

Roles of Variables in Three Programming Paradigms

Jorma Sajaniemi

Department of Computer Science, University of Joensuu*

Mordechai Ben-Ari

Department of Science Teaching, Weizmann Institute of Science

Pauli Byckling, Petri Gerdt, Yevgeniya Kulikova

Department of Computer Science, University of Joensuu

September 28, 2005

Abstract

Roles can be assigned to occurrences of variables in programs according to a small number of stereotypical patterns of use. Studies on explicitly teaching roles to novices learning programming have shown that roles are an excellent pedagogical tool for clarifying the structure and meaning of programs and that their use improves students' programming skills. This paper describes how roles can be applied in various programming paradigms and presents the results of three studies designed to test the understandability and acceptability of the role concept and of the individual roles in procedural, object-oriented and functional programming. Based on the results, two new roles and small modifications to the definitions of the original roles are suggested.

*Corresponding author: P.O.Box 111, Joensuu FI-80101 Finland, Jorma.Sajaniemi@Joensuu.Fi,
Tel: + 358 13 2517933, Fax: + 358 13 2517955.

1 Introduction

There is a tendency for the teaching of programming to degenerate into teaching a sequence of unrelated ideas. Perhaps we start with expressions and then assignment statements, and continue with control statements, or perhaps we teach “objects first” and start with objects, classes, methods and constructors. While we claim that our overall goal is to teach problem solving using a computer and a programming language, all too often learning gets bogged down into the minutiae of the syntax, semantics and pragmatics of writing a program.

In object-oriented programming, the class supplies a unifying concept that enables the student to make sense of the structure of a program without immersing himself in the details. The popularity of the UML notation—even in an educational setting (Kölling et al., 2003; Schulte et al., 2003)—testifies to the advantages of giving this concept a central place in learning programming. Similarly, functional programming is based upon the unifying concept of a side-effect free mathematical function. The simple syntax and semantics that result make functional programming a popular paradigm for teaching introductory programming (Felleisen et al., 2004).

The problem with these concepts (class, mathematical function) is that they concern the structure of a program and hence are static. The concept of *roles of variables* discussed in this paper is proposed as a unifying concept for studying the *dynamic* aspects of a program: the sequence of values taken on by the variables. A further advantage of the use of this concept is its focus on data, rather than on executable statements. Understanding a program requires above all an understanding of its variables; the statements then become a means to manipulate the values of the variables.

Roles (Sajaniemi, 2002, 2005) are a classification of stereotypical behaviors of variables that occur repeatedly in programs. For example, the role *stepper* is a generalization of counters, and covers variables that step through a systematic and predictable series of values. Roles are part of expert programmers’ mental representations (Sajaniemi and Navarro Prieto, 2005) and their number is so small that they can be studied in elementary programming courses. In a classroom experiment, roles were found to give students a new vocabulary, a better way to understand programs, and improved programming skills (Sajaniemi and Kuittinen, 2005; Byckling and Sajaniemi, 2005).

An investigation into professional programmers’ knowledge about variables revealed that there are individual differences in classifying behavior of variables (Sajaniemi and Navarro Prieto, 2005). Roles are not absolute in the sense that all expert programmers would recognize the same set of roles with the same set of distinguishing definitions. The use of roles in teaching, however, requires that

there be a basic set of roles that everybody can agree on—even though individual interpretations are allowed. In order to discover a set of roles suitable for teaching, we have conducted a series of studies in three different programming paradigms: procedural, object-oriented, and functional programming. In each paradigm, we have first looked at several elementary programming textbooks in order to see what individual roles are needed to classify variables occurring in the books. We have validated the suitability of the identified roles by surveys among computer science educators. The rationale behind this approach is that if educators do not find the role concept intuitive and easy to apply, it would be unrealistic to expect them to use roles in teaching.

Conducting studies in three paradigms instead of one, provides two advantages. First, the dynamics of program execution are much the same in any paradigm, and if roles are to be a unifying concept, the differences among the paradigms should not be too great. The resulting role set should thus be coherent and applicable in teaching programming independently of the paradigm used. Second, the differences in roles that are uncovered point to real differences between paradigms and can contribute to a better understanding of the effect of choosing a paradigm to teach programming, as well as to a better understanding of the pedagogy of each individual paradigm.

