Roles of Variables in Teaching

Juha Sorva, Ville Karavirta, and Ari Korhonen
Department of Computer Science and Engineering,
Helsinki University of Technology, Espoo, Finland

jsorva@cs.hut.fi; vkaravir@cs.hut.fi; archie@cs.hut.fi

Executive Summary

Expert programmers possess schemas, abstractions of concrete experiences, which help them solve programming problems and lessen the load on their working memory during problem solving. Possession of schemas is a key difference between novices and experts, which is why instructors need to help students construct them. One recent tool for facilitating schema formation in introductory programming are roles of variables, which represent stereotypes of variable use in computer programs (Sajaniemi, 2002). For instance, a variable with the role STEPPER is assigned values in a systematic and predictable order (e.g. ascending integers 0, 1, 2, ...), whereas a FIXED VALUE is a variable whose value does not change. Roles of variables embody expert programmers' tacit knowledge on variable usage patterns, which can be made explicit and taught to novice programmers. A small set of roles covers the vast majority of variable use in introductory-level programs. Prior results obtained through analysis of examination results and observation of students suggest that using roles of variables in introductory programming education can increase students' skills in comprehending and constructing programs (Byckling & Sajaniemi, 2006; Sajaniemi & Kuittinen, 2005). Little has been published about the experiences of teachers in higher education who have adopted roles in their programming courses, or about the methods of instruction that can be used to introduce roles. The main research questions in this paper are: how can instructors benefit from roles of variables while creating and maintaining their teaching materials, and what kind of influence does roles-based instruction have on teaching and learning in general?

In this paper, we present the experiences of two programming teachers, who have adopted roles of variables into their courses. In one of the two courses, a very casual 'lightweight' approach to introducing roles of variables to students was used. In the other course, somewhat more effort was spent on introducing roles to students. We discuss the changes made to our courses, describe the effects that the process had on our teaching methods and materials, and suggest new ideas on how to make use of roles of variables in pedagogy. Using feedback questionnaires and analyses of examination answers, we also explore what impact roles of variables had on our students.

From a teachers' point of view, we found roles of variables to be very useful. Roles are concrete and easily linked to code. They help with teaching stepwise refinement of program designs, and can be taught in introductory courses naturally alongside other variable-related concepts, such as type and scope. Roles provided us with new ways to assess and improve our teaching materials and methods.

Our assessment of the impact of roles of variables on CS1 students is in agree-

Editor: Zlatko Kovacic

ment with earlier findings and suggests that roles-based instruction can improve students' programming knowledge, perhaps especially with program comprehension tasks. Not all students involved in our experiment found roles useful for learning programming, however, and our results indicate that more than a lightweight introduction to roles is needed in order for students to adopt them into active use. We conclude that roles of variables are a very promising pedagogical tool for introductory programming teachers.

Keywords: roles of variables, introductory programming, CS1, CS2, pedagogy.

Introduction

Learning to program is not easy. Recent experimental studies into students' programming ability have shown that university students in many countries have similar difficulties in writing (McCracken et al., 2001), tracing (Lister et al., 2004), and designing (Eckerdal, McCartney, Moström, Ratcliffe, & Zander, 2006) programs. The emergence of the expert programmer from the novice is a process that involves the formation of multiple mental models, deep and interlinked knowledge hierarchies, and abstract, generalized problem-solving patterns (Winslow, 1996). As programming instructors, our job is to initiate and facilitate this process. In this paper, we look at one tool – roles of variables – which may help us with this task.

Schemas

An expert solves problems fluently and fast and possesses domain-specific problem-solving strategies. A novice is uncoordinated and slow and commonly resorts to generic problem-solving strategies, such as trial-and-error and means-ends analysis. What is a complex problem for the novice may be for the expert merely a task with a familiar solution pattern. One explanation for this difference is suggested by schema theory. Schema theory argues that people bundle similar experiences into cognitive constructs called *schemas*. Schemas are abstractions of concrete cases of experience and have a crucial role in the way people store, organize, and understand information. A schema can be triggered and brought to bear – consciously or unconsciously – in new problem-solving situations similar to previously encountered ones. The importance of schemas, sometimes called *plans*, has been explored in many fields, including computer programming. (See e.g. Adelson, 1981; Chi, Glaser, & Rees, 1982; Détienne, 1990; Fix, Wiedenbeck, & Scholtz, 1993; Rist, 1989; Soloway & Ehrlich, 1984; Sweller & Levine, 1982.) For instance, say a student has encountered examples of **(a)** sequentially searching an array for the smallest integer, and **(b)** searching a linked list of `Movie` objects for the movie with the highest gross income. With enough practice and reflection, encountering such concrete cases contributes towards the eventual formation of a schema for sequential searching, which can be applied to similar problems. An expert has many schemas that accommodate various kinds of scenarios he or she is likely to encounter. A novice, lacking these schemas, flounders.

Cognitive Load

Human working memory is very limited in capacity (Baddeley, 1994), and a novice's working memory resources are greatly taxed by problem-solving situations such as programming. Novice students of programming are commonly asked to complete assignments that require understanding of multiple interacting program components, algorithmic design, the sometimes less-than-obvious program syntax, language semantics, and the notional machine (Du Boulay, 1986). In addition, they may need to use an integrated development environment or similar tools, or to apply third-party libraries in their programs. As the novice juggles all these new concepts in their working memory, it is burdened with a high *cognitive load*. Excessive cognitive load hinders

meaningful learning and schema formation (see e.g. Paas, Renkl, & Sweller, 2003; Tuovinen, 2000).

Schemas combine related bits of knowledge into larger chunks, which can be brought to working memory and processed as a single entity. Sequential searching through a collection of elements, for instance, eventually becomes a single chunk of knowledge whose details do not need to be kept in working memory. With experience, a learner becomes increasingly familiar with common schemas. As this happens, triggering such schemas becomes increasingly unconscious and automatic, further lessening the strain on working memory.

