, and updates the relation between holes and substrategies. A refinement rule is only applicable when the holes it refines are present in the AST. For instance, in the strategy for the application of the *length* function to the empty list, the *app* refinement rule is applied *before* the *var* $\texttt{"length"}$ 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.

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, due to the fact that the concrete syntax does not contain hole identifiers. Since we need holes with identifiers as specified in the strategy, we use information about the steps that a student has taken so far. We use the strategy and these steps to reconstruct the AST with

the correct hole identifiers, which we can compare against the program of the student. Reconstructing the AST is easy because information about previous steps is maintained and communicated back and forth between the front- and back-end.

6 A strategy language

In the previous section we introduced strategies and strategy combinators informally. This section defines the semantics of these combinators, and the laws they satisfy. Our strategy language is very similar to the language for specifying context-free grammars (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., 2008, Heeren and Jeuring, 2008, 2009, 2010, 2011].

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 refinement or rewrite rules. We use $a, b, c, \ldots$ 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, by interpreting it as a context-free grammar.

6.1 Rules

The basic components of our strategy language, the alphabet, are the rewrite and refinement rules. The language of a strategy consisting of a single rule is just that rule:

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

6.2 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:

$$\mathcal{L}\,(s <|> t) \;=\; \mathcal{L}\,(s) \cup \mathcal{L}\,(t)$$

The *fail* combinator is a strategy that always fails. Its set of sentences is empty:

$$\mathcal{L}\,(fail) \;=\; \varnothing$$

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

$$fail <|> s \;=\; s$$
$$s <|> fail \;=\; s$$

6.3 Sequence

Often, a program is developed in a particular order: when developing the application of a function to an argument, we usually first develop the function, and only then the argument. So if *fS* is a strategy for developing *f*, and *eS* is a strategy for developing *e*, to develop *f e*, we first perform *fS* and then *eS*. Thus the development of this program follows a particular order. The *sequence* combinator, denoted by $\Leftrightarrow$, 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:

$$\mathcal{L}\ (s \Leftrightarrow t)\ =\ \{\,xy \mid x \in \mathcal{L}\ (s), y \in \mathcal{L}\ (t)\,\}$$

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

$$\mathcal{L}\ (succeed)\ =\ \{\epsilon\}$$

The *fail* combinator is a zero element of $\Leftrightarrow$, and *succeed* is a unit element:

$$
\begin{aligned}
fail \Leftrightarrow s &= fail \\
s \Leftrightarrow fail &= fail \\[1em]
succeed \Leftrightarrow s &= s \\
s \Leftrightarrow succeed &= s
\end{aligned}
$$

6.4 Interleave

In a **case**-expression like

> **case** *xs* **of**
>   [] $\rightarrow \perp$
>   *x* : *xs* $\rightarrow \perp$

a student may refine any of the two right-hand sides, in any order. She may even interleave the refinement of the two right-hand sides. 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:

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

22

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 field traditionally defines interleave in terms of left-interleave (and "communication interleave") to obtain a sound and complete axiomatisation [Fokkink, 2000].

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

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

The set *abc* $\%>$ *de* (where *abc* and *de* are now sentences) only contains the six sentences that start with symbol *a*. It is worth noting that the number of inter-leavings for two sentences of lengths n and m equals $\frac{(n+m)!}{n!m!}$. This number grows quickly with longer sentences. An alternative definition of interleaving two sequences, presented by Hoare in his influential book on CSP [Hoare, 1985], is by means of three laws:

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

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:

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

For instance, $\{a, ab\} <\%> \{c, cd\}$ 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 $(X \%> Y) \%> Z$ is equal to $X \%> (Y <\%> Z)$.

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, $a \langle bc \rangle \text{ <%> } \langle de \rangle f$ 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:

$$\begin{aligned}
\mathcal{L} \left(\langle s \rangle \right) &= \{ \langle x \rangle \mid x \in \mathcal{L} \left(s \right) \} \\
\mathcal{L} \left(s \text{ <%> } t \right) &= \mathcal{L} \left(s \right) \text{ <%> } \mathcal{L} \left(t \right) \\
\mathcal{L} \left(s \text{ %> } t \right) &= \mathcal{L} \left(s \right) \text{ %> } \mathcal{L} \left(t \right)
\end{aligned}$$

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

$$\begin{aligned}
s \text{ <%> } t &= t \text{ <%> } s \\
s \text{ <%> } (t \text{ <%> } u) &= (s \text{ <%> } t) \text{ <%> } u \\
s \text{ <%> } succeed &= s
\end{aligned}$$

Because interleaving distributes over choice

$$s \text{ <%> } (t \text{ <|> } u) = (s \text{ <%> } t) \text{ <|> } (s \text{ <%> } u)$$

we have a second semi-ring. Also left-interleave distributes over choice. The operator that makes a strategy atomic is idempotent, and distributes over choice $\langle s \text{ <|> } t \rangle = \langle s \rangle \text{ <|> } \langle t \rangle$. Many more properties can be found in the literature on ACP [Bergstra and Klop, 1985].

6.5 Label

When developing a program, a student may ask for a hint at any time. Of course, the tutor should take the actions of the student until he asks for a hint into account. 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.

$$\mathcal{L} \left(label \ \ell \ s \right) = \{ \text{ENTER}_\ell \ x \ \text{EXIT}_\ell \mid x \in \mathcal{L} \left(s \right) \}$$

This interpretation introduces the special rules ENTER and EXIT (parameterised by some label ℓ) that show up in sentences. These rules are minor rules that are only used for tracing positions in strategies. Except for tracing, the label combinator is semantically the identity function.