The paper is structured as follows. Section 2 introduces the role concept and its application to various programming paradigms, as well as the first role set used as a starting point for the studies. Section 3 describes the methodology of the individual studies and the results that were obtained. Section 4 contains the discussion of the results and suggests a revised role set to be used for novice level programming in all the three paradigms. It also discusses differences of paradigms as identified in the studies. Finally, Section 5 contains the conclusion.

2 The Role Concept

Variables are not used in programs in a random or *ad-hoc* way; instead, there are several standard patterns of use that occur repeatedly. In programming textbooks, two patterns are typically described: the counter and the temporary. Sajaniemi (2002) has generalized this idea to the concept of the *roles of variables*, which he obtained as a result of a search for a comprehensive, yet compact, set of characterizations of variables for the purposes of teaching programming and analyzing large-scale programs. His work was based on earlier studies on the use of variables (Ehrlich and Soloway, 1984; Rist, 1989; Green and Cornah, 1985).

In Sajaniemi’s approach, the *role* of a variable characterizes the dynamic nature—the behavior—of a variable: the sequence of its successive values as related to other

variables and to external events. The way the value of a variable is used has no effect on the role, e.g., a variable whose value does not change is considered to be a *fixed value* whether it is used to limit the number of rounds in a loop or as a divisor in a single assignment. Furthermore, as roles describe behavior, they are related to the *deep structure* (Détienne, 2002) of programs, i.e., the logical connections between program constructs. The *surface structure*, e.g., the form of assignment used to update a variable, is much less relevant to the concept of roles.

The original role set suggested by Sajaniemi (2002) applied to both variables and value parameters within procedures and functions. It consisted of nine roles but was later extended by Ben-Ari and Sajaniemi (2004) with one new role, *transformation*, that covered part of the variables that were earlier considered to be *most-recent holders* or *fixed values*. The resulting roles are given in Table 1 together with their informal definitions; exact definitions can be found in the Roles of Variables Home Page (Sajaniemi, 2005).

As an example of assigning roles, consider the Pascal program in Figure 1, which contains three variables: `data`, `count`, and `value`. In the first loop, the user is requested to enter the number of values to be later processed in the second loop. The number is requested repeatedly until the user gives a positive value, and the variable `data` is used to store the last input read. The variable `value` is used similarly in the second loop: it stores the last input. There is no possibility for the programmer to guess what value the user will enter next. Since these variables always hold the latest in a sequence of unrelated values, their role is that of *most-recent holders*. The variable `count`, however, behaves very differently. Unlike the other variables for which there is no known relation between the successive values, once the variable `count` has been initialized, its future values are known. The role of this variable is that of a *stepper*.

In object-oriented programming, roles can be assigned not only to parameters and variables inside methods but also to attributes in classes. For example, in the Java class `Student` of Figure 2, the attribute `student_id` is a *fixed value*, and the attribute `total_credits` is a *gatherer*. Moreover, some objects can also have a variable-like behavior. In Java, for example, an object of type `String` or `Integer` encapsulates a single value, and the whole object behaves like a variable of a primitive type. In such cases, the object is considered to have the role of its only attribute.

In functional programming there are no variables. However, parameters as well as the return values of functions have a role-like behavior over recursive calls. For example, consider the function `max` in Figure 3 that finds the maximum in a list of values. During recursive calls, the parameter `a` is always the largest value found so far (a *most-wanted holder*), and the parameter `h` is the current value (a *most-recent holder*). Thus the set of entities that have roles is different in different

```

program doubles;
    var data, count, value: integer;
begin
    repeat
        write('Enter count: '); readln(data)
    until data > 0;
    count := data;
    while count > 0 do begin
        write('Enter value: '); readln(value);
        writeln('Two times ', value, ' is ', 2*value);
        count := count - 1
    end
end.

```

Figure 1: A simple procedural program.

```

class Student {
    int student_id;
    double total_credits;
    Vector course_list;

    public Student (int s_id) {
        student_id = s_id;
        total_credits = 0;
        course_list = new Vector();
    }

    public void passCourse (Course c) {
        total_credits += c.getCredits();
        course_list.add(c);
    }
}

```

Figure 2: A simple class in object-oriented programming.