Pedagogical Implications

If schemas play a crucial part in turning a novice into an expert, then it is crucial that we help our students construct them. Eckerdal and Berglund (2005), drawing on work by Hazzan (2003), called for pedagogy that helps programming students discern 'canonical procedures' before maturing towards a more abstract 'object conception'. A 'canonical procedure' is a problem-solving procedure that is more or less automatically triggered by a given problem. We see canonical procedures as closely related to schemas. As students form and internalize schemas, they learn procedures that can be applied instinctively. Fostering schema formation will therefore aid the student in developing canonical procedure.

In addition to facilitating the formation of schemas, it is also important to ensure that students need to cope with as little extraneous cognitive load as possible. Cognitive load can be adjusted by carefully choosing instructional methods and example materials (Tuovinen, 2000) and through curricular redesign (Mead et al., 2006). The two goals of helping students form schemas and managing cognitive load are linked. On the one hand, schemas decrease cognitive load; on the other hand, cognitive load level affects schema formation.

Many recent works in the field of computer science education can be viewed as attempts to support schema formation by making explicit various patterns that occur in programs (e.g. Muller, 2005; Sajaniemi & Kuittinen, 2005; Wallingford, 2003). Of these approaches, *roles of variables* (Sajaniemi & Kuittinen, 2005) are systematically linked to code constructs (i.e., variables), and earlier results suggest that roles provide an excellent coverage of variable use in the implementation of algorithms. For these reasons, and since we felt that variable roles hold intuitive appeal as a succinct way to express algorithm-related schemas and explain and discuss experts' programming knowledge, we integrated roles of variables into our teaching in two courses at Helsinki University of Technology. In this paper, we relate our experiences of this experiment from a teachers' point of view. We will also present some early evidence of how our students took to roles of variables.

Research Questions

Even though roles of variables are a concept that can be utilized in teaching programming to novices, roles also have potential uses in analyzing programs, including large-scale ones, and may be useful to experts as well as novices. In this research, we do not only study our experiences with students, but also report on what instructors have learned from this experiment. Thus, the research questions are two-fold:

1. How can instructors make use of roles of variables in their teaching?

The method we use to study the first question includes adoption of roles in two different courses and collecting evidence of how this contributes to the quality of courses and materials.

2. What kind of influence does roles-based instruction have on learning and teaching in general?

The focus of the second question is more on students and their reactions to the experiment. However, best practices for instructors have also been suggested for teaching novice–level algorithm design.

The next section, *Roles of Variables*, introduces the concept of roles of variables in more detail. In the following section, *Two Cases*, we describe the ways in which roles of variables were used in our teaching and the method we used to examine our students' reactions to the experiment. The section *Results and Discussion* examines how the experiment worked out from the teachers' points of view; the results of our preliminary study of the students are also presented in this section. We conclude with a brief summary of our findings.

Roles of Variables

Roles of variables are stereotypes of variable use in computer programs (Sajaniemi, 2002). Roles embody expert programmers' tacit knowledge on variable usage patterns, which can be made explicit and taught to students (Sajaniemi & Navarro Prieto, 2005). The concept of roles of variables was introduced by Sajaniemi (2002), who analyzed programs written by novice-level programmers. He concluded that 99% of the variables used in novice-level programs can be described using a small set of role names that denote certain common variable usage patterns. The original role set was found through analysis of procedural programs. Since then, the set of roles has evolved, and roles have been applied to object-oriented as well as functional programs. In recursive programs, roles can be applied not only to variables, but to the behavior of parameters and return values over nested recursive calls (Sajaniemi, Ben-Ari, Byckling, Gerdt, & Kulikova, 2006).

Research suggests that students can benefit from instruction that uses roles. Sajaniemi and Kuittinen (2005) analyzed the examination answers of groups of introductory programming students who had received the same amount of instruction through different teaching methods. They found that students who are taught elementary programming using roles of variables gain better program comprehension skills than students taught in an otherwise similar way but without use of roles as a pedagogical tool. Byckling and Sajaniemi (2006) further discovered that roles-based instruction facilitates the development of program construction skills better than traditional instruction. Their results also suggest that the development of program construction skills can be aided by the use of a role-based program animator tool instead of a regular visual debugger. Ben-Ari and Sajaniemi (2004) showed that the concept of roles of variables is easily grasped by computer science educators. So far, little has been published about the experiences of teachers in higher education who have adopted roles in their programming courses or about the methods of instruction that can be used to introduce roles.

In the following, we briefly introduce each role in the current version of Sajaniemi's role set. Like Sajaniemi, we have divided the roles into a group of eight general roles that can apply to variables of any type, and a group of three roles related to data structure use. For a more verbose introduction to each role and concrete program examples, we refer the reader to Sajaniemi (2003).

General Roles

1. A variable has the role FIXED VALUE if the variable's value is not changed after it is initialized.
2. A variable has the role STEPPER if it is assigned values in a systematic and predictable order. An example of a STEPPER is an index counter used when looping through array elements.
3. A variable has the role MOST-RECENT HOLDER if it holds the latest value in a sequence of unpredictable data values. For instance, a MOST-RECENT HOLDER could be used to store the latest

element encountered while iterating through a collection of data elements, or the latest value that has been assigned to an object's attribute (i.e., to an instance variable that is a MOST-RECENT HOLDER) by a setter method.

