Abstract Interpretation of Student Programs as a Strategy for Courseware Development

Colin G. Johnson
University of Kent at Canterbury, England
C.G.Johnson@kent.ac.uk

Abstract

What kinds of feedback can we give to students as part of a computer-based system for supporting programming? One kind of feedback is whether the *idioms* that students use in their programs are typical of experienced programmers. In this paper we talk about how such idioms/patterns/roles/clichés are used in a tacit fashion by experienced programmers, and how natural language tags for such idioms (e.g. *roles of variables*) can be used to articulate this knowledge. Based on these ideas we suggest a strategy for giving feedback to students in courseware: students annotate their program using an annotation language that captures these idioms; then abstract interpretation of programs is used by the courseware to check these annotations and provide focused feedback. As a case study, we show how roles of variables can be annotated and those annotations automatically checked in the *BlueJ* programming environment.

1 Introduction

The canonical error message looks like this:

```
syntax error at line 27
```

In teaching programming, we would also like error messages that look like this

```
semantic error at line 27
```

In this paper we discuss approaches to detecting certain kinds of semantic errors in student-written programs. In particular we distinguish between those semantic concepts that are peculiar to the domain being studied, and those that represent clichés that experienced programmers draw on in program construction.

The idea rests on two basic ideas: an *annotation language* and a technique for *analysing* programs. An annotation language is a semi-formal language through which students can annotate their programs with information about what role a particular programming language structure is playing in their program. Once such annotations have been carried out, *analysis tools* are needed to check whether the annotation is correct. It is typically not possible to do analysis this by running the program on samples of data, instead we need to carry out some kind of static analysis of the program.

We illustrate this by means of a system that we have developed[Bis05, BJ05], which is a BlueJ [KQPR03] extension that checks students annotation of roles of variables [SK05] within a program. In this context the *annotation language* is tagging each variable with a statement of the role that it plays in the program; the *analysis* is a mixture of static analysis of the data-flow graph of the program, and analysis of key phrases in the program text.

The paper is structured as follows. Section 2 discusses the various kinds of knowledge that programmers use when construction programs. Section 3 discusses the idea of using student annotations programming idioms, and automated checking of those annotations. We then present a case study in section 4. Section 5 describes various approaches to checking annotations, and in the final section we raise various questions and discuss future work.

2 Learning to program: syntax, problem domain semantics, and cliché semantics

When a programmer creates a part of a program (e.g. a variable or a control structure) they make use of various kinds of knowledge about the programming language and the problem domain. In learning to program we need to learn about these different kinds of knowledge. However, some of this knowledge is acquired explicitly, supported by pieces of language (whether computer language or natural language) that articulate the ideas. Some of the knowledge is acquired *tacitly* [Pol58]: we do not explicitly articulate the concepts when they are learned, and there is no (natural or computer) language support for the concept. Nonetheless these concepts are acquired by experience.

It has been suggested that we can develop languages to explicitly articulate knowledge that is traditionally acquired tacitly by experience. These languages can then be used by teachers to directly communicate the concepts; by students when they are attempting to learn the concepts; and, by professionals in communicating between each other. The origins of this lie in Christopher Alexander's work [AIS77, Ale79] in the domain of architecture: in this work he attempts to articulate a "pattern language" that makes explicit many of the features that make successful buildings work. This idea has subsequently been adopted in many other domains.

The most explicit use of these concepts in computing is in the idea of a *design pattern* [GHJV94]. Such a pattern consists of a high-level description of how objects playing certain roles to achieve a certain kind of interaction can be most efficiently structured.

We can see such work as an attempt to classify and articulate the structure of *clichés* that are used by experienced practitioners. In many domains, the notion of a cliché is a negative one; for example clichés are seen as something to be avoided in creative writing. However, in more formal domains, clichés reflect patterns of usage that typify the expert. It can sometimes seem strange that most programming activities take a rich computer language and use it in a very narrow way. Actually, though, it is this paucity of expressivity that we want to communicate when we are teaching programming: we want to teach students how to string clichés together in a creative way!

