A Canonicalizing Model for Building Programming Tutors

Kelly Rivers and Kenneth R. Koedinger

Carnegie Mellon University

Abstract. It is difficult to build intelligent tutoring systems in the domain of programming due to the complexity and variety of possible answers. To simplify this process, we have constructed a language-independent canonicalized model for programming solutions. This model allows for much greater overlap across different students than a basic text model, which enables more self-sustaining hint generation methods in programming tutors.

Keywords: canonicalization, programming tutors, abstract syntax trees.

1 Introduction

Though interest has continually been shown in creating intelligent tutors for programming topics, few solutions have been found that have been applied to widespread classes [1]. This is partially due to constraints already existing in the classroom such as programming language, development environment, and curriculum choices. We aim to simplify the tutor-building process by creating a language-independent method for turning students' programs into canonicalized models which can be more easily examined and compared than text programs. We also discuss ideas for self-sustaining hint generators that would not require as much instructor input.

2 Model Creation

Our model is based on **abstract syntax trees** (ASTs). ASTs represent the underlying structure of a program by branching complex statements out into smaller sub-statements. They are commonly used in program transformations, which means that modules already exist for creating and modifying ASTs from text for many different programming languages; they're also constructed from basic programming concepts, so they can be made equivalent across languages.

Once a student's program has been converted into an AST, we can gather relevant information on what data structures and algorithms the student is using by examining the tree. This information can later be used to unearth basic problems. For example, a student uncomfortable with variables might try to write an entire program in one line rather than use any assignments, while another student might write code after a return statement without realizing that it isn't being run.

p1

```
def intToPlayingCard(value):
    suit = ”CDHS”[value / 13]
    faceValue = ”23456789TJQKA”[value % 13]
    return faceValue + suit
```

p2

```
def intToPlayingCard(value):
    x = value/13
    suits = ”CDHS”
    suit = suits[x]
    print suit
    cardvalue = ”23456789TJQKA”
    card = cardvalue[value%13]
    print card
    print card + suit
    return(card + suit)
```

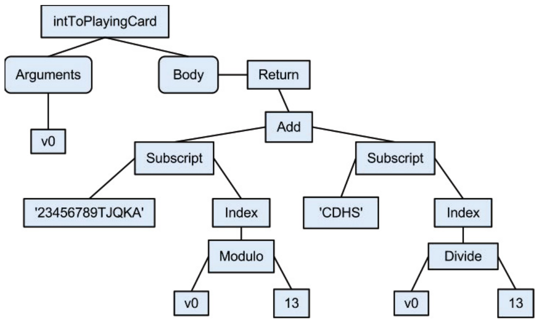

Fig. 1. Above, the two programs shown canonicalize to the same model

At that point, canonicalizing functions can be run over the AST to change it into a format more likely to match other students' submissions. These functions are commonly used in compiler optimizations and result in trees which can be shown to be semantically equivalent [2], so they will not change the student's output. The functions we use currently include:

- Collapsing constant operations
- Propagating expressions assigned to variables
- Using De Morgan's laws to propagate the *not* op inside boolean statements
- Normalizing the direction of comparisons
- Ordering commutative operators with a strict comparison function
- Removing unreachable and unused code
- Inlining helper functions

We did preliminary testing of this model using solutions to basic programming problems taken from an introductory programming course composed of around five hundred students. A median of 70% of the students could be mapped to common solution groups (where groups were composed of 2 to 300 students). Fig. 1 demonstrates how this includes submissions that look completely different on a textual level. We are currently extending the model to work for more complicated problems, and results have been promising (a median of 25% of the students map to groups in multi-function problems using control structures).

Next, we plan to utilize machine learning algorithms to determine the best methodology for creating a clustering of canonicalized models, using unit test results, tree substructures, tokens, and any other information which proves useful to construct the clustering algorithm. We are also considering using text mining techniques on the tokens of the canonicalized abstract syntax trees. We plan to verify the correctness of the resulting algorithms by checking it against original grades and results from unit tests run on the original submissions.

3 Hint Generation and Future Work

The next steps involve experimenting with different ways to generate hints based off of canonicalized models. A few approaches which could be adapted include:

Model Driven: Basic hints could be created based entirely on the student's underlying model and the canonicalizing functions used to create it. This would look for the typical red flags of bad code- unreachable statements, infinite loops, etc.- to give suggestions for improvement. It could also be trained to look for typical novice mistakes, such as off-by-one errors and stylistic mistakes.

Data Driven: In this approach (inspired by work done on creating automatic hints in a logic tutor [3]), the clustering of models would be used in conjunction with compile-time data about how previous programs changed over time until they reached a solution. The solutions found by other students whose models were closest in the clustering would be used to determine the optimal next step for the student asking for a hint.

Crowd Driven: Instead of being programmatically based, this option uses crowd-sourcing amongst students to slowly build a database of hints. Students would type quick conceptual explanations of how they had fixed a problem after progressing past a state; these statements could then be re-used as hints for future students stuck at the same state. A simple voting system could bring the best hints to the top, and a filtering mechanism could keep the explanations from giving away exact solutions.

We plan to continue work on this concept using a corpus of final submissions from the introductory programming course at our university. If this method is successful, we hope to use it in a system for programming instructors requiring little input or upkeep, which would be ideal for the large-scale courses which have become popular recently; in such an environment, solutions would be submitted rapidly enough to provide tutoring for complex problems. We also hope to explore how canonicalization could be used as a method for grouping submissions (for purposes such as general grading and plagiarism detection) and how canonicalizing functions should best be classified for instructor use.

Acknowledgements. This work was supported in part by Graduate Training Grant awarded to Carnegie Mellon University by the Department of Education (# R305B090023).

References

1. Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., Paterson, J.: A survey of literature on the teaching of introductory programming. ACM SIGCSE Bulletin 39(4), 204–223 (2007)
2. Xu, S., Chee, Y.S.: Transformation-Based Diagnosis of Student Programs for Programming Tutoring Systems. IEEE Transactions on Software Engineering 29(4), 360–384 (2003)
3. Barnes, T., Stamper, J.: Toward Automatic Hint Generation for Logic Proof Tutoring Using Historical Student Data. In: Woolf, B.P., Aïmeur, E., Nkambou, R., Lajoie, S. (eds.) ITS 2008. LNCS, vol. 5091, pp. 373–382. Springer, Heidelberg (2008)