A Programming Tutor for Haskell

Johan Jeuring

Joint work with Alex Gerdes and Bastiaan Heeren

Computer Science
Utrecht University and Open Universiteit Nederland

CEFP, Budapest, Hungary

June 2011

Open Universiteit
www.ou.nl

Learning to program

Learning to program is hard. We don't know exactly why, but:

- Beginners often have misconceptions about the syntax and semantics of a programming language
- Analysing and creating a model of the problem that can be implemented is difficult for a beginner
- Decomposing a complex problem into smaller subproblems requires experience
- Most compilers give poor error messages

Can we develop an environment that supports learning to program?

[CEFP: A Programming Tutor for Haskell]

A programming tutor for Haskell at CEFP?

- How do you write a functional program? How can I learn it?
- Answer depends on who is asking
- Beginners: practice with many small exercises, and learn from the feedback you get
- More experienced functional programmers: study a large software system, and refactor and extend it at several points.

These lectures: a programming tutor for Haskell targetting beginners, which has been implemented in Haskell, using quite a few advanced Haskell constructs.

Outline of presentation

Programming environments for novices

Programming tutors

An example

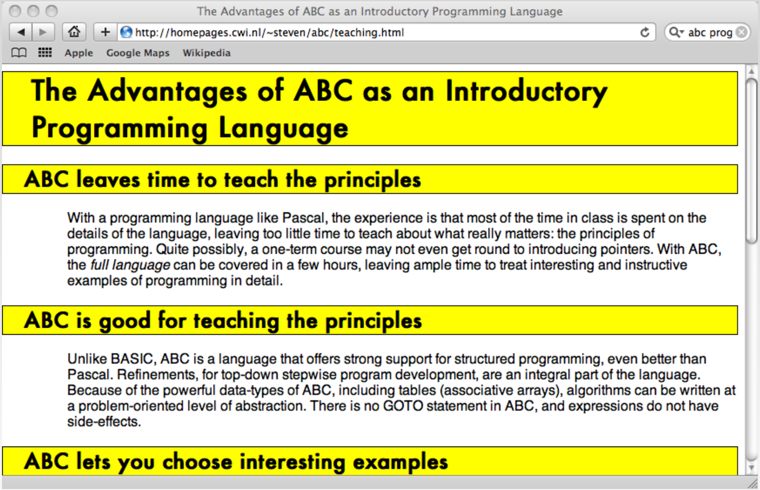
Strategies for programming

Wrap up

Programming environments for novices

[CEFP: A Programming Tutor for Haskell]

[CEFP: A Programming Tutor for Haskell]



The Advantages of ABC as an Introductory Programming Language

ABC leaves time to teach the principles

With a programming language like Pascal, the experience is that most of the time in class is spent on the details of the language, leaving too little time to teach about what really matters: the principles of programming. Quite possibly, a one-term course may not even get round to introducing pointers. With ABC, the *full language* can be covered in a few hours, leaving ample time to treat interesting and instructive examples of programming in detail.

ABC is good for teaching the principles

Unlike BASIC, ABC is a language that offers strong support for structured programming, even better than Pascal. Refinements, for top-down stepwise program development, are an integral part of the language. Because of the powerful data-types of ABC, including tables (associative arrays), algorithms can be written at a problem-oriented level of abstraction. There is no GOTO statement in ABC, and expressions do not have side-effects.