4. The role MOST-WANTED HOLDER describes variables that hold the 'best' value encountered in a sequence of values. Depending on the program and the type of the data, the ''best' value may be the largest, smallest, alphabetically first, or otherwise most appropriate value.
5. A variable has the role GATHERER if the variable is used to somehow combine data values that are encountered in a sequence of values, and the variable's value represents this accumulated result. For instance, a variable keeping track of the balance of a bank account object (the sum of deposits and withdrawals) is a GATHERER.
6. A FOLLOWER is a variable that always holds the most recent previous value of another variable. Whenever the value of the followed variable changes, the value of the FOLLOWER is also changed. For example, the 'previous node pointer' used in linked list traversal is a FOLLOWER.
7. A variable is a ONE-WAY FLAG if it only has two possible values and if a change to the variable's value is permanent. That is, once a ONE-WAY FLAG is changed from its initial value to the other possible value, it is never changed back. For example, a Boolean variable keeping track of whether or not errors have occurred during processing of input is a ONE-WAY FLAG.
8. A variable has the role TEMPORARY if the value of the variable is needed only for a short period. For example, an intermediate result of a calculation can be stored in a TEMPORARY in order to make it more convenient or efficient to use in later calculations.

Roles Related to Data Structures

1. An ORGANIZER is a variable that stores a collection of elements for the purpose of having that collection's contents rearranged. An example of an ORGANIZER is a variable that contains an array of numbers during sorting.
2. A variable is a CONTAINER if it stores a collection of elements in which more elements can be added (and, typically, can be removed as well). For example, a variable that references a stack could be a CONTAINER.
3. A WALKER is a variable whose values traverse a data structure, moving from one location in the structure to another. For instance, a variable that contains a reference to a node in a tree traversal algorithm, and a variable that keeps track of the search index in a binary search algorithm can be considered to be WALKERS.

Two Cases

At Helsinki University of Technology, roles of variables were first adopted in the second programming course (CS2) lectured by the third author in Spring 2006. In Fall 2006, roles were used in an introductory programming course (CS1) taught by the first author. The following subsections describe what measures we took to make use of roles of variables in our courses. The reader should note that despite the fact that students take CS1 before CS2, our experiment involves one set of CS2 students and another set of (the following year's) CS1 students. Therefore, none of our students enrolled in either course had been previously taught roles of variables.

CS2 - Data Structures and Algorithms

Our CS2 course teaches data structures and algorithms independently of any programming language. While we make use of code and pseudocode examples, the focus is not on program code but on a more abstract level. Students are required to write little code, since our goal is to give a broader overview of data structures and algorithms than would be possible in a course that includes many coding tasks. The topics covered include basic data structures, sorting, priority queues, dictionaries (balanced search trees and hashing), and graph algorithms. The course also covers basic algorithm analysis. We examine a number of different sorting algorithms, for instance, in order to demonstrate the fact that there is no single sorting method that is conclusively better than the rest.

'Lightweight' introduction of roles

We did not go out of our way to change the examples we used or the order in which they were presented in order to accommodate for roles of variables. Instead, we used existing examples and attached roles of variables to them by annotating course materials with role names. We introduced roles very briefly as they appeared in existing examples. We discussed examples using role names quite casually, and no extra effort was put into defining the roles for the students. Roles were not learning objectives of the course, but merely a learning aid, which students could make use of if they wished. Lecturer's notes included links to the roles of variables home page (Sajaniemi, 2003) in case someone wanted to learn more. In previous experiments with using roles in teaching, more effort was made to define each role to students (Kuittinen & Sajaniemi, 2004); our approach in the CS2 course was decidedly more casual and 'lightweight'.

There were two groups of students in the CS2 course, which we refer to as CS majors (Computer Science students) and CS minors (students from other engineering disciplines such as Electrical Engineering and Engineering Physics). Lectures were the same for both groups, and exposure to roles of variables through lecturer's notes and slides was identical for both CS majors and minors.

In addition to lectures, CS majors had classroom exercise sessions, which made use of code examples annotated with role names. CS majors, therefore, had an additional chance to discuss roles in class, although we did not require the use of roles in the classroom sessions either. The classroom exercises covered basic algorithm analysis, designing short new algorithms, simple proofs, etc. Thus, only a subset of the exercises was such that roles were applicable. For each exercise, one of the students was selected to present his or her solution to the rest of the group. If students wanted to use roles when explaining a solution, it was up to them to do so; we did not specifically encourage it. In addition, even though our teaching assistants were aware of the existence of roles of variables as a concept, we did not instruct them to pay any specific attention to roles.

Instead of classroom exercises, CS minors had a teamwork assignment, which was also not strongly linked to roles of variables. In this assignment, students were asked to design a solution for a larger software application. Typically, several algorithms and data structures were needed in order to complete the assignment. Again, if roles were used to explain a team's solution, this arose voluntarily from the students themselves.

Assessing impact on students

We wanted to see how the 'casual' introduction of roles of variables affected our CS2 students. To find out whether or not students found roles of variables useful, we wrote a multiple choice question in our CS2 course feedback form. Students were asked to select the statement that best described how they felt about roles of variables. Giving feedback was voluntary and anonymous,

but students were awarded with one extra point for the final examination if they filled the feedback form.

We wished to find out whether or not students actually learned about roles of variables. We assumed that if our students had learned about roles, they would demonstrate this by using role names in their descriptions of the behavior of algorithms and in their algorithm implementations. To find out if this was the case, we went through the final examination papers of the CS majors and counted the occurrences of role names in their answers. We chose this group of students since they had had more exposure to roles through the classroom exercises. Moreover, as indicated by the results presented later in this paper, our CS majors had found roles of variables somewhat more helpful than our CS minors. We analyzed the answers to two questions in the examination. The first question required the students to explain the behavior of the Quicksort algorithm given in pseudocode. For the second question, each student had to design a depth-first search (DFS) algorithm using pseudocode or some programming language and then explain the behavior of the algorithm.

CS1 - Basics of Programming

Our CS1 course introduces students to computer programming using object-oriented programming and the Java programming language.