In recent years there has been an attempt to articulate some of the lower level programming clichés in the form of *roles of variables* [BAS04, SK05]. The core idea here is that most variables are used by experienced programmers in one of a small number of ways, if seen at an appropriate level of abstraction. The idea of a role has been summarised thus:

> Variables are not used in programs in a random or *ad-hoc* way but

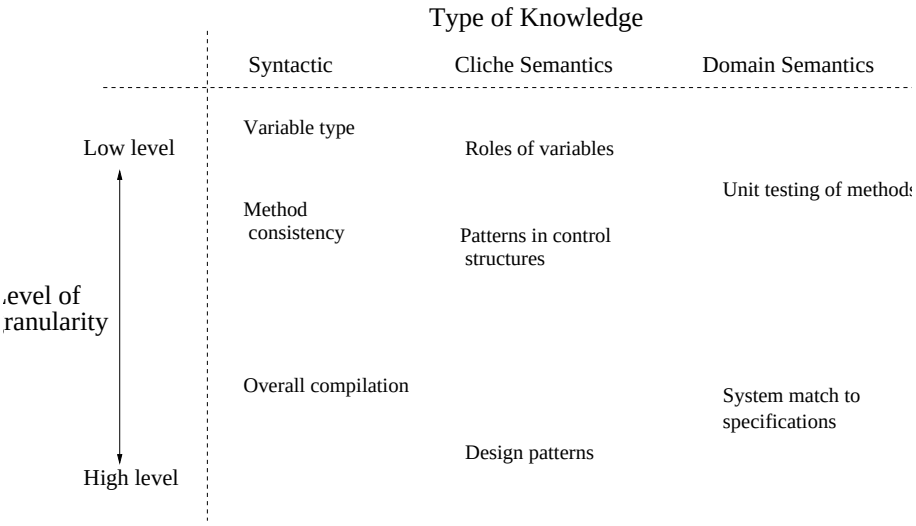

Figure 1: Different kinds of knowledge

> here are several standard use patterns that occur over and over again. In programming textbooks, two patterns are typically described: the counter and the temporary. The *role* of a variable captures this kind of behavior by characteristics the dynamic nature of a variable: the sequence of its successive values as related to other variables and external events.[KS04]

It has been shown that ten roles are sufficient to characterise the usage patterns of around 99% of [KS04, SK05], and that there is a large degree of consistency in the role-assignments made by experienced programmers [BAS04].

Roles of variables can be seen as one example of what we will term the *cliché semantics* of a programming language. Typically, when we think of what something means in a program, we are relating that program-language structure to the domain in which the problem is being solved, or else some intermediate structure that has been created to help solve the problem (the latter can be seen merely as an artificial problem domain).

A diagram summarising some of these different kinds of knowledge about programs, sorted by level of abstraction, is given in figure 1.

3 An approach: annotation and automated checking

In this section we will outline an approach to computer-aided support for learning to program based on explicit use of a language, elements of which refer to roles and clichés within a programming language.

The core idea is as follows: the student will annotate their programs using an annotation language for programming idioms. These annotations will then be checked at compile-time by an automated system. This checking can be carried out in a number of ways: for the purposes of the following section, we will mostly be using static analysis of the control-flow graph of the program; other techniques are suggested in section 5. If the annotations are incorrect according to the check, then a message giving the reason for failure will be returned to the student.

The idea of doing this for problem domain semantics is well known, and underpins much of the work on formal methods. This fails to provide a useful basis for teaching programming, as learning to write a description of "what the program should do" in a formal language is as difficult for beginning programmers as writing the program in the first place!

However, by defintion, the number of roles that a structure can play in a language of programming clichés is small. Therefore it seems reasonable that we might be able to develop systems that can automatically check programs for these clichés, without the student or tutor needing to provide detailed information.

4 Case study: checking roles of variables

In this section we present a case study, *viz.* the *RoleChecker* BlueJ extension. This is described in detail in [Bis05, BJ05].

An image from the BlueJ system, illustrating the state just after compilation of a program with an incorrect role assignment, is given in figure 2. In this system, the student is able to annotate their variable declarations with a comment in a particular form: for example

```
private int fib;  //%%fib%%fixed value%%
```

indicating that the variable `fib` takes the role "fixed value"; i.e. it is a constant. Clearly this is wrong in the program in figure 2; the value changes in the highlighted line. This is detected automatically and an appropriate (semantic) error message given to the student in the box at the lowest part of the window.

The analysis consists of a number of stages (a fuller account is in [Bis05, BJ05]). Firstly, a number of *slices* [Tip95, HH01] of the program are created, one for each annotated variable. A slice is a transformation of a program with respect to a variable v that removes all code that could not possibly change the value of v. For each of these slices, a control-flow diagram (similar to that used by compilers for checking certain properties of programs [ASU85]) is created. This summarises the possible routes that the control-flow of the program can take, and provides a context for the state of the variable in question at each significant point in the program.