Table 1: Roles of variables in novice-level procedural programming.

| Role | Informal description |
|--------------------|--|
| Fixed value | A variable initialized without any calculation and not changed thereafter. |
| Stepper | A variable stepping through a systematic, predictable succession of values. |
| Most-recent holder | A variable holding the latest value encountered in going through a succession of values, or simply the latest value obtained as input. |
| Most-wanted holder | A variable holding the best or otherwise most appropriate value encountered so far. |
| Gatherer | A variable accumulating the effect of individual values. |
| Follower | A variable that gets its new value always from the old value of some other variable. |
| Transformation | A variable that always gets its new value with the same calculation from value(s) of other variable(s). |
| One-way flag | A two-valued variable that cannot get its initial value once its value has been changed. |
| Temporary | A variable holding some value for a very short time only. |
| Organizer | An array used for rearranging its elements. |

programming paradigms: variables in procedural programming; parameters and return values in functional programming; variables, attributes and (some) objects in object-oriented programming.

Even though roles have technical definitions, they are a cognitive concept. As a result people may disagree on a role. The definitions are not entirely mutually exclusive and in specific instances different people may stress different aspects of the definitions. Sajaniemi and Navarro Prieto (2005) have identified two sources of variation in expert programmers' judgment of role-like information: what behavior do programmers perceive from the lifetime of a variable, and what behaviors are considered to be similar.

The behavior of a variable may be perceived differently by two persons even

```

fun max(a, nil)      = a
|   max(a, (h::t))  = if h>a then max(h,t)
                        else max(a,t)

```

Figure 3: Finding maximum in functional programming.

though they look at the same variable, at the same operations on the variable, and at the same value sequence. As a result, they will perceive a different role for the same variable. An example of this type of variation is a variable that takes on the values of the Fibonacci sequence by adding up pairs of previous values in the sequence. A mathematician can predict the sequence as clearly as a novice can predict the sequence of values of the index of a simple for-loop, so she may assign the role of *stepper*, because the values “can be predicted as soon as the succession starts.” On the other hand, a novice who has never seen the Fibonacci sequence before may assign the role of *gatherer*, because the variable accumulates the previous values.

As an example of the second source of variation consider repeated addition by a constant on one hand, and repeated division by a constant on the other hand. In the investigation reported by Sajaniemi and Navarro Prieto, some experts considered them to be similar behaviors (and thus sorted them together in a group corresponding to the role *stepper*), while others considered them to be two different behaviors, and still others were unsure about their similarity. This variation is manifested in vague role boundaries and in differences in the granularity of the roles.

Variable roles are programming knowledge that has traditionally been tacit, but it can be explicitly taught to students (Kuittinen and Sajaniemi, 2004). Roles provide teachers and students with a new vocabulary to talk about programs and provide a new way of seeing connections between statements and expressions dispersed around a program. In a classroom experiment comparing traditional teaching with teaching that used roles and role-based animation, the introduction of roles improved considerably students’ program comprehension and program writing skills (Byckling and Sajaniemi, 2005; Sajaniemi and Kuittinen, 2005). Even in those cases where the assignment of a role is controversial, the debate itself can be an excellent pedagogical tool for clarifying the structure of programs in introductory courses. It is important to emphasize that we do not regard roles as an end in themselves and we do not think that students should be graded on their ability to assign roles. Instead, roles of variables are design rules and pedagogical aids intended to help novices over the hurdle of learning programming.

Potential uses of the role theory are not limited to computer science education. For example, professional programmers spend lots of time in comprehending program code written by others. Automatic detection of roles could be used to provide

meaningful information that would make the comprehension task easier.

3 Studies on Roles in Various Paradigms

We have conducted investigations of roles in three different programming paradigms: procedural, object-oriented, and functional programming. All these studies have followed the same overall strategy. First we have looked at elementary programming textbooks in order to see what individual roles are needed in order to explain the behaviors of variables in novice-level programming in the paradigm. Next we have validated the suitability of the roles that were identified by a web-based studies about computer science educators' attitudes to the role concept and their ability to identify individual roles. This section summarizes the results of these studies. Some of this research has been reported previously (Sajaniemi, 2002; Ben-Ari and Sajaniemi, 2004; Kulikova, 2005; Byckling et al., 2005).