General introduction to roles

Variables are a fundamental programming construct, and they appear in programs in numerous different ways. Depending on programming paradigm and language we can have variables containing numbers, characters, objects, pointers, and references. We can have local variables, global variables, static variables, instance variables, and parameter variables. We can have STEPPERS, MOST-WANTED HOLDERS, and FIXED VALUES. The lists go on. To help our CS1 students navigate this conceptual and terminological jungle, we presented roles as an aspect of variables to be placed side by side with two other aspects that are routinely and explicitly taught in CS1: type and scope. (We would like to note that different uses for variables – i.e., roles – are also routinely taught in CS1, but not always explicitly discussed or named.) We emphasized how each variable is characterized by these three aspects; to illustrate this, students were shown diagrams similar to those in Figure 1. This was done during week 2 of our CS1 course when the term role of variable was first introduced.

Introduction of individual roles

As in the CS2 course in the previous spring, we again did not set out to restructure our course materials and, instead, introduced roles as they appeared in existing examples. As a consequence of some idiosyncrasies in course structure, roles were not introduced quite in the order suggested by Kuittinen and Sajaniemi (2004). As we use Java `ArrayList`s very early on, the role CONTAINER was presented at the beginning of our course, for example, and we ended up introducing instance variables that are MOST-WANTED HOLDERS before introducing MOST-RECENT HOLDERS.

Following the advice of Sajaniemi (2003), each time a new role was encountered, it was defined to the students informally in course materials and during lectures, with an emphasis on how this new role differs from previously encountered ones. At this point, we also listed some example situations where the new role may be useful and discussed how variables with this role typically appear in Java source code. Each role was associated with an illustration as it was introduced. The illustrations we used were mostly different ones than the visual metaphors used in the PlanAni animator tool (Sajaniemi & Kuittinen, 2003). As the course proceeded, we continued to point out

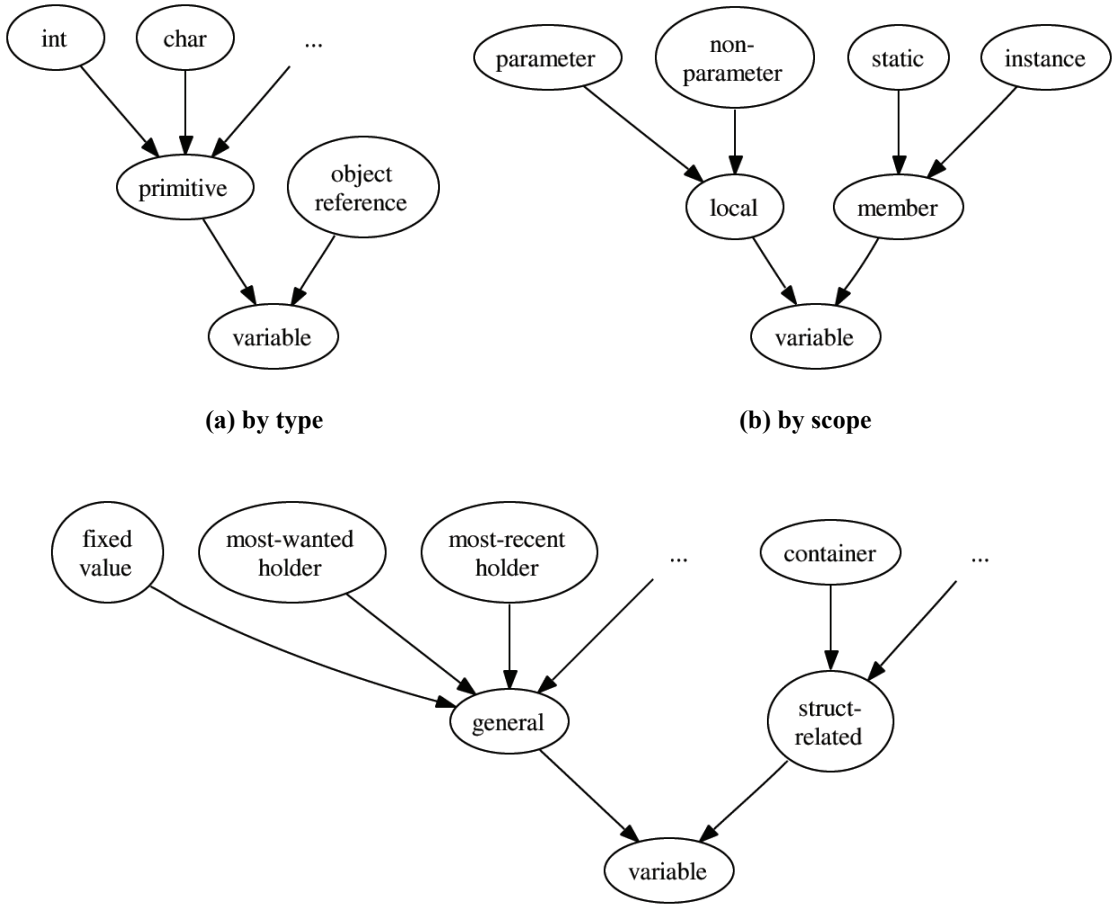

Figure 1: Categorizations of Java variables

cases where a previously introduced role was encountered again in lecture examples, and used roles to draw parallels between examples. Students who did not come to lectures had less exposure to roles than those who did, despite the availability of lecture notes. (Lecture attendance was voluntary.)

The programming assignments in our CS1 course require students to read plenty of given program code, which they then extend, modify, and correct. We annotated this given code – a few thousand lines of commented code in total – with comments indicating the role of each variable.

Not all variables were annotated in this way, however; parameter variables were considered FIXED VALUEs 'by default', for instance.

Roles and algorithm design

When writing code of their own, we wanted our students to think about the intended use of each variable (the role) just as they think of the kind of data to be stored in the variable (the type). To this end, we used roles as a tool during lectures when discussing how to implement the classes and methods in Java programs.

In algorithmic design, we commonly used stepwise refinement (Wirth, 1971) to progress from a pseudocode solution towards actual Java code. Roles served as a stepping stone in this process,

helping us bridge the gap from idea to code. After an initial pseudocode solution was created, we discussed the kinds of programming constructs (loops, variables, etc.) needed for the actual implementation. As a part of this process, we selected roles and types for variables. We then refined our pseudocode to explicitly use these constructs. Students were encouraged to follow a similar procedure when working on their assignments.

