Using Strategies for Assessment of Programming Exercises

Alex Gerdes

Johan Jeuring

Bastiaan Heeren

Technical Report UU-CS-2009-031
December 2009

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Using Strategies for Assessment of Programming Exercises

Alex Gerdes
School of Computer Science,
Open Universiteit Nederland
alex.gerdes@ou.nl

Johan T. Jeuring
Department of Information and
Computing Sciences,
Utrecht University
johanj@cs.uu.nl

Bastiaan J. Heeren
School of Computer Science,
Open Universiteit Nederland
bastiaan.heeren@ou.nl

ABSTRACT

Programming exercise assessment tools alleviate the task of teachers, and increase consistency of markings. Many programming exercise assessment tools are based on testing. A test-based assessment tool for programming exercises cannot ensure that a solution is correct. Moreover, it is difficult to test if a student has used good programming practices. This is unfortunate, because teachers want students to adopt good programming techniques. We propose to use strategies, in combination with program transformations, as a foundation for functional programming exercise assessment. Expert knowledge, in the form of model solutions, can be expressed as programming strategies. Using these strategies we can guarantee that a student program is equivalent to a model solution, and we can report which solution strategy has been used to solve the programming problem.

Categories and Subject Descriptors

K.3.1 [**Computer Uses in Education**]: Computer-assisted instruction (CAI); K.3.2 [**Computer and Information Science Education**]: Computer science education

General Terms

Languages, Human Factors, Measurement

Keywords

Automatic assessment, functional programming, Haskell, strategies

1. INTRODUCTION

Every computer science curriculum offers courses in which beginners learn to program, and every year, thousands of students take such courses. To support learning programming, it is important to assess the students' progress and provide timely feedback. Traditionally, a teacher or an assistant assesses a student's abilities and progress. However, in the case of large class sizes, providing timely feedback is not always possible. Furthermore, repeatedly assessing student exercises is tedious, time consuming, and error prone. It is difficult to keep judgements consistent and fair. To assist teachers in assessing programming assignments, many assessment tools have been developed.

Most automated assessment tools for programming languages are based on some form of testing [1]. Test-based assessment tools try to determine correctness by comparing the output of a student program to the expected results on test data. Using testing for assessment has a number of problems. First, an inherent problem of testing is coverage: how do you know you have tested enough? Testing does not ensure that the student program is correct. Second, assessing design features, such as the use of good programming techniques or the absence of imperfections, is hard if not impossible with testing. Consider the following function that solves the problem of converting a list of binary numbers to its decimal representation:

```
fromBin :: [Int] → Int
fromBin = fromBin′ 2
fromBin′ n [ ]      = 0
fromBin′ n (x : xs) = x ∗ n^(length (x : xs) − 1)
                      + fromBin′ n xs
```

This function returns correct results, hence test-based assessment tools will most likely accept this as a good solution. However, this implementation contains at least one imperfection: the length calculation is inefficient (an element is added to the list and then the length of the list is subtracted by one). We found this imperfection frequently in a set of student solutions. It would certainly help if we had means to report and explain such inefficient or imperfect solutions. Third, testing cannot reveal which algorithm has been used. For instance, when asked to implement quicksort, it is difficult to discriminate between bubblesort and quicksort. Fourth, testing is a dynamic process and is therefore vulnerable to bugs, and even malicious features, that may be present in solutions.

strategy language [3]. We use this language to capture expert knowledge about how to solve an exercise. The framework can handle exercises from multiple domains, such as proposition logic, linear algebra, etc. Recently, we have extended our framework with the ability to reason about programming exercises [2]. We use this framework, in combination with program transformations, based on the lambda calculus, to assess functional programming exercises. Our approach is rather different from testing: we can *guaran-*

SIGCSE'10, March 10–13, 2010, Milwaukee, Wisconsin, USA.

tee that the submitted student program is equivalent to a model program. We can recognise many different equivalent solutions from a model solution. For example, the following student solution:

```
fromBin = fromBaseN 2
fromBaseN b n = fromBaseN′ b (reverse n)
  where
    fromBaseN′ _ [ ]        = 0
    fromBaseN′ b′ (c : cs) = c + b′ ∗ (fromBaseN′ b′ cs)
```

is recognised from this model solution:

```
fromBin = foldl ((+) ∘ (2∗)) 0
```

Despite the fact that this solution appears very different from the model solution, it will be recognised as equivalent. The two solutions are essentially the same. The *foldl* function can be defined as a *foldr*:

```
foldl op b = foldr (flip op) b ∘ reverse
```

The first solution makes use of this equivalence. It uses, however, the the explicit recursive definition of *foldr*.