6.6 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 *fix* is defined by:

$$\mathcal{L}\ (\textit{fix } f) \ = \ \mathcal{L}\ (f\ (\textit{fix } f))$$

The *fix* operator is mainly used in traversals over the abstract syntax tree. It is the responsibility of the user to specify meaningful fixed-points. In our recogniser for the strategy language we specify a cutoff for the fixed-point operator.

6.7 Overview

We list the components of our strategy language introduced in the previous subsection in the following definition.

A strategy is an element of the language of the following grammar:

$$
\begin{array}{lll}
s & ::= & r \\
  & | & s <|> s \quad | \quad \textit{fail} \\
  & | & s <\!\!*\!\!> s \quad | \quad \textit{succeed} \\
  & | & \textit{label } \ell \ s \\
  & | & \textit{fix } f \\
  & | & \langle s \rangle \quad | \quad s <\%> s \quad | \quad s \%> s
\end{array}
$$

where r is a rewrite rule or a refinement rule, ℓ is a label, and f is a function that takes a strategy as argument, and returns a strategy.

The language of a strategy is defined by:

$$
\begin{array}{lll}
\mathcal{L}\ (r) & = & \{r\} \\
\mathcal{L}\ (s <|> t) & = & \mathcal{L}\ (s) \cup \mathcal{L}\ (t) \\
\mathcal{L}\ (\textit{fail}) & = & \varnothing \\
\mathcal{L}\ (s <\!\!*\!\!> t) & = & \{xy \mid x \in \mathcal{L}\ (s), y \in \mathcal{L}\ (t)\} \\
\mathcal{L}\ (\textit{succeed}) & = & \{\epsilon\} \\
\mathcal{L}\ (\textit{label } \ell \ s) & = & \{\text{ENTER}_\ell\ x\ \text{EXIT}_\ell \mid x \in \mathcal{L}\ (s)\} \\
\mathcal{L}\ (\textit{fix } f) & = & \mathcal{L}\ (f\ (\textit{fix } f)) \\
\mathcal{L}\ (\langle s \rangle) & = & \{\langle x \rangle \mid x \in \mathcal{L}\ (s)\} \\
\mathcal{L}\ (s_1 <\%> s_2) & = & \mathcal{L}\ (s_1) <\%> \mathcal{L}\ (s_2) \\
\mathcal{L}\ (s_1 \%> s_2) & = & \mathcal{L}\ (s_1) \%> \mathcal{L}\ (s_2)
\end{array}
$$

This definition can be used to tell whether a sequence of rules follows a strategy or not: the sequence of rules should be a sentence in the language generated by the strategy, or a prefix of a sentence, since we solve exercises incrementally. Not all sequences make sense, however. An exercise gives us an initial term (say t_0), and we are only interested in sequences of rules that can be applied successively to this term. Suppose that we have terms (denoted by t_i) and rules (denoted by r_i), and let t_{i+1} be the result of applying rule r_i to term t_i by means of function *apply*. Function *apply* takes a refinement or a rewrite rule and a term, tries to unify the term with the left-hand side of the rule, and, if it succeeds, applies the substitution obtained from unification to the right-hand side of the rule to obtain the rewritten or refined rule. A possible derivation that starts with t_0 can be depicted in the following way:

$$t_0 \xrightarrow{r_0} t_1 \xrightarrow{r_1} t_2 \xrightarrow{r_2} t_3 \xrightarrow{r_3} \ldots$$

To be precise, applying a rule to a term can yield multiple results, but most domain rules, such as the refinement rules for functional programs in Figure 3, return at most one term. Running a strategy with an initial term returns a set of terms, and is specified by:

$$run\ s\ t_0 = \{t_{n+1} \mid r_0 .. r_n \in \mathcal{L}(s), \forall_{i \in 0...n} : t_{i+1} \in apply\ r_i\ t_i\}$$

Recognising a strategy amounts to tracing the steps that a student takes, but how does a tutor get the sequence of rules? In a tutor that offers free input, such as our functional programming tutor, students submit intermediate terms. Therefore, the tutor first has to determine which of the known rules has been applied, or even which combination of rules has been used. Discovering which (sequence of) rule(s) has been used is obviously an important part of a tutor, and it influences the quality of the generated feedback. It is, however, not the topic of these notes, more information can be found in Gerdes et al. [2012]. An alternative to free input is to let students select a rule, which is then applied automatically to the current term. In this setup, it is no longer a problem to detect which rule has been used.

6.8 Applications of strategies in other domains

Using our strategy language we can specify strategies for an arbitrary domain in which procedures are expressed in terms of rewriting and refinement rules. In this subsection we introduce two examples not related to the domain of functional programming in which we use our strategy language. The first example shows a general pattern that occurs in many different domains, the second example describes a procedure for calculating with fractions.

