Using Strategies for Assessment of Functional Programming Exercises

Alex Gerdes

Joint work with Johan Jeuring and Bastiaan Heeren

Open Universiteit Nederland
School of Computer Science

8 January 2010

Assessment of programming exercises

- Every year, thousands of computer science students learn to program
- It is important to assess the students abilities and to provide timely feedback
- Traditionally, a teacher assesses programming exercises
- Assessing is tedious, time consuming, and error prone work
- Many assessment tools have been developed to assist teachers
- Most tools are based on testing

Disadvantages of test-based assessment

Test-based assessment tools try to determine correctness by comparing the output of a student program to the expected results. Test-based assessment has a number of disadvantages:

1. Coverage: how do you know you have tested enough?
2. Testing is a dynamic process and therefore vulnerable to bugs
3. Inability to assess design features, such as good programming practices
4. Testing cannot reveal which algorithm has been used

Example

A small exercise, typical for learning how to program in Haskell, is to write a function that converts a list of binary numbers to its decimal representation:

$$\mathit{fromBin}\ [1,0,1,0,1,0]$$
$$\Rightarrow 42$$

The following definition that implements this function:

$$\mathit{fromBin} :: [\mathit{Int}] \rightarrow \mathit{Int}$$
$$\mathit{fromBin} = \mathit{fromBin}'\ 2$$

$$\mathit{fromBin}'\ n\ [\,] = 0$$
$$\mathit{fromBin}'\ n\ (x : xs) = x * n\ \hat{}\ (\mathit{length}\ (x : xs) - 1) + \mathit{fromBin}'\ n\ xs$$

Example

Test-based assessment tools will most likely accept the solution. However, it contains a number of 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.

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

Our solution (1/2)

We propose to use strategies in combination with program transformations based on the λ-calculus, to assess programming exercises

- A programming strategy is derived from a set of model solutions
- We generate a set of equivalent solutions based on a programming strategy
- Strategies do not generate all equivalent solutions
- We increase the number of accepted correct solutions by normalisation
- After normalisation, we compare solutions syntactically

Our solution (2/2)

We assess the following features:

- Correctness
- Design

Our approach has the following advantages:

- If a program is determined to be equivalent, it is guaranteed to be correct
- We can recognise and report imperfections
- We can determine which algorithm has been implemented
- Strategy-based assessment is carried out statically.

A disadvantage of our approach is that we cannot prove a student solution to be incorrect.

Example assessment

- We applied our tool to student solutions from a lab assignment in a first-year FP-course at Utrecht University
- In total we received 94 student solutions
- We were not involved in any aspect of the assignment

The students had to implement the *fromBin* function.

Model solutions (1/2)

There are a number of model solutions, which differ quite a bit from one another. All of them use recommended programming techniques:

$$fromBin = foldl\ ((+) \circ (2*))\ 0$$

$$\begin{aligned}
&fromBin\ xs = fromBin'\ (length\ xs - 1)\ xs \\
&\quad \textbf{where} \\
&\qquad fromBin'\ _\ []\qquad\quad = 0 \\
&\qquad fromBin'\ l\ \ (x:xs) = x * 2 \mathbin{\char`^} l + fromBin'\ (l-1)\ xs
\end{aligned}$$

$$fromBin = sum \circ zipWith\ (*)\ (iterate\ (*2)\ 1) \circ reverse$$

Model solutions (2/2)

The last model solution we consider is simple, but inefficient:

$$\begin{aligned} &\mathit{fromBin}\ [\,] &&= 0 \\ &\mathit{fromBin}\ (x : xs) &&= x * 2 \,^\wedge\, \mathit{length}\ xs + \mathit{fromBin}\ xs \end{aligned}$$

The length of the list is calculated in each recursive call. A teacher can:

- Accept or reject this solution
- Turn the model solution into a buggy strategy and report to the student why their solution is rejected