In this paper we show how programming strategies and program transformations can be used to assess functional programming exercises. Using strategies for assessing student programs solves the four problems of using testing for assessment described above: 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; and strategy-based assessment is carried out *statically*. In contrast with our approach, test-based assessment tools can give a judgement of all programs including incorrect ones. Test-based assessment tools can prove a program to be incorrect by providing a counter-example. However, finding a counter-example also suffers from the coverage problem.

We use the functional programming language Haskell as our object language. We have integrated the Haskell compiler Helium [4] into our assessment tool. We believe, however, that our approach is equally well applicable to other programming languages and programming paradigms.

This paper has the following contributions:

- We automatically assess student programs based on a set of model solutions using programming strategies, and program transformations (sections 2 and 3).
- We show the results of applying our assessment tool to 94 student solutions of a typical functional programming exercise (Section 4).

2. STRATEGY BASED ASSESSMENT

The most important features we want to assess in a student program are:

- Correctness: does the program implement the requirements?
- Design: has the program been implemented following good programming practices?

A model solution is a preferred way to solve a problem. Model solutions use good programming techniques, and we want our students to use these techniques. Model solutions can be transformed to model strategies automatically. These programming strategies are defined in the strategy language mentioned above.

We propose to use programming strategies as a foundation for assessment of programming exercises. We generate a set of solutions equivalent to the model solution(s) described by a programming strategy. A student solution is correct if it is an element of the generated set. Usually, however, strategies do not generate all solutions that are in essence equivalent to the model solution described. For example, a student should be free in choosing the identifiers in a program, and for this we incorporate α-renaming in our framework. To increase the number of accepted correct student programs, we *normalise* the generated set of model solutions and the student program using meaning preserving program transformations from the lambda calculus, such as β- and η-reduction. After normalising we syntactically check if the program is an element of the set of normalised model solutions.

A strategy is a well-defined plan or a sequence of steps for solving a particular problem. It represents expert knowledge for problem solving. Strategies are also known as procedural skills. We have implemented a programming strategy as a *context free grammar* with refinement rules as symbols. A refinement rule refines an undefined part (a hole) of an incomplete program. Examples of refinement rules are: introducing a variable, and distinguishing the empty list case from the non-empty list case. A programming strategy describes a sequence of refinement steps necessary for constructing a program.

We have developed a library with an embedded domain-specific language for specifying strategies for exercises. The language can be used for many domains, provided that the domains are based upon rewrite rules or refinement rules. The strategy language consists of a set of strategy combinators. These combinators are very similar to parser combinators [6] and are used to define strategies that consist of several substrategies. Examples of strategy combinators are: a sequence combinator that applies its argument strategies one after another, and a choice combinator that offers the possibility to choose between two strategies.

Strategies can also be used to detect common mistakes. These are called *buggy strategies*. For example, a common mistake we detected in a set of student solutions is the inefficient way of calculating the length of a list described in the introduction of this paper. Using buggy strategies we can report to the student that a common mistake was detected, and provide detailed feedback by explaining what is wrong, and how this should be corrected.

Programming strategies can be automatically derived from model solutions. To use our assessment tool, a teacher merely needs to specify one or more model solution. A teacher does not have to learn a formalism for specifying strategies.

2.1 Standard strategies

Using strategy combinators we have defined a set of standard programming strategies. These standard strategies are strategies for the functions in the standard library, and for the basic language constructs of Haskell. The standard library (the Haskell Prelude) defines common basic functions, such as addition and concatenation. Examples of Haskell language constructs are lambda expressions and *where* clauses. The 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 composition expressed in terms of the lambda expression from its definition. This is used, amongst others, to check that the following two programs are equivalent:

```
fromBin = foldl ((+) ∘ (2∗)) 0
fromBin = foldl (λx y → 2 ∗ x + y) 0
```

3. PROGRAM TRANSFORMATIONS

One of the advantages of assessing programming exercises using strategies is that an accepted program is guaranteed to be equivalent to a model solution. However, most basic strategies are rather strict, and might reject programs that are equivalent but have some differences. Not all of these differences can or should be captured in a strategy, because they are standard transformations of a program, independent of a particular strategy. For example, inlining a helper-function should be considered correct in any strategy. We use program transformations to transform a program to a normal form[1]. We are not so much interested in the exact normal form, as long as it can be used to efficiently compare two terms for equality.