Example 1. Repetition, zero or more occurrences of something, is a well-known recursion pattern. We can define this pattern using our fixed point recursion combinator:

Rules		**Buggy rules**	
ADD:	$\dfrac{a}{c} + \dfrac{b}{c} = \dfrac{a+b}{c}$	B1:	$\dfrac{a}{b} + \dfrac{c}{d} \neq \dfrac{a+c}{b+d}$
MUL:	$\dfrac{a}{b} \times \dfrac{c}{d} = \dfrac{a \times c}{b \times d}$	B2:	$a \times \dfrac{b}{c} \neq \dfrac{a \times b}{a \times c}$
RENAME:	$\dfrac{b}{c} = \dfrac{a \times b}{a \times c}$	B3:	$a + \dfrac{b}{c} \neq \dfrac{a+b}{c}$
SIMPL:	$\dfrac{a+b}{b} = 1 + \dfrac{a}{b}$		

Fig. 4. Rules and buggy rules for fractions

many s = fix $(\lambda x \rightarrow succeed <|> (s \Leftrightarrow x))$

The strategy that applies transformation rule *r* zero or more times would thus be:

many r
  = succeed <|> (r ⟨⋆⟩ many r)
  = succeed <|> (r ⟨⋆⟩ (succeed <|> (r ⟨⋆⟩ many r)))
  = . . .

Example 2. Consider the problem of adding two fractions, for example, $\frac{2}{5}$ and $\frac{2}{3}$: if the result is an improper fraction (the numerator is larger than or equal to the denominator), then it should be converted to a mixed number. Figure 4 displays four rewrite rules on fractions. The three rules at the right (B1 to B3) are buggy rules that capture common mistakes. 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

We use the strategy combinators to turn this informal strategy description into a strategy specification:

addFractions = label ℓ_0 (*label* ℓ_1 LCD
                    ⟨⋆⟩ *label* ℓ_2 (*repeat* (*somewhere* RENAME))
                    ⟨⋆⟩ *label* ℓ_3 ADD
                    ⟨⋆⟩ *label* ℓ_4 (*try* SIMPL)
)

The strategy contains the labels ℓ_0 to ℓ_4, and uses the transformation rules given in Figure 4. The transformation LCD is somewhat different: it is a minor rule

that does not change the term, but calculates the least common denominator and stores this in an environment. The rule RENAME for renaming a fraction uses the computed lcd to determine the value of *a* in its right-hand side.

The definition of *addFractions* uses the strategy combinators *repeat*, *try*, and *somewhere*. In an earlier paper [Heeren et al., 2008], we discussed how these combinators, and many others, can be defined conveniently in terms of the strategy language. The combinator *repeat* is a variant of the *many* combinator: it applies its argument strategy exhaustively. The check that the strategy can no longer be applied is a minor rule. The *try* combinator takes a strategy as argument, and tries to apply it. If the strategy cannot be applied, it succeeds.

The combinator *somewhere* changes the focus in an abstract syntax tree by means of one or more minor navigation rules, before it applies its argument strategy. The navigation rules are inspired by the operations on the zipper data structure [Huet, 1997]. These rules, usually called DOWN (go to the left-most child), RIGHT, LEFT, and UP, are used to navigate to a point of focus. Until now we have used holes to denote locations in terms, instead of a zipper. Using navigation rules and the zipper in the functional programming domain is less convenient. Whereas the strategy for adding fractions given above applies to any fraction, so that we do not know up front where our rules will be applied, our functional programming strategies describe the construction of a particular functional program, and recognise the construction of alternative versions. A strategy for a functional program describes exactly where all substrategies should be applied. For example, for the strategy for *reverse*, only the second argument to *foldl* should be refined to the empty list [ ]. This refinement is not applied bottom-up or somewhere: it is exactly applied at the location of the second argument of *foldl*. We could have specified this location by means of navigation rules, in which case we would have obtained a strategy for *reverse* consisting of amongst others:

$$
\begin{array}{ll}
\ldots & \textit{foldlS} \\
& \text{<*> DOWN} \\
& \text{<*> (\textit{flipS consS})} \\
& \text{<*> RIGHT} \\
& \text{<*> \textit{nilS}} \\
& \text{<*> UP}
\end{array}
$$

and similarly for all other strategies. Since this information can be inferred automatically, as explained in Section 5, we use holes to denote locations in the functional programming domain.

6.9 Restrictions

To use strategies for tracking student behaviour and give feedback, we impose some restrictions on the form of strategies. These restrictions are similar to some of the restrictions imposed by parsing algorithms on context-free grammars.

Left-recursion. 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:

$$leftRecursive = \textit{fix} \ (\lambda x \rightarrow x \ \lessdot\!\!\ast\!\!\gtrdot \ \textsc{Add})$$

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

$$leftRecursive' = \textit{fix} \ (\lambda x \rightarrow \textsc{Down} \ \lessdot\!\!\ast\!\!\gtrdot \ x \ \lessdot\!\!\ast\!\!\gtrdot \ \textsc{Add})$$

the minor rule $\textsc{Down}$ is applied repeatedly until we reach the leaf of an expression tree, and stop. This strategy is not left-recursive. However, this is caused by a property of $\textsc{Down}$ that is not shared by all other minor rules.

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 (programs, but also derivations for other exercises). However, 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. Similarly, for a strategy to be used in our domain reasoner, it should not be left-recursive. In particular, 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].

Left-recursive strategies are not the only source of non-terminating strategy calculations. The fact that our strategy language has a fixed-point combinator (and hence recursion) implies that we are vulnerable to non-termination. The implementation of our strategy language has been augmented with a 'time-out' that stops the execution of a strategy when a threshold is reached, and reports an error message.