For each annotated variable, we then create a list of *features*, drawn from the program text and the control-flow graph, which will help in deciding roles. Two examples are "the variable is directly toggled within a loop" and "the variable appears directly on both sides of an assignment statement". Some of these can be derived directly from the text of a statement involving the variable; in others we need both the text of the statement, *and* its context as given by where that statement appears in the control-flow graph.

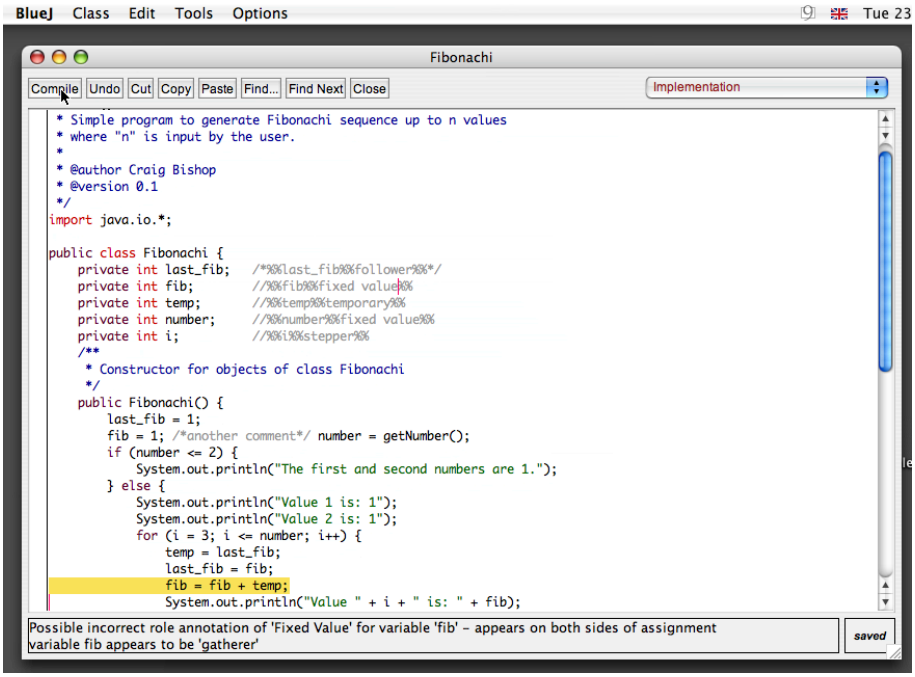

Figure 2: BlueJ with the *RoleChecker* extension running.

Failure conditions for the various roles are then generated. For example, we can say that if a variable is used outside of the block in which it is assigned, then it is *not* playing the role *temporary*; if it is not used in a comparison operation, then it will not be playing the role of *most wanted holder*. Each role has a list of such features, and if any of the features is found, it is concluded that the role cannot be being played by that variable. Individual error messages are generated depending on (1) the role that was asserted by the student and (2) the failure condition that has been triggered.

5 Various approaches to automated checking

In the previous section, we gave an example of how an annotation language describing typical usage could be used to enhance a teaching tool. The idea of roles could, in theory, be extended to other areas of programming. Perhaps the most obvious is a set of roles for control structures such as loops. For example we might characterise a loop as being an iterator through a data structure, a repetition where a fixed action is repeated a certain number of times, an accumulation where something is added to some structure each time the program loops round the loop, ...

In the remainder of this section we attempt to outline various techniques that might be used to check roles: both the roles of variables discussed above; and, a broader set of roles representing other program characteristics.

5.1 Analysing the program text

In the case study above we used direct analysis of the program text as one way to assess when a role assignment had been made incorrectly. Clearly this needs to be done for many different kinds of analysis, however on its own, it is only meaningful where we do not need to know the context for the statement.

In extending the notion of roles to other program features other than variables, we might use direct analysis of the program text. For example, we might analyse the condition for a loop to stop, and determine whether the condition is a constant or whether it depends on something within the loop.

5.2 Analysing the control-flow graph

This is the other main technique that was used in our case study. The control-flow graph summarises the program in the form of a graph that branches whenever a branch or loop structure is met. It is useful in analysing roles because (1) it summarises the entire set of possibilities for routes through a program; and, (2) it provides a summary of the context for a particular statement: is it in a loop, if so what other variables are also active in that loop, when does it go out of scope, et cetera. Many features of the control graph were used as part of the feature set in the case study.

5.3 Model checking

Another way to derive properties of programs is through the use of *model checking* algorithms [HR00, CJGP99]. These algorithms allow the user to specify desired properties of the programs in terms of statements in a temporal logic about how variables change with time. So for example we might provide statements like "A cannot be true until B has become false", "x cannot exceed 10 while y is still positive".