Our program transformations are based on the *lambda calculus*. The lambda calculus is at the core of our functional programming language, and its reduction rules form the heart of the evaluation machinery. In particular, we use η- and β-reduction, and α-conversion. In general, comparing two lambda terms for equality is undecidable. However, we can decide equivalence for many terms using program transformations. Our equivalence checker may reject equivalent programs and hence we may have false-negatives. Until now we have not found this to be a problem in practice. If programs are found to be equivalent, they are semantically equivalent, so we do not obtain false-positives.

3.1 Preprocessing

In addition to the reduction rules of the lambda calculus, which we will discuss in the next subsection, we apply a number of preprocessing transformation steps. These steps are either for removing syntactic sugar or superficial syntax, or to trigger other transformations steps. We give a brief description of the most interesting transformation steps.

Constant arguments. Wherever possible we try to detect constant arguments. A couple of student solutions of the *fromBin* problem introduced in Section 1 use a constant argument:

```
fromBin = fromBaseN 2
fromBaseN b n = fromBaseN' b (reverse n)
  where
    fromBaseN' _  []      = 0
    fromBaseN' b' (c : cs) = c + b' ∗ (fromBaseN' b' cs)
```

Here the argument b' is constant. A technique often used in compilers is to optimise such constant arguments away:

```
fromBin = fromBaseN 2
fromBaseN b n = fromBaseN' b (reverse n)
  where
    fromBaseN' b' =
      let fromBaseN'' []       = 0
          fromBaseN'' (c : cs) = c + b' ∗ (fromBaseN'' cs)
      in  fromBaseN''
```

The reason to transform constant arguments away is to enable the inlining and β-reduction steps.

Inlining. Inlining replaces an expression by its definition. After inlining a lambda expression, we can often perform a β-reduction. Inlining is rather tricky: scope and binding play an important role. Before inlining a few other preprocessing steps have to be performed, such as rewriting *where* clauses to *let* expressions. The following definition shows the result of applying these preprocessing steps (including inlining) to the *fromBin* solution given above:

```
fromBin =
  ( let fromBaseN' b' =
          let fromBaseN'' []       = 0
              fromBaseN'' (c : cs) = c + b' ∗ (fromBaseN'' cs)
          in  fromBaseN''
    in λb n → fromBaseN' b (reverse n)
  ) 2
```

Prefix notation. Operators may be used in infix or prefix position both when they are defined and used. For example, both $f\ x\ y = x + y$ and $f\ x\ y = (+)\ x\ y$ may be used. We have chosen to use the prefix notation in our abstract syntax trees, and transform every infix application to the equivalent prefix application.

3.2 α-conversion, η,β-reduction

α-conversion renames bound variables. To check that a program is syntactically equivalent to a model solution, we need to α-convert both the submitted student program as well as the model solution. α-conversion ensures that all variable names are unique, and this simplifies other transformation steps, such as β-reduction. α-conversion is included in the preprocessing step.

Previous transformation steps have been illustrated with a student solution for *fromBin*, using the names from the student program. After α-conversion this program looks as follows:

```
fromBin =
  (let x5 x6 =
         let x5 []         = 0
             x5 (x7 : x8)  = (+) x7 ((∗) x6 (x5 x8))
         in  x5
   in  λx3 x4 → x5 x3 (reverse x4)
  ) 2
```

η-reduction replaces $\lambda x.fx$ by f if x does not appear free in f. We η-reduce the abstract syntax tree at all possible locations. Function bindings are η-reduced as well, so $f\ x = not\ x$ is replaced by $f = not$.

In lambda calculus, β-reduction is used when applying a function. β-reduction is defined using substitution: $(\lambda x \rightarrow expr)\ y \Rightarrow_{\beta} expr[x := y]$. The substitution $[x := y]$ replaces all free occurrences of the variable x by the expression y. We don't expect a student to write a program containing a β-redex, but in practice this happens. The main reason we need β-reduction is to inline helper-functions.

After applying all transformations, including β-reduction, the *fromBin* function looks as follows:

[1] We consider a program in normal form if it cannot be transformed by any of our rules anymore.

```
fromBin = λx2 →
  let x3 []        = 0
      x3 (x4 : x5) = (+) ((∗) 2 (x3 x5)) x4
  in  x3 (reverse x2)
```

4. USING OUR ASSESSMENT TOOL

We have applied our assessment tool to student solutions that were obtained from a lab assignment in a first-year functional programming course at Utrecht University (2008). We were not involved in any aspect of the assignment, and received the solutions after they had been graded ('by hand') by the teaching assistants. In total we received 94 student solutions.