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 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.

At the moment, a strategy that contains left-factors may lead to problems. Consider the following, somewhat contrived, strategy:

$$leftFactor \;=\; label\ \ell_1\ (\text{ADD} <\!\!*\!\!> \text{SIMPL})$$
$$<\!|\!> label\ \ell_2\ (\text{ADD} <\!\!*\!\!> \text{RENAME})$$

The two sub-strategies labelled ℓ_1 and ℓ_2 have a left-factor: the rewrite rule ADD. After the application of ADD, we have to decide which sub-strategy to follow. Either we follow sub-strategy ℓ_1, or we follow sub-strategy ℓ_2. Committing to a choice after recognising that ADD has been applied is unfortunate, since it will force the student to follow the same sub-strategy. For example, if ℓ_1 is chosen after a student applies ADD, and a student subsequently performs the RENAME step, we erroneously report that that step does not follow the strategy. Left-factoring a strategy is essential to not commit early to a particular sub-strategy. The example strategy is left-factored as follows:

$$leftFactor' = \text{ADD} <\!\!*\!\!> (\text{SIMPL} <\!|\!> \text{RENAME})$$

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,

$$leftFactor'' = \text{ADD} <\!\!*\!\!> (label\ \ell_1\ \text{SIMPL} <\!|\!> label\ \ell_2\ \text{RENAME})$$

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 text, which very likely becomes inaccurate if labels are moved automatically.

At the moment we require strategies to be left-factored, so that we can decide which production to take based on the next input symbol, as in LL(1) grammars. However, this is very undesirable, since it makes it hard if not impossible to generate functional programming strategies from model solutions. We intend to use parallel top-down recursive parsing techniques to solve this problem. If we encounter a left-factor, i.e., the *firsts* set contains duplicates, we fork the parser into two or more parsers, depending on the number of duplicates, that run in parallel. Whenever a parsing branch fails, it is discarded. We have started implementing this approach, and the first results indicate that it is indeed possible to solve this problem [Gerdes et al., 2012].

7   Design of a strategy recogniser

The function *run*, defined in the previous section, specifies how to run a strategy. For this, it enumerates all sentences in the language of a strategy, and then applies the rules in such a sentence in sequence, starting with some initial term. Enumerating all sentences does not result in an efficient implementation because the number of sentences quickly becomes too large, making this approach

infeasible in practice. Often, the language of a strategy is an infinite set. In our domain reasoners we take a different, more efficient approach to recognise student steps against a strategy definition. In this section we discuss the design of such a strategy recogniser.

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 around [Hutton, 1992, Swierstra and Duponcheel, 1996], 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.
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 very undesirable in learning environments.
4. Labels help to describe the structure of a strategy in the same way as nonterminals 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. Current parsing libraries do not offer parser combinators for interleaving parsers, except for a (rather involved) extension implemented by Doaitse Swierstra on top of his parser combinator library [Swierstra, 2009].
6. A strategy should be serialisable, for instance because we want to communicate with other on-line 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.

7.1 Representing grammars

Because strategies are 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 6, 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.

Example 3. The repetition combinator *many*, which we defined in Example 1, can be encoded with the *Grammar* datatype in the following way:

many :: *Grammar a* → *Grammar a*
many s = *Rec* 0 (*Succeed* :|: (*s* :*: *Var* 0))

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

7.2 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: *empty* $(s) = \epsilon \in \mathcal{L}(s)$. 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:

```
empty :: Grammar a → Bool
empty (Symbol a) = False
empty Succeed    = True
empty Fail       = False
empty (s :‖: t)      = empty s ∨ empty t
empty (s :⋇: t)      = empty s ∧ empty t
empty (s :%: t)      = empty s ∧ empty t
empty (s :%>: t)   = False
empty (Atomic s) = empty s
empty (Rec i s)    = empty s
empty (Var i)       = False
```

The left-interleave operator expresses that the first symbol is taken from its left-hand side operand. Hence, such a strategy cannot yield the empty sentence. The definition for the pattern *Rec i s* may come as a surprise: it calls *empty* recursively on *s* without changing the *Var*s that are bound by this *Rec*. We define *empty* (*Var i*) to be *False*. Note that there is no need to inspect recursive occurrences to determine the empty property for a strategy.

Given some strategy *s*, the function *firsts* returns every symbol that can start a sentence for *s*, paired with a strategy that represents the remainder of that sentence. This is made more precise in the following property (where *a* represents a symbol, and *x* a sequence of symbols):

$$\forall a, x : ax \in \mathcal{L}\ (s) \Leftrightarrow \exists s' : (a, s') \in firsts\ (s) \land x \in \mathcal{L}\ (s')$$

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 *s* %> *t* based on the firsts for *s* and *t* (i.e., in a compositional way) since that would require more information about the atomic blocks in *s* and *t*. For a strategy *s* %> *t*, we split *s* into an atomic part and a remainder, say *Atomic s'* <⋇> *s''*. After *s'* without the empty sentence, we can continue with *s''* <%> *t*. This approach is summarised by the following property, where the use of symbol *a* takes care of the non-empty condition:

$$(\langle a \Leftrightarrow s \rangle \Leftrightarrow t)\ \%> u = \langle a \Leftrightarrow s \rangle \Leftrightarrow (t \Leftrightarrow\%> u)$$

