

A Programming Tutor for Haskell

Alex Gerdes

Joint work with Johan Jeuring and Bastiaan Heeren

Computer Science

Open Universiteit Nederland and Utrecht University

Chalmers, Göteborg

October 2011

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?



Example

An error message when using a Haskell compiler:

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

<interactive>:0:27:
  Couldn't match expected type 'IO b0'
    with actual type 'Char -> IO ()'
  In the second argument of '(>>)', namely 'putChar'
  In the expression: putChar 'a' >> putChar
  In an equation for 'main': main = putChar 'a' >> putChar
```

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



Challenges for programming tutors

- ▶ Programming tutors focus on learning through feedback
- ▶ Instantly provide: hints, worked-out solutions, and more

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



A programming tutor for Haskell

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

- ▶ develop her program incrementally
- ▶ 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.





Functional Programming Tutor

[Strategy hint](#)[Step hint](#)[Step](#)[All steps](#)[Solution](#)[Finished?](#)[Back](#)[Exercise list](#)

Exercise:

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

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

```
Data.List> myreverse [1,2,3,4]  
[4,3,2,1]
```

Feedback:

```
1 ...
```

[Check](#)

An example interactive session

Programming task: write a program that reverses a list:

| `reverse = ⊥`

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:

```
reverse = reverse' ⊥  
  where  
    reverse' acc ⊥ = ⊥
```

Tutor response on Hint:

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



An example interactive session

Programming task: write a program that reverses a list:

```
reverse = reverse' []
```

where

```
reverse' acc [] = acc
```

```
reverse' acc (x : xs) = reverse' (y : acc) ⊥
```

Tutor response on Check:

```
Error: Undefined variable y
```



Strategies for programming exercises



Strategies for programming exercises

- ▶ 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
- ▶ The strategy language is implemented as a DSEL



Model solutions for *reverse*

In order to use our tutor, we need to define model solutions.

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

$reverse_1 [] = []$
 $reverse_1 (x : xs) = reverse_1 xs ++ [x]$

$reverse_2 = reverse'_2 []$
where $reverse'_2 acc [] = acc$
 $reverse'_2 acc (x : xs) = reverse'_2 (x : acc) xs$

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



Strategy example

The third program for *reverse*:

```
| reverse3 = foldl (flip (:)) []
```

is recognised by the strategy:

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



Representing strategies

Components of our strategy language:

1. Rewrite and refinement rules
2. Choice $\sigma \langle \diamond \rangle \tau$
3. Sequence $\sigma \langle \star \rangle \tau$
4. Interleave $\sigma \langle \% \rangle \tau$
5. Left-interleave $\sigma \% \rangle \tau$
6. Atomic $\langle \sigma \rangle$
7. Unit elements *succeed, fail*
8. Labels *label $\ell \sigma$*
9. 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.
- ▶ No left-recursion



The language of a strategy

The language of a strategy is defined by:

$$\begin{aligned}\mathcal{L}(r) &= \{r\} \\ \mathcal{L}(\sigma \triangleleft \tau) &= \mathcal{L}(\sigma) \cup \mathcal{L}(\tau) \\ \mathcal{L}(\text{fail}) &= \emptyset \\ \mathcal{L}(\sigma \triangleleft^* \tau) &= \{xy \mid x \in \mathcal{L}(\sigma), y \in \mathcal{L}(\tau)\} \\ \mathcal{L}(\text{succeed}) &= \{\epsilon\} \\ \mathcal{L}(\text{label } \ell \sigma) &= \{\text{Enter}_\ell x \text{Exit}_\ell \mid x \in \mathcal{L}(\sigma)\} \\ \mathcal{L}(\text{fix } f) &= \mathcal{L}(f(\text{fix } f)) \\ \mathcal{L}(\langle \sigma \rangle) &= \{\langle x \rangle \mid x \in \mathcal{L}(\sigma)\} \\ \mathcal{L}(\sigma_1 \langle \% \rangle \sigma_2) &= \mathcal{L}(\sigma_1) \langle \% \rangle \mathcal{L}(\sigma_2) \\ \mathcal{L}(\sigma_1 \% \rangle \sigma_2) &= \mathcal{L}(\sigma_1) \% \rangle \mathcal{L}(\sigma_2)\end{aligned}$$



Refinement rules

A refinement rule refines a **hole**.

Expression refinement rules:

$\perp \Rightarrow \lambda \perp \rightarrow \perp$ -- Introduce lambda abstraction

$\perp \Rightarrow$ **if** \perp -- Introduce **if-then-else**
then \perp
else \perp

$\perp \Rightarrow v$ -- Introduce variable v

Declaration refinement rule:

$\perp \Rightarrow f \perp = \perp$ -- Introduce a function binding



Holes

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

The **abstract syntax** is augmented with hole constructors.