A simple example of an algorithm in pseudocode that makes use of roles is shown in Figure 2. Given a collection of `Movie` objects, the algorithm finds and returns the one with the highest box office income. (The reader may note that this example is not explicit about how `greatestSoFar` is initialized and what happens when its initial value is compared to the first element in the collection. This can lead to an in-class discussion about the kind of bugs you can expect if you neglect to initialize a MOST-WANTED HOLDER properly, and to a more refined pseudocode solution.)

```
Say greatestSoFar is a MOST-WANTED HOLDER of type Movie.
Say current is a MOST-RECENT HOLDER of type Movie.
For each element in the collection this.movies:
  1. Store the element in the MOST-RECENT HOLDER current.
  2. If current has grossed more than greatestSoFar, it is now the
     most wanted value. Assign the value of current to greatestSoFar.
Finally, return the value of greatestSoFar.
```

Figure 2: Pseudocode using role names

Assessing Impact on Students

In our online course feedback questionnaire at the end of CS1, we included a few questions related to roles of variables. Using a four-point Likert-like scale, students were asked whether they saw roles of variables as something useful, whether roles had helped them understand given code, whether roles had helped them understand examples of step-by-step method design using pseudocode, and whether roles had helped them design and write their own programs. Answering the questionnaire was an obligatory part of passing the CS1 course, but the answers were nevertheless processed anonymously. We had some non-Finnish-speaking students taking the course but their course feedback is not considered in this paper, since they had different materials and a different feedback form than the Finnish-speaking majority. The feedback questions asked of the students were not quite identical in the CS1 course compared to those used in the CS2 course. The CS1 questions were designed later than the CS2 ones and probed the topic in some more detail; this was deemed to be useful by the teacher for reasons of course development, despite having an adverse effect on the comparability of the feedback results of the two courses.

What We Did Not Do

In the CS2 course in particular, the way we introduced roles was decidedly ‘lightweight’. We did not spoon-feed roles of variables to those of our students who were not interested in learning them. Although various CS1 lecture examples in particular will have been more difficult to understand without grasping role terminology, it was perfectly possible for a student so inclined to ignore any explicit teaching of variable roles and still pass the courses. We had no assignments in either course specifically about roles, nor did we require students to explicitly use roles in any assignment. We did not use roles-based animations of programs in either course (cf. Sajaniemi & Kuittinen, 2003).

Results and Discussion

In the following, we take a retrospective look into the process of applying roles of variables in the CS2 and CS1 courses as outlined in the previous section.

CS2 - Data Structures and Algorithms

The original motivation to adopt roles of variables into our teaching was to let the instructors experiment with these novel concepts and to monitor how students assimilate this new knowledge. As it turned out with the CS2 course, the experiment was a learning experience perhaps especially for the instructor.

Roles of variables helped us develop our materials and teaching processes further. This happened not only because roles helped us provide additional documentation of algorithm behavior in code and pseudocode, but also because adopting roles forced us to re-examine our program examples in detail. This brought to light some anomalies, which had previously gone unnoticed. Some 25 different program examples were annotated, five of which needed to be transfigured. We give two examples of these in the following subsections. We then report on the same experiment from the students' point of view.

Conflicting roles and names of variables

The name given to a variable can be very misleading if it is not consistent with the way the variable is used. The code fragment in Figure 3 is an example taken from the lecturing material that we used in CS2 before adopting roles. This material made widespread use of variables named `temp`. In Figure 3, the role of the variable `temp` is not a TEMPORARY even though its name suggests such an interpretation. The variable is a GATHERER, which iteratively accumulates a hash value for a key string. Each intermediate result stored in the GATHERER variable represents the hash value for an increasingly long prefix of the key string `k`. Each consecutive hash value is calculated based on the previous one, until the hash value for the entire key string has been formed. The variable name `hashValueOfPrefix` would be more informative and more descriptive of the variable's role in the algorithm.

```
1  temp = 0;
2  for (i = 0; i < key_length; i++)
3     temp = ( (32*temp) + value(k[i]) ) modulo N;
```

Figure 3: A code fragment, which uses Horner's rule to determine the hash value of a key string `k`.

Obviously, one can improve uninformative variable names without thinking about them in terms of roles. However, we found roles helpful in that they triggered this process of careful examination of variable names in our CS2 examples, and provided a framework that enabled us to systematically examine and discuss the ways variables are used and how this usage should be reflected in variable names.

Variables with dual responsibilities

As we looked through our CS2 teaching materials, it turned out that some variables in our examples had ambiguous behavior. A single variable was sometimes used for multiple different tasks, and closer examination revealed that minor changes to code removed this ambiguity and significantly clarified such examples.

In Figure 4(a), a recursive implementation for mergesorting linked lists is presented. The local variables `first` and `last` are both FIXED VALUEs; `i` is a STEPPER. The behavior of `N` over recursive calls is that of a STEPPER. (It indicates the length of the linked list that is currently being processed. The length decreases systematically as the recursion deepens.) The behavior of the variable `list` is less easy to define. Locally within the function body, `list` is used as a WALKER, and traverses through the linked list sequentially, searching for the middle point. Its recursive behavior as a parameter is also that of a WALKER as it keeps track of locations in the data structure. However, the manner in which the variable 'walks' through the linked list during recursion is different from the sequential traversal that is used locally within a method activation. In recursion, the value of the parameter `list` jumps to the beginning of a sublist at each recursive call. This dual responsibility of the variable can be confusing for novice programmers – and experts – who are trying to grasp the algorithm.

```
 1 node mergesort(node list; int N) {
 2
 3   node first,
 4        last;
 5   int i;
 6   if (list->next == NULL) {
 7     return list;
 8   } else {
 9     first = list;
10     // locate middle point
11     for (i = 2; i <= (N / 2); i++)
12       list = list->next;
13     last = list->next;
14     list->next = NULL;
15     return merge(
16       mergesort(first, N / 2),
17       mergesort(last, N-(N/2)));
18   }
19 }
```