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

```
split :: Grammar a → [(a, Grammar a, Grammar a)]
split (Symbol a) = [(a, Succeed, Succeed)]
```

$$
\begin{aligned}
split\ Succeed\ &=\ [\,] \\
split\ Fail\ &=\ [\,] \\
split\ (s :\!|\!:\ t)\ &=\ split\ s \mathbin{+\!\!+} split\ t \\
split\ (s :\!*\!:\ t)\ &=\ [(a, x, y :\!*\!:\ t) \mid (a, x, y) \leftarrow split\ s] \mathbin{+\!\!+} \\
&\qquad \textbf{if}\ empty\ s\ \textbf{then}\ split\ t\ \textbf{else}\ [\,] \\
split\ (s :\!\%\!:\ t)\ &=\ split\ (s :\!\%\!>\!:\ t) \mathbin{+\!\!+} split\ (t :\!\%\!>\!:\ s) \\
split\ (s :\!\%\!>\!:\ t)\ &=\ [(a, x, y :\!\%\!:\ t) \mid (a, x, y) \leftarrow split\ s] \\
split\ (Atomic\ s)\ &=\ [(a, x :\!*\!:\ y, Succeed) \mid (a, x, y) \leftarrow split\ s] \\
split\ (Rec\ i\ s)\ &=\ split\ (replaceVar\ i\ (Rec\ i\ s)\ s) \\
split\ (Var\ i)\ &=\ error\ \text{"unbound Var"}
\end{aligned}
$$

For a sequence $s :\!*\!:\ t$, we determine which symbols can appear first for s, and we change the results to reflect that t is part of the remaining grammar. Furthermore, if s can be empty, then we also have to look at the *firsts* for t. 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 *firsts* for *Rec i s*, we have to look inside the body s. 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 s t* replaces all free occurrences of *Var i* in t by s. Hence, if we encounter a *Var*, it is unbound, which we do not allow. Recall that we assume our grammars to be closed.

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

– *Case (Atomic s).* Because atomicity distributes over choice, we can consider the elements of *split s* (the recursive call) one by one. The transformation

$$\langle\langle a \mathbin{<\!*\!>} x\rangle \mathbin{<\!*\!>} y\rangle \ =\ \langle a \mathbin{<\!*\!>} (x \mathbin{<\!*\!>} y)\rangle \mathbin{<\!*\!>} succeed$$

is proven by first removing the inner atomic block, and basic properties of sequence.
– *Case ($s_1 :\!\%\!:\ s_2$).* Expressing this strategy in terms of left-interleave is justified by the definition of $\mathcal{L}\ (s_1 \mathbin{<\!\%\!>} s_2)$. For function *split*, we only have to consider the non-empty sentences.
– *Case ($s_1 :\!\%\!>\!:\ s_2$).* Left-interleave can be distributed over the alternatives. Furthermore, $(\langle a \mathbin{<\!*\!>} x\rangle \mathbin{<\!*\!>} y) \mathbin{\%\!>} t = \langle a \mathbin{<\!*\!>} x\rangle \mathbin{<\!*\!>} (y \mathbin{<\!\%\!>} t)$ 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 most of our feedback services:

$$
\begin{aligned}
&firsts :: Grammar\ a \rightarrow [(a, Grammar\ a)] \\
&firsts\ s = [(a, x :\!*\!:\ y) \mid (a, x, y) \leftarrow split\ s]
\end{aligned}
$$

In Section 6.9 we discussed restrictions that are imposed on strategies. It should now be clear from the definition of *firsts* why left-recursion is problematic. 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 *s* accepts the empty sentence, then running the grammar *many s* can result in non-termination. The problem with left recursion can be partially circumvented by restricting the number of recursion points (*Rec*s and *Var*s) that are unfolded in the definition of *split* (*Rec i s*).

7.3 Dealing with labels

The *Grammar* datatype lacks an alternative for labels. Nevertheless, we can use label information to trace where we are in the 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.

   The sentences of the language generated for a strategy contain rules, ENTER steps, and EXIT steps, for which we introduce the following datatype:

   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 only a string. The type *Rule* is parameterised by the type of values on which the rule can be applied. 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*

The (un)wrapping of 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.

35

data *LabeledStrategy a = Label* {*labelInfo* :: *LabelInfo, unlabel* :: *Strategy a* }

instance *IsStrategy LabeledStrategy* **where**
   toStrategy (*Label a s*) = *fromStep* (*Enter a*) *<*> s <*> fromStep* (*Exit a*)

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

7.4   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 grammars. We introduce a smart constructor for every alternative of the strategy language given in Section 6.7. Definitions for *succeed* and *fail* are straightforward, and are given for consistency:

succeed, fail :: *Strategy a*
succeed = *S Succeed*
fail   = *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. It is the context that will turn the 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* $\Rightarrow$ *LabelInfo* $\rightarrow$ *f a* $\rightarrow$ *LabeledStrategy a*
label s = *Label s* $\circ$ *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.