These techniques have the potential to be used for checking roles in a "positive" way, as well as the "negative" way of checking whether a failure condition is met, as we have used above. So, for example, we might model check the *fixed value* role of a variable x by the statement "in all possible future states, x has the value that it has in the starting state". More complex roles can also be written as model checking statements. For example we might define the *most-wanted holder* by the statement that it is always the case that the value of interest increases (with regard to the comparator of interest), and that it always ends up holding the highest value in the set examined.

There would seem to be a lot of potential here; however it is possible that it might be harder to express the roles at a sufficient level of generality as model-checking statements. Even if such statements could be found, it might require students to express more about the variable in question (e.g. "most wanted with regard to what?") rather than concentrating on structural features that suggest the role.

5.4 Machine learning

In the case study above, we created the failure conditions "by hand", looking at programs and their control-flow graphs and determining where role assignment failures could be detected. An alternative approach is to use machine learning methods

[Mit97]to discover patterns in programs that suggest that the program should be classified under a particular role.

In particular, this is an example of a *classification* problem. In such a problem we have a large number of data items, each of which has a certain number of *attributes* and a *class* to which it belongs. In the case of roles in programs, the attributes are features of the program with regard to a particular variable; the class is the role which that variable plays.

An approach to doing this for roles of variables has been suggested by Gerdt and Sajaniemi [GS04]. For each variable in a large database of programs, they annotate the variable with a role. For each variable thus annotated, they derive a set of attribute data consisting of a set of *flow characteristics*, i.e. features of the control-flow graph for the program. They then propose to apply a classification technique to learn which flow characteristics are most strongly correlated with each role.

6 Questions and future work

An important set of questions, which can only be resolved by empirical work, is whether explicit articulation of tacit knowledge really does lead to better learning. Perhaps the cognitive load imposed by the additional language makes it *harder* to understand.

It would be interesting to find out (1) how students use natural language to help them with structuring their work in programming and (2) whether there are examples in other domains of whether explicit labelling of normally tacit concepts, and whether that has helped in learning and teaching.

The core "next step" in this work will be evaluating the effect that this kind of use of articulation of concepts has on learning. This will focus both on the effect on student performance in conventional programming tasks, and directly on measuring the impact on the way students talk about programming, describe programs, and the structure of students concepts about programming (perhaps using methods such as those in [Pet03, San05].

References

[AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language*. Oxford University Press, 1977.

[Ale79] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[ASU85] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.

[BAS04] Mordechai Ben-Ari and Jorma Sajaniemi. Roles of variables as seen by CS educators. In Boyle [Boy04], pages 52–56.

[Bis05] Craig Bishop. Roles of variables and program analysis. Master's thesis, University of Kent at Canterbury, Computing Laboratory, 2005.

[BJ05] Craig Bishop and Colin G. Johnson. Assessing roles of variables by program analysis. In *Proceedings of the 5th Finnish/Baltic Conference on Computer Science Education*, pages 148–153, 2005.

[Boy04] Roger Boyle, editor. *Proceedings of the 9th Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE 2004)*. Association for Computing Machinery, 2004.

[CJGP99] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[GS04] Petri Gerdt and Jorma Sajaniemi. An approach to automatic detection of variable roles in program animation. In A. Korhonen, editor, *Proceedings of the Third Program Visualization Workshop*. University of Warwick Department of Computer Science Report CS-RR-407, 2004.

[HH01] Mark Harman and Rob Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[HR00] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2000.

[KQPR03] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *Computer Science Education*, 13(4), December 2003.

[KS04] Marja Kuittinen and Jorma Sajaniemi. Teaching roles of variables in elementary programming courses. In Boyle [Boy04], pages 57–61.

[Mit97] Thomas Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[Pet03] Marian Petre et al. My criterion is "is it a Boolean?": A card-sort elicitation of students' knowledge of programming constructs. Technical Report 6-03, Computing Laboratory, University of Kent, 2003.

[Pol58] Michael Polyani. *Personal Knowledge: Towards a Post-Critical Philosophy*. Routledge and Kegan Paul, 1958.

[San05] Kate Sanders et al. A multi-institutional, multi-national study of programming concepts using card sort data. *Expert Systems*, 22(3):121–128, 2005.

[SK05] Jorma Sajaniemi and Marja Kuittinen. An experiment on using roles of variables in teaching introductory programming. *Computer Science Education*, 15(1):59–82, 2005.

[Tip95] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995.