(a) before

```
 1 node mergesort(node first; int N) {
 2     /* first: walker, N: stepper */
 3   node curr;    /* curr: walker   */
 4   node mid;     /* mid: temporary */
 5   int i;        /* i: stepper     */
 6   if (first->next == z) {
 7     return first;
 8   } else {
 9     curr = first;
10     // locate middle point
11     for (i = 2; i < (N / 2); i++)
12       curr = curr->next;
13    mid = curr->next;
14    curr->next = NULL;
15    return merge(
16      mergesort(first, N / 2),
17      mergesort(mid, N-(N/2)));
18    }
19  }
```

(b) after

Figure 4: Implementations for the mergesort algorithm before and after the process of role-based code clarification.

In Figure 4(b), the code has been changed in order to prevent the dual WALKER role of the variable `list`, and the role of each variable is marked explicitly in the code. In the loop on lines 11 and 12, a non-parameter local variable `curr` is used for sequential list traversal. A separate parameter variable `first` is used for the recursive WALKER behavior. (In the case of Figure 4, the number of variables in the code did not increase. The change was effected by analyzing the responsibilities of each variable and reassigning these responsibilities to the existing variables.)

Again, such clarification of source code is something that can be accomplished without using roles of variables. Roles-based analysis of programs can and did, however, help in locating problematic examples.

Assessing impact on students

Table 1 shows the results of the multiple choice question in our CS2 feedback form. The number of students who chose each answering option is indicated on the corresponding line in the table, as is the relative popularity of each answer. We examined the reactions of CS majors and CS mi-

nors separately since their exposure to roles during the course had been different. We did Pearson's chi-square test for the data and the probability of a chi-square of 6.42 with four degrees of freedom is p=0.1696. We would have rejected the null hypothesis "there is no difference between the data sets" if the p-value were smaller than the significance level [alpha] = 0.05. However, this was not the case and we concluded that there were no statistically significant differences between the CS majors and CS minors. As indicated by Table 1, almost one third of the CS2 students were oblivious to roles of variables. Another third did not understand roles or did not consider them useful. The remaining third of the students considered roles of variables interesting or useful (though not necessarily for the purposes of our CS2 course).

Table 1: CS2 students' opinions about roles of variables

Answer	CS majors		CS minors	
	N	%	N	%
I have not heard of roles of variables	25	30	98	32
I did not quite understand the roles	14	17	73	24
I understood the roles but do not consider them useful	16	20	39	13
Roles helped me understand algorithms	21	26	60	19
Roles were interesting but did not help with this course	6	7	40	13

As noted above, we analyzed the final examination answers of the CS majors to see if roles of variables featured in their answers. Four out of the 87 CS majors who participated in the final examination used role names in their description of the Quicksort algorithm. Two students clearly used the name of a variable role in their answers to the DFS algorithm design question. All these students used the role names correctly. Three more students used role-like terminology in their answers ('flag', 'current node' (x2)). In an earlier experiment, Kuittinen and Sajaniemi (2004) had found that of students that had been taught using roles throughout a CS1 course, 35% used role names when asked to freely summarize a given program in an examination. Due to the different nature of our course and examination question, our result can not be directly compared to this prior result. Nevertheless, it seems clear that a lightweight introduction in CS2 has a significantly lesser impact on students than that observed by Kuittinen and Sajaniemi.

We did not, then, observe widespread use of roles of variables by students who had received a lightweight introduction to the concept. The most obvious explanation is that the limited amount of teaching time we allocated for roles in the CS2 course is not enough for the roles to be adopted into active use by students. A more emphatic use of roles in teaching could result in better learning of roles and of the course topics. An alternative explanation for the limited use of roles by students is that the algorithms in question have competing vocabulary strongly related to the algorithm itself. In the Quicksort explanations in the examination, for example, the term *pivot* was used in most of the answers. Similarly, in DFS, it is natural to divide the vertices of the graph in three parts: visited, fringe, and unvisited. The relatively high number of DFS-specific terms used in the students' answers indicates an understanding of the algorithm on a higher level of abstraction than the code level where roles of variables operate.

There was a sizable minority of students who felt they benefited from roles of variables. We find this result acceptable in light of the very small costs of the lightweight introduction we used, despite the lack of widespread adoption of roles by students. In the future, our students will have had a greater exposure to roles before they even start the CS2 course.

CS1 - Basics of Programming

Teacher perspective

We found the inclusion of roles in our CS1 teaching materials to be exciting and easier than we expected. Annotating our example materials, redesigning pseudocode examples, and writing introductions for each role did not take many days, even if one includes the time it took to read up on relevant roles literature. We found that many of our examples were readily explained using roles. We felt that roles made it easier to point out what similarities there were between different examples and how programming patterns/recipes could be applied to similar tasks, using the same combination of roles. Roles helped us document how multiple code locations that make use of the same variable work together and implement a 'delocalized plan' to accomplish a task (cf. Soloway, Pinto, Letovsky, Littman, & Lampert, 1988). It seems likely to us that all this aids students in schema formation and helps instill them with 'canonical procedure'.

Prior to refurbishing our teaching materials, we had expected roles to be a useful documentative aid suitable in particular for the procedural parts of our programs (local variables inside methods). We had had some reservations about how well the roles concept is applicable to object-oriented programs. As a result, we were pleasantly surprised to find that member variables could also be annotated quite conveniently, which often succinctly clarified the inner workings of classes.

Further windfall from adopting roles appeared as we were able to identify points in our lecture sequence where multiple roles were introduced in quick succession. In one case, we had even introduced two new roles in the same example, possibly causing an unnecessarily high cognitive load for our students. Identifying such examples and restructuring them so that different ways of using variables appear more gradually may help with cognitive load management.