   -- Lift a unary/binary function on grammars to one on strategies
lift1 op = *S* $\circ$ *op* $\circ$ *unS* $\circ$ *toStrategy*
lift2 op = *lift1* $\circ$ *op* $\circ$ *unS* $\circ$ *toStrategy*

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

(*<|>*) :: (*IsStrategy f, IsStrategy g*) $\Rightarrow$ *f a* $\rightarrow$ *g a* $\rightarrow$ *Strategy a*
(*<|>*) = *lift2 op*
  where
    op :: *Grammar a* $\rightarrow$ *Grammar a* $\rightarrow$ *Grammar a*

```
op Fail    t    = t
op s       Fail = s
op (s ⫴: t) u   = s ‘op’ (t ‘op’ u)
op s       t    = s ⫴: t
```

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 t       = t
    op s       Succeed = s
    op Fail    _       = Fail
    op _       Fail    = Fail
    op (s ⋇: t) u      = s ‘op’ (t ‘op’ u)
    op s       t       = s ⋇: t
```

The binary combinators for interleaving, <%> and %>, are defined in a similar fashion. The smart constructor *atomic*, which was denoted by ⟨·⟩ in Section 6, 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 s) = op s
    op (s ⫴: t)   = op s ⫴: op t
    op s          = Atomic s
```