ABC lets you choose interesting examples

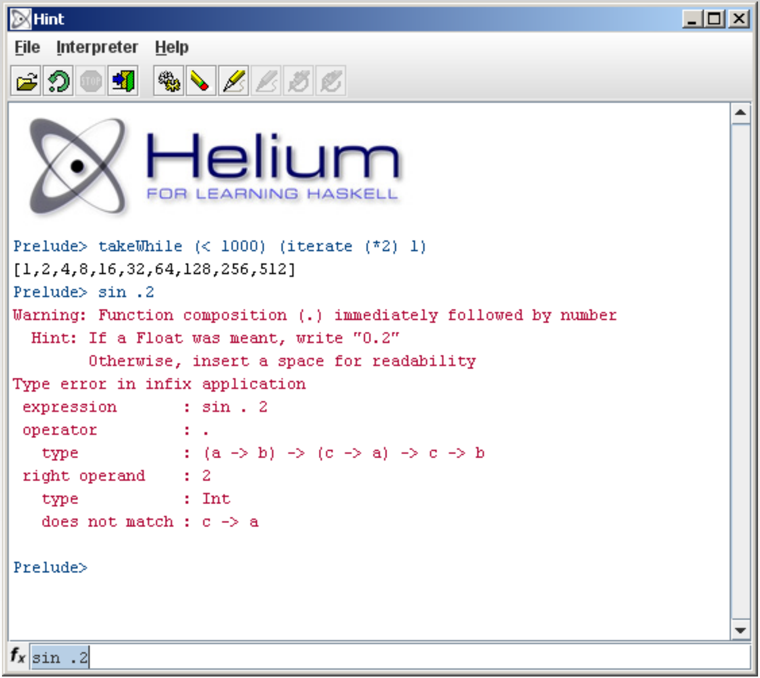
[CEFP: A Programming Tutor for Haskell]

An error message when using a Haskell compiler:

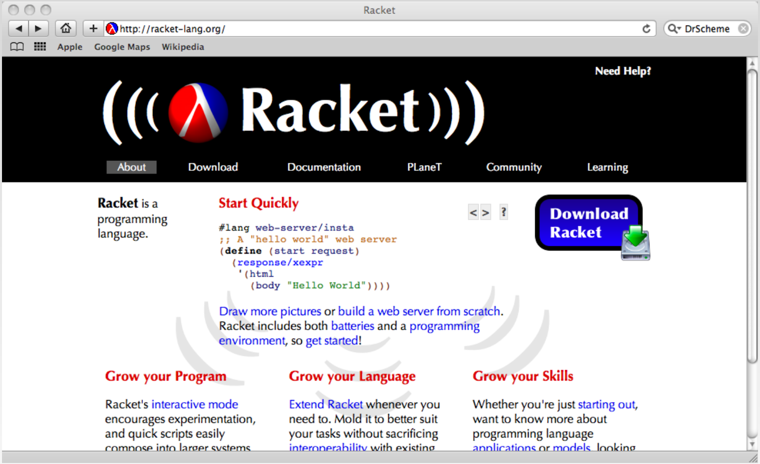
```
Prelude> let main = putChar 'a' >> putChar

<interactive>:1:26:
    Couldn't match expected type `IO b'
           against inferred type `Char -> IO ()'
    In the second argument of `(>>)', namely `putChar'
    In the expression: putChar 'a' >> putChar
    In the definition of `main': main = putChar 'a' >> putChar
```

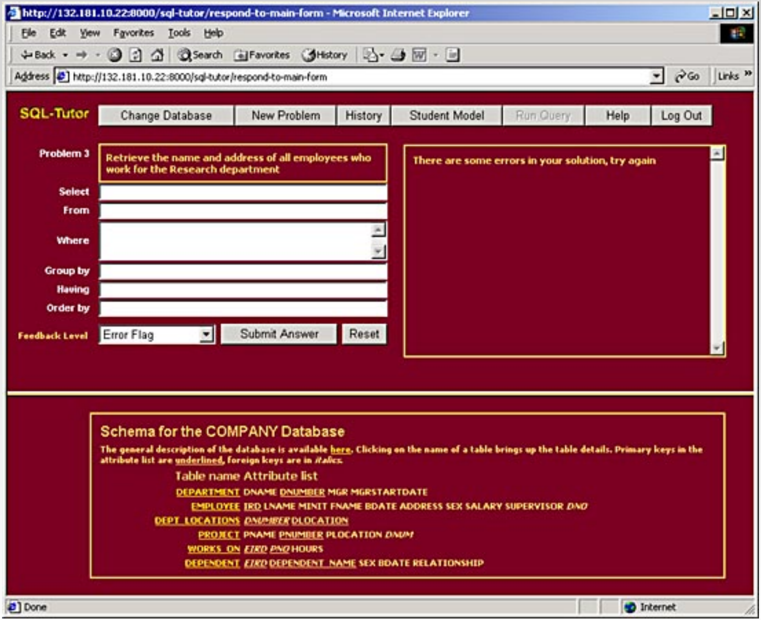
Mentioning that the function `putChar` is applied to too few arguments, is probably more helpful for novice programmers.