In our course materials, most roles were used for both member variables and local variables. Annotating a member variable and a local variable with the same role name indicates that we think of them as similar. However, our experience suggests that in many people's perception a MOST-RECENT HOLDER member variable, for instance, is used rather differently than a MOST-RECENT HOLDER local variable. A settable attribute of an object (the name of a person object, say) is experienced as being quite different from a local variable that stores the most recent element encountered in a collection during iteration. While such variables do have a certain similarity in usage, this similarity is not very obvious to all. We are undecided as to whether it is a similarity that needs or should be emphasized to novices. This kind of dividedness of roles is potentially confusing and may pose a challenge to instructors and role set developers.

We can not discard out of hand the argument that learning roles of variables adds to students' cognitive load as they have to struggle with even more variable-related terminology. However, we would like to stress that when we used roles to teach variable use, we did not teach our students to do anything with variables that we would not otherwise have taught. Roles just provided us with a vocabulary to better discuss these topics. We find it unlikely that any cognitive load inherent in learning roles would not be compensated for by the way roles clarify program behavior. We feel that systematically presenting roles as a 'third dimension' of variables as per Figure 1 may further assist in the matter.

Assessing impact on students

We observed that some CS1 students started voluntarily using role names when discussing programs during the course, and at least a few expressed chagrin when faced with the task of reading code that was not annotated with role names. On the other hand, there were other students who had been quite oblivious to roles despite their inclusion in teaching. No clear complaints about

using roles reached the ears of the instructors, although a few of the more experienced programmers in our class argued that they were a 'trivial' waste of time. This perhaps serves to underline the fact that roles are a vehicle for conveying tacit expert knowledge to novices.

The results of our course questionnaire indicated that the vast majority (over 90%) of the 290 Finnish-speaking respondents had at least heard of roles of variables by the end of the CS1 course (see Table 2). As indicated in Table 3, when asked whether roles of variables seemed useful, over three quarters of the students who commented on the issue at least 'somewhat agreed.' The majority also agreed that roles had helped with understanding given code and method design examples. A sizable minority of students felt that roles had at least somewhat helped them when they designed or wrote their own programs. In general, a larger proportion of programming beginners felt they had benefited from roles compared to students with prior programming experience; the same was true of students who attended more lectures compared to students who did not come to many lectures.

Table 2: CS1 students' opinions about roles of variables

Answer	N	%
I had not heard of roles of variables before reading this question	24	8
I had noticed roles but did not pay too much attention to them	170	59
I studied and/or made use of the roles during the course	92	32

Table 3: CS1 course students opinions about roles of variables continued (0=no comment, 1=totally disagree, 2=somewhat disagree, 3=somewhat agree, 4=totally agree)

Statement	0	1	2	3	4
Roles of variables seem potentially useful	33	10	47	166	34
Roles in the program examples helped me understand how the code works	44	32	65	107	42
Roles of variables in the pseudo-code design examples helped me understand the examples	64	16	49	125	35
Roles of variables were useful when designing and writing code of my own	50	45	91	82	20

Our assessment of the impact of roles is based on students' self-evaluations of the usefulness of roles, a different method than the external analyses of Sajaniemi and his colleagues (Byckling & Sajaniemi, 2006; Sajaniemi & Kuittinen, 2005). Our results conform to those prior results and suggest that roles of variables can help students in gaining programming knowledge. The fact that our students felt roles were less useful for program construction tasks compared to program comprehension tasks agrees with the earlier finding that introducing roles without using role-based simulations of programs may have a limited impact on program construction skills (Byckling & Sajaniemi, 2006).

Conclusions

In this paper, we have described our experiment of integrating roles of variables into our teaching on two computer science courses. Our motivation for this experiment arose out of a wish to learn better how instructors could benefit from roles of variables in their work and enhance students' schema formation as well as to manage students' cognitive load during introductory programming education. We have discussed the effects of these experiments on a teacher's work, described new

ideas on how to make use of roles in pedagogy, and assessed the effect of the experiment on the students.

From a teachers' point of view, we found roles of variables to be very useful. Roles are concrete, easily linked to code and to pseudocode designs, and we found them easy to use both as a documentative tool and when doing stepwise refinement of programs. It feels natural to teach them in introductory programming alongside other variable-related concepts, such as type and scope. In short, roles provided us with new ways to assess and improve our teaching materials and methods. In some cases, examples in teaching materials even became clearer for the instructor himself. Surely, this can only be good for the students as well.

We have no clear evidence as to whether using role names significantly increases students' cognitive load when the concept and terminology need to be learned and understood. However, we believe that the long-term benefits of facilitating schema formation are very likely to outweigh any initial costs of learning roles. This matter calls for further study.

Our assessment of the impact of roles on CS1 students agrees with earlier findings and suggests that roles-based instruction can improve students' programming knowledge, perhaps especially with program comprehension tasks. However, not all students involved in our experiment found roles useful for learning programming, and it is clear that some of our students were not motivated to learn about roles. This was particularly evident in our CS2 course, where fewer than 25% of the students felt that roles of variables had helped them understand algorithms. We must use caution when drawing conclusions from this result. The result does not mean that roles of variables do not help novices understand algorithms. Indeed, we believe that many more of our students could have benefited from thinking about programs in terms of roles, had they chosen to do so. The result quoted above may say more about the 'lightweight' manner in which we introduced roles in the CS2 course. One third of the students did not even recognize the term role of variable at the end of the course. If many students do not even consider adopting roles of variables into their thinking when roles are introduced 'casually', then we should examine the phenomenon further and find ways to improve our teaching of roles. This is in line with the previous studies that suggest using role-based program animators to teach roles of variables more explicitly rather than lightweight (Sajaniemi & Kuittinen, 2005). Other ways to introduce roles of variables, however, could also be considered. For example, an explicit requirement to think about roles of variables in exercises might work to this effect.