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 much more convenient to write the recursion as a fixed-point, without worrying about the numbering. 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
```

$$is \quad = usedNumbers\ (unS\ (make\ 0))$$
$$i \quad = \textbf{if}\ null\ is\ \textbf{then}\ 0\ \textbf{else}\ maximum\ is + 1$$

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 (variable *is* of type $[Int]$). Based on this information, we can 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 inspect their argument.

Example 4. We return to Example 3, and define the repetition combinator *many* with the smart constructors. Observe that *many*'s argument is also overloaded because of the smart constructors.

$$many :: IsStrategy\ f \Rightarrow f\ a \rightarrow Strategy\ a$$
$$many\ s = fix\ \$\ \lambda x \rightarrow succeed <|> (s <\!\!*\!\!> x)$$

7.5 Running a strategy

So far, nothing specific about recognising strategies has been discussed. A strategy is a grammar over rewrite rules and *Enter* and *Exit* steps for labels. We first define a type class with the method *apply*: this function was already used in the *run* method defined in Section 6.7. It returns a list of results. Given that rules can be applied, we also give an instance declaration for the *Step* datatype, where the *Enter* and *Exit* steps simply 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    -- implementation provided in framework
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 (see Section 6.7 for *run*'s specification). The implementation is based on the functions *empty* and *firsts*.

$$run :: Apply\ f \Rightarrow Grammar\ (f\ a) \rightarrow a \rightarrow [a]$$
$$run\ s\ a = [a \mid empty\ s] +\!\!+ [c \mid (f,t) \leftarrow firsts\ s, b \leftarrow apply\ f\ a, c \leftarrow run\ t\ b]$$

The list of results returned by *run* consists of two parts: the first part tests whether *empty s* holds, and if so, it yields the singleton list containing the term a. The second part takes care of the non-empty alternatives. Let f be one of the symbols that can appear first in strategy s. We are only interested in f if it can

be applied to the current term *a*, yielding a new term *b*. We run the remainder of the strategy (that is, *t*) on this new term.

Now that we have defined the function *run* we can also make *Strategy* and *LabeledStrategy* instances of class *Apply*:

> **instance** *Apply Strategy* **where**
>   *apply = run ∘ unS*
>
> **instance** *Apply LabeledStrategy* **where**
>   *apply = apply ∘ toStrategy*

The function *run* can produce an infinite list. In most cases, however, we are only interested in a single result (and rely on lazy evaluation). The part that considers the empty sentence is put at the front to return sentences with few rewrite rules early. Nonetheless, the definition returns results in a depth-first manner. We define a variant of *run* which exposes breadth-first behaviour:

> *runBF :: Apply f ⇒ Grammar (f a) → a → [[a]]*
> *runBF s a = [a | empty s] : merge [runBF t b | (f, t) ← firsts s, b ← apply f a]*
>   **where** *merge = map concat ∘ transpose*

The function *runBF* produces a list of lists: results are grouped by the number of rewrite steps that have been applied, thus making explicit the breadth-first nature of the function. The helper-function *merge* merges the results of the recursive calls: by transposing the list of results, we combine results with the same number of steps.

7.6   Tracing a strategy

The *run* functions defined in the previous section do nothing with the labels. However, if we want to recognise (intermediate) terms submitted by a student, and report an informative feedback message if the answer is incorrect, then labels become important. Fortunately, it is rather straightforward to extend *run*'s definition, and to keep a trace of the steps that have been applied:

> *runTrace :: Apply f ⇒ Grammar (f a) → a → [(a, [f a])]*
> *runTrace s a =*
>   *[(a, []) | empty s] ++*
>   *[(c, (f : fs)) | (f, t) ← firsts s, b ← apply f a, (c, fs) ← runTrace t 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.

Example 5. We return to the strategy for adding two fractions (*addFractions*, defined in 6.8). 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. In fact, 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 much more informative to step through such a derivation and see the intermediate steps.

[	*Enter ℓ_0,*	*Enter ℓ_1,*	*Step* LCD,	*Exit ℓ_1,*	*Enter ℓ_2*
,	*Step down$_{(0)}$,*	*Step* <u>RENAME</u>,	*Step up,*	*Step down$_{(1)}$,*	*Step* <u>RENAME</u>
,	*Step up,*	*Step not,*	*Exit ℓ_2,*	*Enter ℓ_3,*	*Step* <u>ADD</u>
,	*Exit ℓ_3,*	*Enter ℓ_4,*	*Step* <u>SIMPL</u>,	*Exit ℓ_4,*	*Exit ℓ_0*
]					

The list has twenty steps, but only four correspond to actual steps from the derivation: the rules of those steps are underlined. The other rules are administrative: the navigation rules *up* and *down* are introduced by the *somewhere* combinator, whereas *not* comes from the use of *repeat*. Also observe that each *Enter* step has a matching *Exit* step. In principle, a label can be visited multiple times by a strategy.

The example clearly shows that we determine at each point in the derivation where we are in the strategy by enumerating the *Enter* steps without their corresponding *Exit* step. 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. For reporting textual messages, we use feedback scripts, which is explained in the next section.

7.7 Feedback scripts

All textual messages are declared in so-called *feedback scripts*. These scripts are external text files containing appropriate responses for various situations. Depending on the diagnosis that was made (e.g., a common mistake was recognised, or the submitted term is correct and complies with the specified strategy), a feedback message is selected from the script and reported back to the student. One of the criteria on which this selection can be based is the current location in the strategy, i.e., one of the labels in the strategy. Other selection criteria are the name of the rule that was recognised (possibly a buggy rule), or the submitted term being correct or not.

For the functional programming tutor, we give three levels of hints, which can be categorised as follows [Vanlehn et al., 2005]:

– *general:* a general, high-level statement about the next step to take;
– *specific:* a more detailed explanation of the next step in words;
– *bottom-out:* the exact next step to carry out, possibly accompanied with some literal code.

The level of the message is another available selection criterion in the feedback scripts. All textual messages are assigned to one of these three levels.

Having only static texts in the feedback scripts (that is, texts that appear verbatim in the script) severely 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 situation at hand. In this way, messages can for instance contain snippets of code from the original student program, or report on the number of steps remaining. Feedback scripts contain some more constructs to facilitate the writing of feedback messages, such as local string definitions and an import mechanism. These topics are work in progress, and lie outside the scope of these lecture notes.

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 (programming) exercises. A final benefit is that the support of multiple languages (as opposed to only English) comes quite natural, since each supported language can have its own feedback script.

8 Conclusions, related and future work

We have discussed the design and implementation of a tutoring system for functional programming. The distinguishing characteristics of our tutoring system 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 author feedback.
- correctness is based on provable equivalence to a model solution, based on normal forms of functional programs.

The tutoring system targets students at the starting academic, or possibly end high-school, level.

8.1 Related work

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]. Programming environments like Scratch and Alice 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 target beginning computer science students, who expect to work with real-life programming languages instead of 'toy' programming languages.

The Lisp tutor [Anderson et al., 1986] is an intelligent tutoring system that supports the incremental construction of Lisp programs. At any point in the development a student can only take a single next step, which makes the interaction style of the tutor a bit restrictive. Furthermore, adding new material to the tutor is still quite some work. Using our approach based on strategies, the interaction style becomes flexible, and adding exercises becomes relatively easy. Soloway [1985] describes programming plans for constructing Lisp programs. These plans are instances of the higher-order function *foldr* and its companions. Our work structures the strategies described by Soloway.

In tutoring systems for Prolog, a number of strategies for Prolog programming have been developed [Hong, 2004]. Hong also uses the *reverse* example to exemplify his approach to Prolog tutoring. Strategies are matched against complete student solutions, and feedback is given after solving the exercise. We expect that these strategies can be translated to our strategy language, and can be reused for a programming language like Haskell.