```
data Expr = Lambda Pattern Expr
          | If Expr Expr Expr
          | App Expr Expr
          | Var String
          | Hole
          | ...
```



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:

$$\mathbf{|} \textit{foldl} (\textit{flip} \perp) \perp \Rightarrow \textit{foldl} (\textit{flip} \perp) \textit{some_argument}$$

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



Recognizing *flip*

For Haskell's prelude function *flip*:

$$| \textit{flip} = \lambda f \ x \ y \rightarrow f \ y \ x$$

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

$$| \begin{array}{l} \textit{flip} \ f \\ \lambda x \ y \rightarrow f \ y \ x \end{array}$$

which explains the implementation of *flipS*:

$$| \begin{array}{l} \textit{flipS} \ fS = \textit{app} \ \langle \star \rangle \ \textit{var} \ \textit{"flip"} \ \langle \star \rangle \ fS \\ \quad \langle \diamond \rangle \ \textit{lambda} \ \langle \star \rangle \ (\textit{pVar} \ x \ \langle \% \rangle \ \textit{pVar} \ y) \\ \quad \quad \quad \langle \star \rangle \ \textit{app} \ \langle \star \rangle \ fS \ \langle \star \rangle \ (\textit{var} \ y \ \langle \% \rangle \ \textit{var} \ x) \end{array}$$

where *x* and *y* do not appear free in *fS*.



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*:

$$\mathbf{|} \text{ foldl } op \ e \equiv \text{ foldr } (\text{flip } op) \ e \circ \text{ reverse}$$



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 "[]")
```



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 \triangleleft (left-factors!)

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 $\triangleleft\% \triangleright$ to add flexibility



Normalisation



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

| $reverse = foldl f []$ **where** $f x y = y : x$

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

| $reverse =$ **let** $f x y = y : x$ **in** $foldl f []$

\Rightarrow { Infix operators to (prefix) functions }

| $reverse =$ **let** $f x y = (:) y x$ **in** $foldl f []$

\Rightarrow { Function bindings to lambda abstractions }

| $reverse =$ **let** $f = \lambda x \rightarrow y \rightarrow (:) y x$ **in** $foldl f []$



Feasibility of using model solutions

- ▶ We only recognise variants of model solutions
- ▶ In an experiment with lab exercises from first-year students:
 - ▶ our tool recognised 90% of the good solutions
 - ▶ using 5 model solutions.
- ▶ We cannot determine whether or not a solution is wrong



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



Recognising multiple steps



Classroom experiments

- ▶ AFP summer school: approx. 35 students
- ▶ UU first year FP-course: > 200 students
- ▶ Preliminary conclusions:
 - ▶ Students take larger steps than anticipated
 - ▶ The given feedback is confusing when students deviate from suggested path
 - ▶ Some glitches here and there



Larger step size

- ▶ Recognising multiple steps
- ▶ Simple: repeatedly call *firsts*
- ▶ Started out with breadth-first search
- ▶ However, bfs does not reflect the way students work. Consider, with

$\sigma = (\sigma_1 \langle \star \rangle \sigma_2 \langle \star \rangle \sigma_3) \langle \% \rangle (\sigma_4 \langle \star \rangle \sigma_5 \langle \star \rangle \sigma_6)$:

$$\begin{array}{|l} \text{map}' \perp_1 \perp_2 = \perp_3 \\ \text{map}' f (\perp_4 : \perp_5) = \perp_6 \end{array}$$

- ▶ Switched to depth-first search
- ▶ Still not good enough: interleaving introduces a **huge** number of possibilities



Number of interleavings

The number of interleavings for two sentences of lengths n and m equals $\frac{(n+m)!}{n!m!}$.

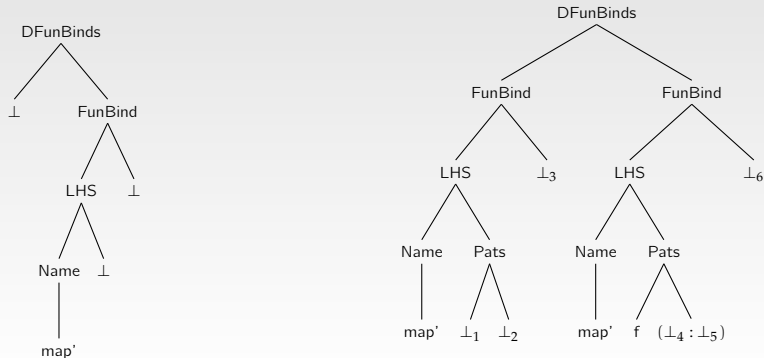
3	4	35
4	5	126
4	6	210
10	11	352716

- ▶ Very often all substrategies are interleaved
- ▶ Number of interleavings increases to $n!$