[CEFP: A Programming Tutor for Haskell]



[CEFP: A Programming Tutor for Haskell]


Racket
http://racket-lang.org/
DrScheme
Apple
Google Maps
Wikipedia
Need Help?
Racket
About
Download
Documentation
PLaneT
Community
Learning
Racket is a programming language.
Start Quickly
#lang web-server/insta
;; A "hello world" web server
(define (start request)
(response/xexpr
'(html
(body "Hello World"))))
Download Racket
Draw more pictures or build a web server from scratch.
Racket includes both batteries and a programming environment, so get started!
Grow your Program
Racket's interactive mode encourages experimentation, and quick scripts easily
Grow your Language
Extend Racket whenever you need to. Mold it to better suit your tasks without sacrificing
Grow your Skills
Whether you're just starting out, want to know more about programming language



[CEFP: A Programming Tutor for Haskell]

Programming environments for novices

Quite a few programming environments for novices have been developed:

- Scratch, Alice, and many predecessors
- ABC, Genie (structured editor for Pascal)
- Special editors for ‘mainstream’ programming languages
- Intelligent programming tutors

The categories of environments focus on (sometimes slightly) different aspects of problems in learning (to program):

- Wanting to learn
- Learning by doing
- Learning through feedback

Programming tutors

Programming tutors

Programming tutors focus on learning through feedback. Most tutors provide feedback to novice programmers by:

- giving hints (in varying level of detail)
- showing worked-out solutions
- reporting erroneous steps

So why aren't programming tutors used everywhere?

[CEFP: **A Programming Tutor for Haskell**]

Challenges for programming tutors

Despite the potential advantages, programming tutors are not widely used.

- Building a tutor is a substantial amount of work
- Using a tutor in a course is hard for a teacher: adapting or extending a tutor is often very difficult or even impossible
- Having to specify feedback with each new exercise is often a lot of work

Preferably, a programming tutor:

- supports easy specification of exercises
- automatically derives feedback and hints

Approach to program construction

Some well-known approaches to constructing correct programs are

- Use pre- and post-conditions to construct or verify a program
- Refine a specification to an executable program
- Transform a program to a program with some desirable properties

Our approach: construct a program that is provably equivalent to a model solution.

Amounts to program refinement, but with a post-condition expressed in terms of program equality.

Approach to program construction

Some well-known approaches to constructing correct programs are

- Use pre- and post-conditions to construct or verify a program
- Refine a specification to an executable program
- Transform a program to a program with some desirable properties

Our approach: construct a program that is provably equivalent to a model solution.

Amounts to program refinement, but with a post-condition expressed in terms of program equality.

A programming tutor for Haskell

We are developing a programming tutor for Haskell. Using the tutor, a student can:

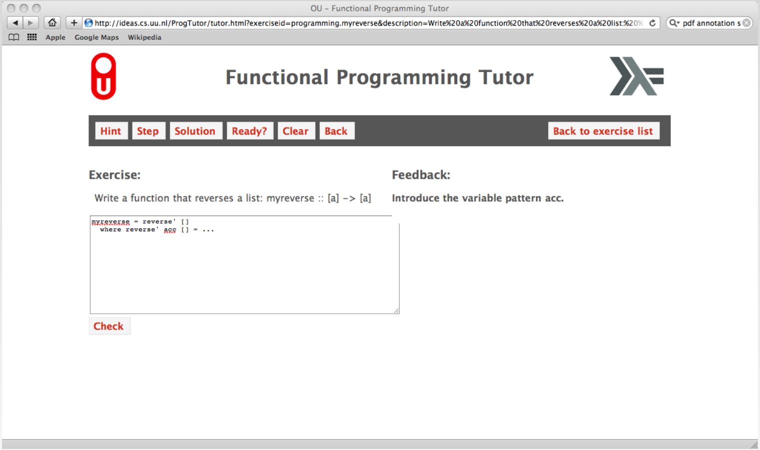
- develop her program incrementally, in a topdown fashion
- receive feedback about whether or not she is on the right track
- can ask for a hint when she is stuck
- see how a complete program is stepwise constructed

A teacher specifies an exercise by means of model solutions.

The tutor targets first-year computer science students.