3.1 Role Usage in Textbooks

To reveal the roles needed in novice-level programming, we have analyzed all variables used in meaningful contexts in several introductory programming textbooks. For procedural programming we selected three introductory Pascal textbooks (Foley, 1991; Jones, 1982; Sajaniemi and Karjalainen, 1985) and looked at all whole programs in the books; for object-oriented programming all programs in two introductory Java textbooks (Peltomäki and Malmirae, 1999; Wikla, 2003) were studied; for functional programming all functions in four introductory ML textbooks (Hansen and Rischel, 1999; Michaelson, 1995; Paulson, 1996; Ullman, 1998) were selected for the analysis.

In the case of procedural programming, the role set was created by one researcher who went through all the programs and created a classification of roles. After this phase, he wrote a short description for each role, and then another researcher made an independent analysis of all the variables. The few cases of different classifications were discussed and the role descriptions were adjusted slightly but there were no problems in reaching mutual understanding. For both object-oriented and functional programming, the classification of variables started with the procedural programming role list and new roles were introduced only if needed. In object-oriented programming, two researchers assigned independently roles to all variables in the first textbook. They then discussed variables and their roles one by one, recorded differences in role assignments and resolved contradicting interpretations. The same procedure was then repeated with the other textbook. In functional programming, a single researcher made the classification.

Table 2: Proportion of roles in programming textbooks (%).

| Role | Procedural | Object-oriented | Functional |
|--------------------|------------|-----------------|------------|
| Fixed value | 18–40 | 44–67 | 27–40 |
| Stepper | 21–37 | 14–21 | 10–17 |
| Most-recent holder | 14–16 | 3–9 | 16–22 |
| Transformation | 10–16 | 8–15 | 13–18 |
| Gatherer | 5–7 | 1–2 | 10–15 |
| Temporary | 1–5 | 0–1 | 0–0 |
| Organizer | 1–5 | 0–1 | 1–3 |
| Most-wanted holder | 1–3 | 0–1 | 0–1 |
| Follower | 0–3 | 1–1 | 0–1 |
| One-way flag | 0–2 | 3–3 | 0–1 |
| Modifier | – | – | 2–5 |
| Selector | – | – | 2–5 |
| Other | 0–2 | 1–4 | 0–0 |

Table 2 gives the frequencies of roles in the textbooks. For each paradigm the lowest and highest percentage in individual textbooks are listed. There are also two special roles—*modifier* and *selector*—that were identified in the functional programming study. These roles are described in Table 3.

Table 3: Special roles used in the functional programming study.

| Role | Informal description |
|----------|---|
| Modifier | A list or a tree whose elements are added and removed. |
| Selector | A variable initialized by several alternative expressions and not changed thereafter. |

The overall distribution of role occurrences is similar in all three paradigms. There are, however, some differences that can be explained by the nature of programs in the different paradigms. The high number of *fixed values* in object-oriented programming is due to the number of parameters that simply pass data from one object to another. Forty-three percent of all *fixed values* in object-oriented programming were of this type reflecting a distinctive difference in the

use of parameters when compared with procedural and even functional programming.

The high number of *steppers* in procedural programming is due to the use of loops—requiring *steppers* for loop control—for both array manipulation and repeated actions. In object-oriented programming, loop control variables are often encapsulated within iterators, which reduces the need for *steppers*. In functional programming, repetition is achieved by recursion and is used mainly for data structure manipulation.

There are notable differences in the tasks carried out by programs in the textbooks. Procedural programs deal mostly with user-supplied input data and apply an algorithm to compute meaningful results. Programs in object-oriented textbooks tend to model static data relationships (yielding a large number of *fixed values*), or demonstrate graphics capabilities of the language (resulting in less input and hence fewer *most-recent holders*). They are not that interested in computing meaningful results of algorithms; this is manifested in relatively small number of *gatherers* and *most-wanted holders*. The large number of *one-way flags* in object-oriented programming is related to stop flags needed in animations and for tracking simple user actions. Finally, functional programming examples do lots of list manipulation yielding a large number of *gatherers*, for example, to build a new list.

No *temporaries* were detected in the functional programming textbooks. A similar effect can be found in other paradigms if a temporary variable is declared locally in the block that needs it. Theoretically this makes the variable either a *fixed value* or a *transformation*. Thus the frequencies of these three roles do not depend on the paradigm only but also on the features of programming languages, i.e., in what places can local variables be declared.