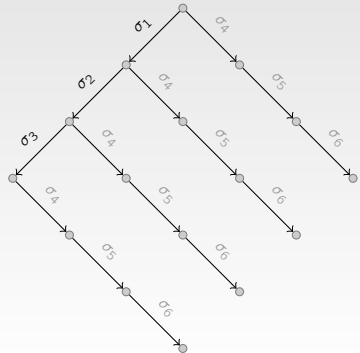
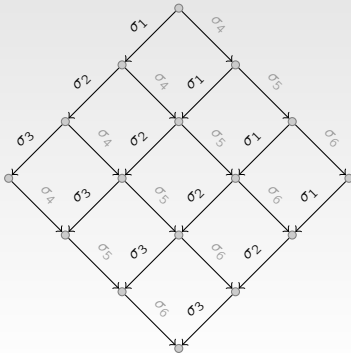
Pruning

- ▶ The search space is too large
- ▶ We **prune** the search space
- ▶ Some intermediate answers will never lead to success
- ▶ The *firsts* set is filtered
- ▶ We check if an element in *firsts* is a **predecessor** of the student submission



We can do better!

- ▶ Not a very efficient approach: first generate then filter
- ▶ Generating many duplicates
- ▶ The order of refinements is irrelevant for multiple steps!
- ▶ Introduce a Search Mode
- ▶ Recall $\sigma = (\sigma_1 \langle \star \rangle \sigma_2 \langle \star \rangle \sigma_3) \langle \% \rangle (\sigma_4 \langle \star \rangle \sigma_5 \langle \star \rangle \sigma_6)$



Alternative interleave combinator

The definition of the interleave combinator:

$$\begin{aligned} \sigma_1 \langle \% \rangle \sigma_2 &= \sigma_1 \% \sigma_2 \diamond \sigma_2 \% \sigma_1 \\ (\langle a \rangle \langle \star \rangle \sigma_3) \% \sigma_2 &= \langle a \rangle \langle \star \rangle (\sigma_3 \langle \% \rangle \sigma_2) \end{aligned}$$

In search mode:

$$\begin{aligned} \sigma_1 \langle \% \rangle^* \sigma_2 &= \sigma_1 \%^* \sigma_2 \diamond \sigma_2 \\ (\langle a \rangle \langle \star \rangle \sigma_3) \%^* \sigma_2 &= \langle a \rangle \langle \star \rangle (\sigma_3 \langle \% \rangle^* \sigma_2) \end{aligned}$$

Property:

$$\mathcal{L}(\sigma_1 \langle \% \rangle^* \sigma_2) \subseteq \mathcal{L}(\sigma_1 \langle \% \rangle \sigma_2)$$



Future work: Automatic contract checking

We want the student's definition $reverse = \perp$ to satisfy the function contract:

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

for some model solution of $reverse$. If a student refines with $\perp \Rightarrow foldl\ \perp_1\ \perp_2$, this holds if both

$$\mid \begin{array}{l} assert\ ((x : true) \rightarrow (y : true) \rightarrow \{z \mid z \equiv flip\ (\cdot)\ x\ y\})\ \perp_1 \\ assert\ (\equiv [])\ \perp_2 \end{array}$$



Why not only do testing?

Consider a function that converts a list of binary numbers to its decimal representation:

```
fromBin [1, 0, 1, 0, 1, 0]  
⇒ 42
```

The following definition that implements this function:

```
fromBin :: [Int] → Int  
fromBin = fromBin' 2
```

```
fromBin' n [] = 0  
fromBin' n (x : xs) = x * n ^ (length (x : xs) - 1)  
                    + fromBin' n xs
```



Why not only do testing?

It does meet the specifications (contract). However, it contains a number of (severe) imperfections:

- ▶ The length calculation is inefficient
- ▶ It takes time quadratic in the size of the input list
- ▶ Argument n is constant and should be abstracted

We found these imperfections frequently in a set of student solutions.

$$\begin{aligned} \text{fromBin} &:: [\text{Int}] \rightarrow \text{Int} \\ \text{fromBin} &= \text{fromBin}' 2 \end{aligned}$$
$$\begin{aligned} \text{fromBin}' n [] &= 0 \\ \text{fromBin}' n (x:xs) &= x * n ^ (\text{length} (x:xs) - 1) \\ &\quad + \text{fromBin}' n xs \end{aligned}$$


Future work: 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 (:)) []
```



Conclusions

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

More info:

- ▶ alex.gerdes@ou.nl
- ▶ General information:
<http://ideas.cs.uu.nl/>
- ▶ Experiment on-line:
<http://ideas.cs.uu.nl/ProgTutor/>
- ▶ Sources:
<http://ideas.cs.uu.nl/trac/wiki/Download>