Our tutor is an environment like ABC, with feedback functionality, for a 'mainstream' language.

[CEFP: **A Programming Tutor for Haskell**]



[CEFP: **A Programming Tutor for Haskell**]

An example

An example interactive session

Programming task: write a program that reverses a list:

$reverse = \langle ? \rangle$

Tutor response on Hint:

There are several ways you can proceed:

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

An example interactive session

Programming task: write a program that reverses a list:

$$
\begin{aligned}
&\mathit{reverse} = \mathit{reverse}' <?> \\
&\quad \textbf{where} \\
&\qquad \mathit{reverse}'\ \mathit{acc}\ <?> = <?>
\end{aligned}
$$

Tutor response on Hint:

Apply $\mathit{reverse}'$ to $[\,]$, or use pattern matching for the second argument of $\mathit{reverse}'$.

An example interactive session

Programming task: write a program that reverses a list:

$$\begin{aligned}
&\mathit{reverse} = \mathit{reverse}'\ [\,]\\
&\quad \textbf{where}\\
&\qquad \mathit{reverse}'\ \mathit{acc}\ [\,] = {<}?{>}
\end{aligned}$$

Tutor response on Hint:

Refine the right hand side of the empty list case.

An example interactive session

Programming task: write a program that reverses a list:

$$
\begin{aligned}
&\mathit{reverse} = \mathit{reverse}'\ [\,] \\
&\quad \textbf{where} \\
&\qquad \mathit{reverse}'\ \mathit{acc}\ [\,] = [\,]
\end{aligned}
$$

Tutor response on Check:

Incorrect $[\,]$ in the right hand side of $\mathit{reverse}'$ on line 3

An example interactive session

Programming task: write a program that reverses a list:

$$
\begin{aligned}
&\mathit{reverse} = \mathit{reverse}'\ [\,]\\
&\quad \textbf{where}\\
&\qquad \mathit{reverse}'\ \mathit{acc}\ [\,] = \mathit{acc}
\end{aligned}
$$

Tutor response on Hint:

Define the non-empty list case of $\mathit{reverse}'$

An example interactive session

Programming task: write a program that reverses a list:

$$
\begin{aligned}
&\mathit{reverse} = \mathit{reverse}'\ [\,]\\
&\quad \textbf{where}\\
&\qquad \mathit{reverse}'\ \mathit{acc}\ [\,] \qquad\quad = \mathit{acc}\\
&\qquad \mathit{reverse}'\ \mathit{acc}\ (x : xs) = \,<?>
\end{aligned}
$$

Tutor response on Hint:

Define the recursive call of *reverse′*

An example interactive session

Programming task: write a program that reverses a list:

$$
\begin{aligned}
&\mathit{reverse} = \mathit{reverse}'\ [\,]\\
&\quad \textbf{where}\\
&\qquad \mathit{reverse}'\ \mathit{acc}\ [\,] \qquad\quad = \mathit{acc}\\
&\qquad \mathit{reverse}'\ \mathit{acc}\ (x : \mathit{xs}) = \mathit{reverse}'\ (y : \mathit{acc})\ <?>
\end{aligned}
$$

Tutor response on Check:

```
Error: Undefined variable y
```

An example interactive session

Programming task: write a program that reverses a list:

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

Tutor response on Ready?:

Done! You have correctly solved the exercise.

Strategies for programming

Strategies for programming

- A strategy specifies how to incrementally refine a program
- We use a strategy to calculate all kinds of feedback
- We have developed a combinator language for strategies, using which we can develop and compose strategies

[CEFP: A Programming Tutor for Haskell]

reverse

There are several ways you can define the function $reverse :: [a] \to [a]$, which reverses a list of elements.

$$
\begin{aligned}
&reverse_1\ [\,] &&= [\,] \\
&reverse_1\ (x : xs) &&= reverse_1\ xs +\!\!+ [x]
\end{aligned}
$$

$$
\begin{aligned}
&reverse_2 = reverse_2'\ [\,] \\
&\quad \textbf{where}\ reverse_2'\ acc\ [\,] \qquad\quad = acc \\
&\qquad\qquad\quad reverse_2'\ acc\ (x : xs) = reverse_2'\ (x : acc)\ xs
\end{aligned}
$$

$$reverse_3 = foldl\ (flip\ (:))\ [\,]$$

Strategy example

The third program for *reverse*:

$$reverse_3 = foldl\ (flip\ (:))\ [\,]$$

is recognised by the strategy:

```
      patBind
<*> pVar "reverse"
<*> app <*> var "foldl"
        <*> (   (paren <*> app <*> var "flip"
                               <*> infixApp <*> con "(:)"
                )
            <%> con "[]"
            )
```