In the analysis of functional programming textbooks two new roles were introduced: *modifier*, which is a data structure that allows modifications, and *selector*, which is in essence a *fixed value* with alternative expressions, one of which is selected. Data structure manipulation is not covered in typical elementary courses of procedural programming and hence it is not reflected in the original set of roles. In addition to functional programming, data structures are treated early in teaching object-oriented programming. However, no new roles were introduced in the object-oriented study even though in some cases there were problems in finding a proper role among the existing role set and role definitions had to be interpreted liberally. We will treat this question in more detail in Section 4.

3.2 Role Assimilation by Computer Science Educators

To study computer science educators' attitude towards the role concept and individual roles and their ability to learn the roles, we conducted three web based studies,

one for each paradigm. All three studies used the same research methodology. The research materials consisted of web pages divided into three phases. The *tutorial phase* introduced the concept of roles of variables, followed by a section for each role containing: the definition of the role (as given in Table 1), a full sample program demonstrating the role, additional examples of the use of the role, and a list of additional properties that can assist in identifying the role.

Following the tutorial, participants were presented with a *training phase* consisting of a sequence of six programs or functions and their task was to assign roles to variables using radio buttons; to reduce the demands on short-term memory and to ensure accuracy in the use of the roles, each button label was linked to the corresponding role definition. After each program or function, the participant was given feedback on his or her assignment of a role to each variable.

The final *analysis phase* was similar in format to the training phase but now all the programs or functions were in a single page. Upon assigning all of the roles, the results were automatically sent by email to the researchers. In the procedural case, the participants were given six small programs with 24 variables, and in the functional case six functions with 34 variables. In order to use object-oriented programming features, one larger program was used in the object-oriented study where six code fragments and 23 variables were selected for the analysis phase. The code fragments were longer than in the other two studies, and in order to make the comprehension task easier, occurrences of a variable were highlighted when a participant clicked on its name in the task frame.

In all cases, participants were given the option to suggest a new role by themselves or to indicate that they did not know which role to select. They also had the opportunity to append comments to their choices and to comment on the role concept itself. Finally, they were asked to indicate their length of experience teaching introductory programming and/or advanced CS courses in high school and/or college or university.

In order to simplify and shorten the tasks of participants, only six roles were included in the materials. In procedural and object-oriented programming the roles *one-way flag*, *temporary* and *organizer* (accounting respectively for only 5% or 4% of all variables in the textbook analyses) were excluded. In functional programming *follower* and *most-wanted holder* were also excluded, but the new roles *selector* and *modifier* were included. The final materials can be found in Sajaniemi (2005).

Participants were recruited by publicizing the URL containing the research material on several mailing lists and by sending email personally to programming teachers whose addresses were collected from Internet. The number of valid participants was 51 in procedural programming, 43 in object-oriented programming, and 26 in functional programming. They worked remotely on the web-based material

Table 4: Proportion of correct assignments of roles by CS educators (%).

| Role | Procedural | Object-oriented | Functional |
|--------------------|------------|-----------------|------------|
| Fixed value | 75–100 | 51–91 | 89–100 |
| Stepper | 49–100 | 37–100 | 78–100 |
| Most-recent holder | 92–92 | 42–93 | 85–100 |
| Transformation | 63–90 | 49–84 | 89–93 |
| Gatherer | 25–94 | 49–70 | 63–85 |
| Most-wanted holder | 55–92 | 84–84 | – |
| Follower | 96–96 | 26–88 | – |
| Selector | – | – | 52–89 |
| Modifier | – | – | 74–89 |

at their own pace; based on the pretests of materials the time need was estimated to vary from 30 to 90 minutes.

On average, participants assigned roles correctly in 85% cases in procedural, 62% in object-oriented, and 88% in functional programming study. As noted in Section 2, in some cases different people may perceive the behavior of a variable differently and consequently assign a different role. If such “controversial” cases are excluded, the proportions of correct role assignments are 93%, 83%, and 91% respectively. Table 4 gives the proportions of correct assignments in each study. For each role the lowest and highest percentage for individual variables are listed. The lowest numbers correspond most often to controversial variables whereas the highest numbers are related to typical uses of the roles.

In the object-oriented programming study the accuracy of assigning roles to variables was much lower than in the other studies. It should be noted, however, that the materials in this study differed from the other studies. First, the program was much longer and more complicated; variables were updated with assignments dispersed among the classes; and control flow was harder to comprehend because of nested method invocations and a smaller indentation step. These choices were made in order to be more faithful to the object-oriented paradigm, but they made comprehension of the behavior of variables harder. Second, the materials contained more controversial cases because the acceptability of normal cases had already been validated by the other studies.