Example

We can recognise many different equivalent solutions from a model solution. For example, the following student solution:

$$
\begin{aligned}
&\mathit{fromBin} = \mathit{fromBaseN}\ 2\\
&\mathit{fromBaseN}\ b\ n = \mathit{fromBaseN}'\ b\ (\mathit{reverse}\ n)\\
&\quad \textbf{where}\\
&\qquad \mathit{fromBaseN}'\ \_\ [\,] \qquad\quad = 0\\
&\qquad \mathit{fromBaseN}'\ b'\ (c : cs) = c + b' * (\mathit{fromBaseN}'\ b'\ cs)
\end{aligned}
$$

is recognised from this model solution:

$$\mathit{fromBin} = \mathit{foldl}\ ((+) \circ (2*))\ 0$$

Categories

We have partitioned the set of student programs into four categories by hand:

- **Good.** A proper solution with respect to the features
- **Modified.** Some students have augmented their solution with sanity checks. We have removed the checks by hand
- **Imperfect.** An imperfect program is a program that is rejected because we want to report the imperfection
- **Incorrect.** A few student programs were incorrect

Results

- 72 programs fall into the good and modified (9) categories; our assessment tool recognises 64 programs (89%)
- The acceptance rate can be increased by adding more model solutions
- All of the incorrect and imperfect programs were rejected
- Some programs that were rejected with reason had gotten full grades from the assistant

We can tell which model solution a student has used:

- 18 students used the *foldl* model solution
- 2 used tupling
- 4 the inner product solution
- 40 solutions were based on explicit recursion

Details of our approach

Strategies

- A strategy is a well-defined plan for solving a particular problem
- A programming strategy is implemented as a context-free grammar with refinement rules as symbols
- We have developed a library with an embedded domain-specific language for specifying strategies
- Strategies can also be used to detect common mistakes. These are called buggy strategies
- Programming strategies can be automatically derived from model solutions

Standard strategies

- We have defined a set of standard programming strategies
- Standard strategies generate many syntactically different solutions from a single model solution
- The automatically derived programming strategies are defined in terms of these standard strategies.

For example, using the strategy for function composition:

$$f \circ g = \lambda x \rightarrow f\ (g\ x)$$

We can recognise both composition itself, and its definition:

$$fromBin = foldl\ ((+) \circ (2*))\ 0$$
$$fromBin = foldl\ (\lambda x\ y \rightarrow 2 * x + y)\ 0$$

Program transformations

- Strategies from model solutions are rather strict and may 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 program transformations, which are based on the λ-calculus, to ignore such differences
- We use η- and β-reduction, and α-conversion
- Additionally, we perform preprocessing rewrite steps such as inlining
- In general, comparing two lambda terms for equality is undecidable. However, we did not encounter any problems

Normalisation

Normalisation is performed using the following rewrite steps:

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

Normalisation example

Recall the student program we have introduced before:

$$
\begin{aligned}
&\mathit{fromBin} = \mathit{fromBaseN}\ 2\\
&\mathit{fromBaseN}\ b\ n = \mathit{fromBaseN}'\ b\ (\mathit{reverse}\ n)\\
&\quad \textbf{where}\\
&\qquad \mathit{fromBaseN}'\ \_\ [\,] \qquad\qquad = 0\\
&\qquad \mathit{fromBaseN}'\ b'\ (c : cs) = c + b' * (\mathit{fromBaseN}'\ b'\ cs)
\end{aligned}
$$

After applying all transformations the student program looks as follows:

$$
\begin{aligned}
&\mathit{fromBin} = \lambda x_2 \rightarrow\\
&\quad \textbf{let}\ x_3\ [\,] \qquad\quad\ = 0\\
&\qquad\ \ x_3\ (x_4 : x_5) = (+)\ ((*)\ 2\ (x_3\ x_5))\ x_4\\
&\quad \textbf{in}\ \ x_3\ (\mathit{reverse}\ x_2)
\end{aligned}
$$

Future work

- Use programming strategies to generate semantically rich feedback. However, program transformations complicate this generation. We want to investigate how we can alleviate this problem
- Investigate how well our approach works for developing programs in programming languages like Java or C++
- Investigate how we can extend our approach with testing, property checking, or static contract checking

Epilogue

- Strategies can be successfully used for programming exercise assessment
- We can guarantee a student solution to be equivalent to a model solution
- We are able to recognise many different student programs from a limited set of model solutions
- Using only 4 model solutions we managed to recognise and characterise 89% of the correct solutions

- Information about our research: `http://ideas.cs.uu.nl`
- E-mail: alex.gerdes@ou.nl