Representing strategies

Components of our strategy language:

1. Rewrite and refinement rules
2. Choice $\sigma \mathbin{<|>} \tau$
3. Sequence $\sigma \mathbin{<\star>} \tau$
4. Interleave $\sigma \mathbin{<\%>} \tau$
5. Unit elements *succeed*, *fail*
6. Labels *label* ℓ σ
7. Recursion *fix* f

- Labels are used to mark positions in a strategy
- Combinators are inspired by context-free grammars, and by the algebra of communicating processes.

Refinement rules

A refinement rule refines a hole.

Expression refinement rules:

$$\begin{array}{lll}
<?> \Rightarrow & \lambda<?> \to <?> & \text{-- Introduce lambda abstraction} \\
<?> \Rightarrow & \textbf{if} \quad <?> & \text{-- Introduce } \textbf{if}\text{-}\textbf{then}\text{-}\textbf{else} \\
 & \textbf{then} \ <?> & \\
 & \textbf{else} \ \ <?> & \\
<?> \Rightarrow & v & \text{-- Introduce variable } v
\end{array}$$

Declaration refinement rule:

$$<?> \Rightarrow f \ <?> = <?> \qquad \text{-- Introduce a function binding}$$

[CEFP: A Programming Tutor for Haskell]

Holes

- A hole (<?>) is a placeholder for an incomplete part of a program
- An exercise is finished when it does not contain holes anymore
- We have holes for the following constructs:
  - declarations, function bindings, expressions, alternatives, patterns

The abstract syntax is augmented with hole constructors.

$$
\begin{aligned}
\textbf{data}\ \mathit{Expr} = {} & \mathit{Lambda}\ \mathit{Pattern}\ \mathit{Expr} \\
\mid {} & \mathit{If}\ \mathit{Expr}\ \mathit{Expr}\ \mathit{Expr} \\
\mid {} & \mathit{App}\ \mathit{Expr}\ \mathit{Expr} \\
\mid {} & \mathit{Var}\ \mathit{String} \\
\mid {} & \mathit{Hole} \\
\mid {} & \ldots
\end{aligned}
$$

[CEFP: A Programming Tutor for Haskell]

Recognizing *flip*

For Haskell's prelude function *flip*:

$$\mathit{flip} = \lambda f\ x\ y \rightarrow f\ y\ x$$

we define the prelude strategy *flipS*, which takes a strategy *fS* recognising a function f, and recognises both:

$$\mathit{flip}\ f$$
$$\lambda x\ y \rightarrow f\ y\ x$$

which explains the implementation of *flipS*:

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

A strategy prelude

- We have defined a strategy prelude for functions in Haskell's prelude
- Besides definition and use, these strategies can also be used to recognise other variants, such as defining *foldl* in terms of *foldr*:

$$foldl\ op\ e \equiv foldr\ (flip\ op)\ e \circ reverse$$

Using the prelude

```
      patBind
<*> pVar "reverse"
<*> app <*> var "foldl"
        <*> (   (paren <*> app <*> var "flip"
                               <*> infixApp <*> con "(:)"
                )
          <%> con "[]"
            )
```

Becomes

```
      patBind
<*> pVar "reverse"
<*> foldlS (paren <*> flipS (infixApp <*> con "(:)"))
           (con "[]")
```

Program transformations

- Strategies derived from model solutions may be rather strict and reject equivalent but only slightly different programs
- Some of these differences cannot or should not be captured in a strategy, such as inlining a helper-function
- We use the program transformations η- and β-reduction, and α-conversion from the λ-calculus, to deal with such differences
- Additionally, we perform desugaring rewrite steps
- Of course, comparing two programs for equality is in general undecidable

Normalisation

Normalisation proceeds as follows:

1. α-conversion
2. desugaring/preprocessing steps
   - optimise constant arguments
   - inlining: replace an expression by its definition
   - rewrite infix notation to prefix
   - rewrite **where** to **let**
   - ...
3. β- and η-reduction

Normalisation example

$$reverse = foldl\ f\ [\,]\ \mathbf{where}\ f\ x\ y = y : x$$

$\Rightarrow$ { **where** to **let** }

$$reverse = \mathbf{let}\ f\ x\ y = y : x\ \mathbf{in}\ foldl\ f\ [\,]$$

$\Rightarrow$ { Infix operators to (prefix) functions }

$$reverse = \mathbf{let}\ f\ x\ y = (:)\ y\ x\ \mathbf{in}\ foldl\ f\ [\,]$$

$\Rightarrow$ { Function bindings to lambda abstractions }

$$reverse = \mathbf{let}\ f = \lambda x\ y \rightarrow (:)\ y\ x\ \mathbf{in}\ foldl\ f\ [\,]$$

$\Rightarrow$ { Remove multiple lambda abstraction arguments }

$$reverse = \mathbf{let}\ f = \lambda x \rightarrow \lambda y \rightarrow (:)\ y\ x\ \mathbf{in}\ foldl\ f\ [\,]$$

Feasibility of using model solutions

- We only recognise variants of model solutions
- We cannot determine whether or not a solution is wrong (but see one of the labs accompanying these lectures)
- In an experiment with lab exercises from first-year students:
  - our tool recognised 90% of the good solutions
  - using 5 model solutions.

Automatically deriving programming strategies

We automatically derive a strategy from a model solution:

- teachers can use Haskell
- much easier than specifying a strategy by hand
- combine solutions using $<|>$

We go from a model solution to a programming strategy by

- Pattern matching on the abstract syntax tree
- Mapping each (possibly combination of) language construct to its corresponding refinement rule
- Using prelude strategies and the interleave combinator $<\%>$ to add flexibility

[CEFP: A Programming Tutor for Haskell]

Calculating feedback

How do we calculate feedback?

- A strategy is specified as a context-free grammar over refinement (or rewrite) rules
- Most feedback is calculated from the grammar functions *empty* and *firsts*
- To verify that a submitted program follows a strategy we:
  - apply all allowed rules to the previous program
  - normalise the programs thus obtained
  - and compare these against the normalised program submitted by the student

Relating strategies to locations in programs

- A program is constructed incrementally
- At the start there is a single hole
- Refinement rules introduce and refine holes
- A refinement rule always targets a particular location in the program:

  > $foldl\ (flip\ {<}?{>})\ {<}?{>} \Rightarrow foldl\ (flip\ {<}?{>})\ some\_argument$

- Every refinement rule is extended with information about the location of the hole it refines

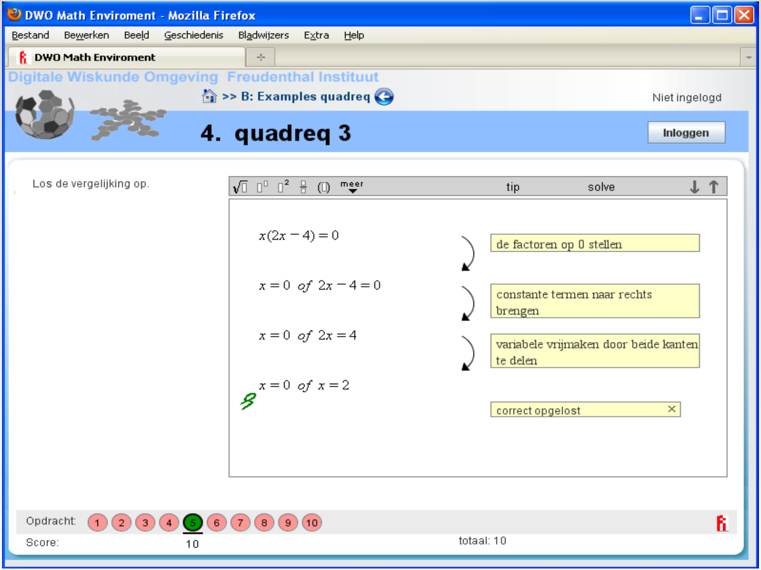
[CEFP: A Programming Tutor for Haskell]