Our work resembles the top-down Pascal editors developed in the Genie project [Miller et al., 1994]. These series of editors provide structure editing support, so that student don't have to remember the particular syntax of a programming language. In our case students do have to write programs using the syntax of Haskell, but the intermediate steps are comparable. The Genie editors did not offer strategical support.

Our functional programming tutoring system grew out of a program assessment tool, which automatically assesses student programs based on model solutions [Gerdes et al., 2010] and program transformations to rewrite programs to normal form. Similar transformations have been developed for C++-like languages [Xu and Chee, 2003].

8.2   Future work

The functional programming tutor grew out of our work on assessing functional programs, and on providing feedback, mainly in learning environments for mathematics. The version presented at this school is the first public release of our tutor. We still need to work on several aspects.

First of all, we want to use the tutor in several courses, to receive feedback from students and teachers. We will start with obtaining feedback about usability and appreciation. For example, do the refinement rules we offer correspond to the refinement rules applied by students? At a later stage, we want to study the learning effect of our tutor together with researchers from the domain of learning sciences.

The restriction that a student cannot proceed if an intermediate solution does not follow a model solution is rather severe. This disallows, for example, a bottom-up approach to developing a program, where first a component is developed, without specifying how the component is used in the final solution. We want to investigate if we can specify properties for a program, which are used to check that a student solution is not provably wrong. We can then let a student go on developing a program as long as the properties specified cannot

be falsified. Once a student connects the developed components to the main program, strategy checking kicks in again to see if the program is equivalent to a model solution. This approach is orthogonal to our current approach: using our tutor we can ensure that a student solution is equivalent to a model solution, and hence correct. However, if a student does not implement a model solution, we don't know if the student is wrong. On the other hand, using property checking we can prove that a student solution is wrong, but the absence of property violations does not necessarily imply that the student program is correct. We would achieve a mixed approach if we determine the propagation of post-conditions to components in our rewrite rules, and verify that the composition of the rewrite rules performed by the student results in a proof that a specified post-condition holds for a given program. However, we would need to manually support a prover to construct the proof in many cases, which might not be desirable for beginning programmers.

Teachers prefer different solutions, and sometimes want students to use particular constructs when solving a programming exercise ('use *foldr* to implement a function to ...'). It is important to offer teachers the possibility to adapt the tutor. We see two ways in which teachers can adapt the tutor. First, additional equalities satisfied by a particular component of a model solution can be specified separately, and can then be used in the normal form calculation. Second, a teacher can annotate a model solution with 'switches' to enforce or switch off particular ways to solve a problem, or to change the order in which subproblems have to be solved. For example, a teacher may want to enforce usage of *foldr* instead of its explicit recursive alternative. Or a teacher may allow the interleaved development of the **then** and **else** branches of an **if-then-else** expression. We want to add these facilities to our tutoring system.

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.

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 think that our programming tutor is *language generic*, and we want to investigate the possibilities for automatically generating large parts of a programming tutor, based on a (probably annotated) grammatical description.

Bibliography

John R. Anderson, Frederick G. Conrad, and Albert T. Corbett. Skill acquisition and the LISP tutor. *Cognitive Science*, 13:467–505, 1986.

Ralph-Johan Back. A calculus of refinements for program derivations. Reports on Computer Science and Mathematics 54, Åbo Akademi, 1987.

Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In Bernhard Möller and John Tucker, editors, *Prospects for Hardware Foundations*, volume 1546 of *Lecture Notes in Computer Science*, pages 624–624. Springer Berlin / Heidelberg, 1998.

Jan A. Bergstra and Jan Willem Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

R.S. 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.

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.

Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18:453–457, 1975.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to design programs: an introduction to programming and computing*. MIT Press, Cambridge, MA, USA, 2002.

Andrzej Filinski and Henning Korsholm Rohde. A denotational account of untyped normalization by evaluation. In Igor Walukiewicz, editor, *Foundations of Software Science and Computation Structures*, volume 2987 of *Lecture Notes in Computer Science*, pages 167–181. Springer Berlin / Heidelberg, 2004.

Sally Fincher and Marian Petre, editors. *Computer Science Education Research*, 2004. RoutledgeFalmer.

Jeroen Fokker. Functional parsers. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

Wan Fokkink. *Introduction to Process Algebra*. Springer-Verlag, 2000. ISBN 354066579X.

Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and Sylvia Stuurman. Feedback services for exercise assistants. In Dan Remenyi, editor, *ECEL 2007: Proceedings of the 7th European Conference on e-Learning*, pages 402–410. Academic Publishing Limited, 2008. Also available as Technical report Utrecht University UU-CS-2008-018.

Alex Gerdes, Johan Jeuring, and Bastiaan Heeren. Using strategies for assessment of programming exercises. In Gary Lewandowski, Steven A. Wolfman, Thomas J. Cortina, and Ellen Lowenfeld Walker, editors, *SIGCSE*, pages 441–445. ACM, 2010.

Alex Gerdes, Johan Jeuring, and Bastiaan Heeren. Teachers and students in charge — using annotated model solutions in a functional programming tutor. Technical report, Utrecht University, Department of Computer Science, 2012. To appear.

Mark Guzdial. Programming environments for novices. In Sally Fincher and Marian Petre, editors, *Computer Science Education Research*. RoutledgeFalmer, 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'09*, volume 5625 of *LNCS*, pages 325–340. Springer-Verlag, 2009.

Bastiaan Heeren and Johan Jeuring. Adapting mathematical domain reasoners. In *Proceedings of MKM'10: the 9th MKM international conference*, LNCS, pages 315–330. Springer-Verlag, 2010.

Bastiaan Heeren and Johan Jeuring. Interleaving strategies. In *Proceedings of the Conference on Intelligent Computer Mathematics*, LNCS. Springer-Verlag, 2011.

Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning Haskell. In *Haskell 2003: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 62 – 71. ACM, 2003.

Bastiaan Heeren, Johan Jeuring, Arthur van Leeuwen, and Alex Gerdes. Specifying strategies for exercises. In *MKM 2008: Mathematical Knowledge management*, volume 5144 of *LNAI*, pages 430–445. Springer-Verlag, 2008.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, 1969.

C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., 1985. ISBN 0-13-153271-5.

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

Gérard Huet. Functional Pearl: The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

Graham Hutton. Higher-order Functions for Parsing. *Journal of Functional Programming*, 2(3):323–343, 1992.

L. 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.

Jeroen J.G. van Merriënboer and Fred G.W.C. Paas. Automation and schema acquisition in learning elementary computer programming: Implications for the design of practice. *Computers in Human Behavior*, 6:273–289, 1990.

Philip Miller, John Pane, Glenn Meter, and Scott Vorthmann. Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University. *Interactive Learning Environments*, 4(2):140–158, 1994.

Carroll Morgan. *Programming from specifications*. Prentice-Hall, Inc., 1990.

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

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.

S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, volume 1129 of *LNCS*, pages 184–207. Springer-Verlag, 1996.

S.D. Swierstra. Combinator parsing: A short tutorial. In *Language Engineering and Rigorous Software Development*, volume 5520 of *LNCS*, pages 252–300. Springer-Verlag, 2009.

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.

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.