In conclusion, we can say that our experiments with roles of variables have been very encouraging. Our experiences and results suggest that roles of variables have the potential to be an excellent tool to teachers of introductory programming, and we are looking forward to making even better use of roles in the future.

Acknowledgements

This work was supported by the Academy of Finland under grant number 210947.

References

Adelson, B. (1981). Problem solving and the development of abstract categories in programming languages. *Memory and Cognition, 9*(4), 422–433.

Baddeley, A. (1994). The magical number seven: Still magic after all these years? *Psychological Review, 101*(2), 353–356.

Ben-Ari, M., & Sajaniemi, J. (2004). Roles of variables as seen by CS educators. *SIGCSE Bulletin, 36*(3), 52–56.

Byckling, P., & Sajaniemi, J. (2006). Roles of variables and programming skills improvement. *SIGCSE Bulletin, 38*(1), 413–417.

Chi, M., Glaser, R., & Rees, E. (1982). Expertise in problem solving. In R. Sternberg (Ed.), *Advances in the psychology of human intelligence* (pp. 7–75). Hillsdale, NJ, USA: Erlbaum.

Détienne, F. (1990). Expert programming knowledge: A schema-based approach. In J. M. Hoc, T. R. G. Green, R. Samurcay, & D. J. Gilmore (Eds.), *Psychology of programming* (pp. 205–222). Academic Press.

Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research, 2*(1), 57–73.

Eckerdal, A., & Berglund, A. (2005). What does it take to learn 'programming thinking'? *Proceedings of the First International Computing Education Research Workshop*, 135–143.

Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., & Zander, C. (2006). Can graduating students design software systems? *SIGCSE Bulletin, 38*(1), 403–407.

Fix, V., Wiedenbeck, S., & Scholtz, J. (1993). Mental representations of programs by novices and experts. *Proceedings of the SIGCHI conference on Human factors in computing systems,* 74 - 79.

Hazzan, O. (2003). How students attempt to reduce abstraction in the learning of computer science. *Computer Science Education, 13*(2), 95–122.

Kuittinen, M., & Sajaniemi, J. (2004). Teaching roles of variables in elementary programming courses. *SIGCSE Bulletin, 36*(3), 57–61.

Lister, R., Seppälä, O., Simon, B., Thomas, L., Adams, E. S., Fitzgerald, S., et al. (2004). A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bulletin, 36*(4), 119–150.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B., et al. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bulletin, 33*(4), 125–180.

Mead, J., Grey, S., Hamer, J., James, R., Sorva, J., St. Clair, C., & Thomas, L. (2006). A cognitive approach to identifying measurable milestones for programming skill acquisition. *SIGCSE Bulletin, 38*(4), 182–194.

Muller, O. (2005). Pattern oriented instruction and the enhancement of analogical reasoning. *Proceedings of the First International Computing Education Research Workshop*, 57–67.

Paas, F., Renkl, A., & Sweller, J. (2003). Cognitive load theory and instructional design: Recent developments. [Introduction to special issue]. *Educational Psychologist, 38*(1).

Rist, R.S. (1989). Schema creation in programming. *Cognitive Science, 13*, 389–414.

Sajaniemi, J. (2002). An empirical analysis of roles of variables in novice-level procedural programs. *Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, 37–39.

Sajaniemi, J. (2003). The roles of variables home page. Retrieved December 5, 2006, from http://cs.joensuu.fi/~saja/var_roles/

Sajaniemi, J., Ben-Ari, M., Byckling, P., Gerdt, P., & Kulikova, Y. (2006). Roles of variables in three programming paradigms. *Computer Science Education, 16*(4), 261–279.

Sajaniemi, J., & Kuittinen, M. (2003). Program animation based on the roles of variables. *Proceedings of the 2003 ACM symposium on Software visualization*, 7–16.

Sajaniemi, J., & Kuittinen, M. (2005). An experiment on using roles of variables in teaching introductory programming. *Computer Science Education, 15*(1), 59–82.

Sajaniemi, J., & Navarro Prieto, R. (2005). Roles of variables in experts' programming knowledge. *Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group*, 45–159.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering, 10(5)*, 595–609.

Soloway, E., Pinto, J., Letovsky, S., Littman, D., & Lampert, R (1988). Designing documentation to compensate for delocalized plans. *Communications of the ACM, 13*(11), 1259–1267.

Sweller, J., & Levine, M. (1982). Effects of goal specificity on means-ends analysis and learning. *Journal of Experimental Psychology: Learning, Memory and Cognition, 8*, 463–474.

Tuovinen, J. E. (2000). Optimising student cognitive load in computer education. *Proceedings of the Australasian Conference on Computing Education,* 235–241.

Wallingford, E. (2003). The elementary patterns home page. Retrieved December 5, 2006, from http://www.cs.uni.edu/~wallingf/patterns/elementary/

Winslow, L. E. (1996). Programming pedagogy – A psychological overview. *SIGCSE Bulletin, 28*(3), 17–22.

Wirth, N., (1971). Program development by stepwise refinement. *Communications of the ACM, 14* (4), 221–227.

Biographies

Juha Sorva is a programming instructor and doctoral student at Helsinki University of Technology, Finland. He received his M. Sc. (Computer Science) from the same university. His research interests are introductory programming pedagogy, novices' understanding of program execution and the underlying computer, educational software, and other topics in the field of computer science education research. His hobbies include schema formation and struggling with cognitive load.

Ville Karavirta is a researcher at Helsinki University of Technology. He received his M.Sc. (Computer Science) in 2005 and is currently a PhD student at HUT. His research interests are computer science education, algorithm visualization, and electronic learning environments.

Ari Korhonen is a researcher and instructor at Helsinki University of Technology (HUT). He received his M.Sc. (Computer Science) in 1997, and his D.Sc. (Tech) diploma in 2003. His research includes data structures and algorithms in software visualization, various applications of computer aided learning environments, and automatic assessment in computer science education.