Wrap up

Background

- We have developed strategies and our strategy language since 2006, and used it in
  - algebra: solving all kinds of (in)equations, simplifying expressions
  - linear algebra
  - propositional logic
- Our feedback services are used by
  - The Freudenthal applets for high-school mathematics, used by tens of thousands of pupils
  - The MathDox mathematical learning environment for mathematics (university and high-school)
  - The European Math-Bridge service for remedial mathematics, used by thousands of starting university students all over Europe

DWO Math Environment (with feedback)



[CEFP: A Programming Tutor for Haskell]

Related work

- Strategies are used in program transformation tools and rewriting systems
- Strategies closely correspond to proof tactics used in Isabelle, Coq, etc.
- Strategies have not been used for recognition/parsing/feedback purposes before

- Existing programming tutors often start with reasoning on an abstract level, pushing a student into a particular direction
- In most tutors, developing an exercise is quite a lot of work
- Tutors do not use strategies to give feedback

[CEFP: **A Programming Tutor for Haskell**]

Rest of the lectures

We have 4 slots to study the tutor for Haskell, its background, and to work on exercises or a research project:

- Slot 1: Introduction, overview, tutors, strategies
- Slot 2: A strategy language
- Slot 3: A strategy recogniser
- Slot 4: Brief overview of the ideas framework. Introduction to the exercises/project work

Learning goals

- Construct a strategy for a particular kind of exercises
- Analyse and describe properties of a strategy
- Adapt our framework:
  - the strategy language
  - the strategy recogniser
  - the feedback

Exercises, projects, slides, and notes

We have made all our material available on

`http://people.cs.uu.nl/johanj/homepage/Publications/CEFP/`

- Exercises: `exercises.pdf`
- Slides:
  - `slides1.pdf`: Introduction, overview, tutors, strategies
  - `slides2.pdf`: The strategy language
  - `slides3.pdf`: A strategy recogniser
  - `slides4.pdf`: Brief overview of the ideas framework. Introduction to the exercises/project work
- Lecture notes: `notes.pdf`

Software

Experiment on-line:

`http://ideas.cs.uu.nl/ProgTutor/`

Build the tutor on your own machine:

`http://ideas.cs.uu.nl/trac/wiki/Download`

Project 1: Adapting feedback

A teacher should be able to add feedback to a model solution.

> $reverse = foldl$ {-# FEEDBACK Note ... #-} $(flip\ (:))\ []$

and it should be possible to disallow or enforce particular solutions described by a strategy:

> $reverse =$ {-# USEDEF #-} $foldl\ (flip\ (:))\ []$

Furthermore, we might want to add a property to a function, and use that in a strategy:

> $reverse =$
>   {-# PROP foldl op e == foldr (flip op) e . reverse #-}
> $foldl\ (flip\ (:))\ []$

Implement these ideas for adapting strategies.

[CEFP: **A Programming Tutor for Haskell**]

Project 2: Automatic contract checking

We want the student's definition $\mathit{reverse} = {<}?{>}$ to satisfy the function contract:

$$(x : \mathit{true}) \rightarrow \{y \mid y \equiv \mathit{reverse}\ x\}$$

for some model solution of *reverse*. If a student refines with ${<}?{>} \Rightarrow \mathit{foldl}\ {<}?_1{>}\ {<}?_2{>}$, this holds if both

$$\mathit{assert}\ ((x : \mathit{true}) \rightarrow (y : \mathit{true}) \rightarrow \{z \mid z \equiv \mathit{flip}\ (:)\ x\ y\}\ {<}?_1{>}$$
$$\mathit{assert}\ ({\equiv}\ [\,])\ {<}?_2{>}$$

Strategies (and normalisation) help in constructing such refinement (proof) steps.

Investigate if we can use contracts for blaming incorrect steps.

[CEFP: A Programming Tutor for Haskell]

Expectation management

- The current release of the tutor has been developed over the last few months, and been released yesterday night
- The tutor still has to be tested in the classroom
- It will contain some glitches here and there
- Please report!

Conclusions

Strategies can be used to calculate feedback for introductory programming tasks.

More info:

- `http://ideas.cs.uu.nl/`
- johanj@cs.uu.nl
- alex.gerdes@ou.nl