The students had to implement the *fromBin* function introduced in Section 1. This function should convert a list of bits to a decimal number. For example, applying *fromBin* to $[1, 0, 1, 0, 1, 0]$ should return 42. This is a small exercise, typical for learning how to program in Haskell. The *fromBin* exercise can be solved in various ways, using different kinds of higher-order functions. There are a number of model solutions, which differ quite a bit from one another. All of them use recommended programming techniques.

The first of our model solutions uses a *foldl*, which is a higher-order function that processes a list from left to right and constructs a return value.

```
fromBin = foldl ((+) ∘ (2∗)) 0
```

Tupling is a well-known programming technique that groups results in a tuple, which is passed around in a recursive function. The second model solution uses this technique. The tuple is passed in the form of multiple function arguments.

```
fromBin xs = fromBin′ (length xs − 1) xs
  where
    fromBin′ _ []       = 0
    fromBin′ l (x : xs) = x ∗ 2^l + fromBin′ (l − 1) xs
```

The third solution reverses the input list, and then computes the inner product of this list and a list of powers of two.

```
fromBin = sum ∘ zipWith (∗) (iterate (∗2) 1) ∘ reverse
```

The above model solutions are both elegant and efficient. The fourth (and last) model solution we consider is simple, but inefficient:

```
fromBin []       = 0
fromBin (x : xs) = x ∗ 2^length xs + fromBin xs
```

Because the length of the list is calculated in each recursive call, this definition takes time quadratic in the size of the input list to calculate its result. The other model solutions are all linear. It is up to the teacher to decide to either accept or reject solutions based on this model. This flexibility is one of the advantages of our approach. It is also possible to turn this solution into a buggy strategy and report to the student why their solution is rejected (or accepted with reservations).

4.1 Categories

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

Good. A good program is a proper solution with respect to the features we assess (correctness and design). It should ideally be equivalent to one of the model solutions.

Good with modifications. Some students have augmented their solution with sanity checks. For example, they check that the input is a list of zeroes and ones. Since the exercise assumes the input has the correct form, we have not incorporated such checks in the model solutions. Furthermore, the transformation machinery is not yet capable of removing these 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. The solution to *fromBin* given in Section 1 is an example of an imperfect solution. Another common imperfection we found is the use of a superfluous case:

```
fromBin []       = 0
fromBin (x : []) = x
fromBin (x : xs) = (x ∗ 2^length xs) + fromBin xs
```

In this student example, the second case is unnecessary.

Incorrect. A few student programs were incorrect. They all contained the same error: no definition for the empty list.

4.2 Results

From the 94 student programs, 64 programs fall into the good category and 8 fall into the good with modifications category. From these, our assessment tool recognises 64 programs (89%). Another, and perhaps better, way of looking at these figures is that 64 student solutions are accepted based on just four model solutions. The acceptance rate can be increased by adding more model solutions. Using our tool a teacher merely needs to assess the remaining student solutions. These remaining student solutions can be either correct or incorrect, our tool can not tell.

All of the incorrect and imperfect programs were rejected in the test. Some of these incorrect programs were not noticed by the teaching assistants that corrected these programs.

It might happen that a student solution does not correspond to a model solution, although it is correct. In such a case a teacher might add the solution to the set of model solutions. It is likely that some student solutions do not qualify as a model solution, although they cannot be considered imperfect or wrong as well. For example, the following student solution uses the tupling technique:

```
fromBin []             = 0
fromBin [s]            = s
fromBin (s : t : rest) = fromBin ((2 ∗ s + t) : rest)
```

Instead of using a tuple or an extra argument, this solution 'misuses' the head of the list to store the result.

By checking all model solutions independently, we can tell which model solution, or strategy, a student has used to solve the exercise. Our test showed that 18 students used the *foldl* model solution, 2 used tupling, 2 the inner product solution, and 40 solutions were based on the last model solution with explicit recursion.

It is unlikely that a solution is accepted by more than one model solution. In our test all solutions were accepted by a single model solution. If model solutions are very similar, it is probably possible to adapt one of the standard strategies to recognise both from a single model solution.

A second assessment shows similar results as the one discussed in this paper.

5. RELATED AND FUTURE WORK

The survey of automated programming assessment by Ala-Mutka [1] shows that many assessment tools are based on dynamic testing. In contrast, our assessment tool statically checks for correctness. The survey provides many pointers to related work. We describe the three closest approaches.

The PASS system, developed by Thorburn and Rowe [7], assesses C programs by evaluating whether a student program conforms to a predefined solution plan. A drawback of the system is that it needs testing for this evaluation. Moreover, a solution plan is much more strict compared to a strategy. For example, the system considers the definition of any helper-function incorrect. Our approach allows a higher degree of freedom by means of standard strategies and program transformations.

The approach of Truong et al. [8] is also based on model solutions and abstract syntax tree inspections. However, their primary use is to assess software quality and not so much correctness. In addition to similarity checks, their system also calculates software metrics, which are used to give feedback to a student. A drawback of their approach is that it does not take the different syntactic forms of a model solution into account. Moreover, the similarity check considers only the outline of a solution and not its details.

Xu and Chee [10] show how to diagnose Smalltalk programs using program transformations. Our work is quite similar to theirs. For a functional programming language the set of transformations is much smaller and simpler. We would like to implement their advanced method for locating errors in student programs.

Program verification tools are used to prove programs correct with respect to some specification [5]. Automatic program verification tools provide as much support as possible in constructing this proof. However, users will always have to give hints or proof steps to complete proofs for non-trivial programs, such as *fromBin*. We expect it is easier both for teachers and students, to check student solutions against model solutions.

Future work. In addition to assessing programming exercises, we also want to use programming strategies to generate semantically rich feedback when a student develops a program step-wise. However, as soon as we use program transformations it is hard to relate the student program to a model solution: the program transformations changes the student program in an unrecognisable way. Therefore, we would like to investigate how we can retain the relation with the student steps as long as possible.

The approach we presented in this paper is focussed on functional programming languages, in particular Haskell. We believe that our approach is also applicable to other programming languages and to other programming paradigms. Our method is not tied to a particular programming language. The concepts on which our approach is based, such as strategies, refinement rules, and program transformations, are applicable to every programming language. We plan to investigate how well our approach works for developing programs in a programming language such as Java.

Our approach can guarantee that a program is correct. Test-based approaches can guarantee that a program is incorrect. We plan to investigate how we can extend our approach with testing, property checking, or static contract checking [9].

6. CONCLUSIONS

We have shown that strategies can be successfully used for programming exercise assessment. Our approach differs from test based assessment tools in that we can *guarantee* a student solution to be equivalent to a model solution. Strategies in combination with programming transformations are able to recognise many different student programs from a limited set of model solutions. In a test we performed on almost 100 student programs we managed to recognise and characterise 89% of the correct solutions, and we found several programs that had been incorrectly graded as correct by student assistants. Programming strategies can be specified in a convenient way and are very flexible.

7. ACKNOWLEDGEMENTS

The authors would like to thank Stefan Holdermans of Utrecht University for providing a set of suitable programming exercises, accompanied with a large amount of corrected student solutions.

8. REFERENCES

[1] K. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005.

[2] A. Gerdes, B. Heeren, and J. Jeuring. Constructing Strategies for Programming. In J. Cordeiro et al., editor, *Proceedings of the First International Conference on Computer Supported Education*, pages 65–72. INSTICC Press, March 2009.

[3] B. Heeren, J. Jeuring, A. v. Leeuwen, and A. Gerdes. Specifying strategies for exercises. In *MKM 2008: Mathematical Knowledge management*, volume 5144 of *LNAI*, pages 430–445. Springer-Verlag, 2008.

[4] B. Heeren, D. Leijen, and A. v. IJzendoorn. Helium, for learning Haskell. In *Haskell 2003*, pages 62 – 71. ACM, 2003.

[5] M. d. Mol, M. v. Eekelen, and R. Plasmeijer. Theorem proving for functional programmers - Sparkle: a functional theorem prover. In *IFL 2001, Selected Papers, volume 2312 of LNCS*, pages 55–72. Springer, 2002.

[6] S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In J. Launchbury et al., editor, *AFP*, volume 1129 of *LNCS*, pages 184–207. Springer-Verlag, 1996.

[7] G. Thorburn and G. Rowe. Pass: an automated system for program assessment. *Computers & Education*, 29(4):195–206, 1997.

[8] N. Truong, P. Roe, and P. Bancroft. Static analysis of students' java programs. In *ACE '04: Proceedings of the sixth conference on Australasian computing education*, pages 317–325, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[9] D. N. Xu. *Static Contract Checking for Haskell*. PhD thesis, Cambridge University, 2008.

[10] S. Xu and Y. S. Chee. Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Transansactions on Software Engineering*, 29(4):360–384, 2003.