In all the studies, the participants’ comments on the role concept in general were mostly positive. They believed that roles could contribute to understanding programs. Some participants hoped for a better integration with the current concepts of the appropriate paradigm: data flow and invariants in procedural pro-

programming, and patterns of stereotypical class relationships in object-oriented programming.

3.3 Difficulties with Specific Roles

In all three studies participants were divided *post hoc* into three groups: high, mediocre, and low performers. We were especially interested in errors made by high performers on non-controversial variables because they are potential indicators of problems in the entire role concept or the definitions of individual roles.

In the procedural case, the only role that caused frequent confusion was *transformation*. It was intended to identify cases where a variable has no independent existence, but merely serves to contain a value obtained by computation. In a sense, this role “usurps” the role or roles assigned to the variables from which the transformation is computed.

In the materials for the procedural study a *one-way flag* was deliberately used even though this role was not described in the tutorial. High performers supplied the largest number of alternative suggestions for other roles providing evidence that the *one-way flag* is a distinct role, not subsumed by or similar to the others.

In the object-oriented study, the *transformation* role caused confusion just like in the procedural case. Another problem was presented by the left index in a binary search that resulted in the most even distribution of suggestions over all roles among high performers. As the behavior of high performers was similar to the case of *one-way flag* above, the problem with the index variable suggests a potential problem in assigning roles to variables used in array traversals. Moreover, high performers’ reluctance to recognize variables stepping through the elements of linked lists as *steppers* indicated a similar problem with data structures other than arrays.

Even though a role describes the behavior of a variable in an individual object, the set of all instances of an attribute seemed to affect role assignment. For example, a *fixed value* was sometimes confused with a *most-recent holder*, even though the value of the attribute did not change within a single object. In this case, new instances of the attribute in new objects may have given the impression of a value succession. This error type is specific to object-oriented programming but its roots are similar to a problem in the procedural case where the behavior of arrays was sometimes unnecessarily separated from the behavior of its elements.

In the functional case, the *transformation* role caused again confusion. Moreover, high performers sometimes mixed the *gatherer* role with the newly introduced *modifier*. In these cases the surface structure was unable to reveal the difference; participants had to understand the total effects of the computation—the deep structure—in order to make the correct decision.

4 Discussion

The introduction of roles of variables in elementary programming courses has been found to facilitate learning programming, provide a new conceptual framework to think about programs, and improve novices' programming skills (Sajaniemi and Kuittinen, 2005; Byckling and Sajaniemi, 2005). The purpose of the current study was to find out what roles are needed in various programming paradigms, how much support do computer science educators need to assimilate the roles, and how they do react to the whole idea. The results indicate that the roles are largely paradigm independent: the same roles apply in procedural, object-oriented and functional programming. Moreover, computer science educators accepted the concept of roles as intuitive and found it easy to assign roles consistently. Roles were identified well in typical cases of use after an hours' introduction consisting of reading a short tutorial and doing a few exercises in assigning roles.

4.1 Revision of the Role Set

Most disagreements among the participants occurred with controversial variables that are variables having either an atypical surface structure or an atypical deep structure. The ability to recognize and go beyond an atypical surface structure is gained by increased expertise and developing this ability is the task of the teacher. On the other hand, atypical deep structure is a sign that a new role might be needed. This claim is justified by the similarity in the participants' approach to atypical deep structures and to the missing role. Since we want to keep the number of roles small so that they can be used in introductory teaching, we prefer that variable behaviors that rarely occur should be embedded within the existing roles.

The original role set does not cover variables needed in data structure manipulation. Elementary procedural programming courses do not introduce data structures other than arrays and consequently variables needed in linked data structures were not encountered in the analysis of the first textbooks. Later studies revealed that data structures are treated more thoroughly both in elementary object-oriented textbooks and in elementary functional programming textbooks. Data structure traversals require a new role, *walker*, which is a variable that moves from element to element so that its new value depends on its old value, traversal strategy, and possibly some search parameters. Typical examples are a pointer to the current element in depth-first search, the index of the middle element in binary search, and a pointer to the last element of a queue where elements are added to the end of the list. Moreover, the definition of *fixed value* must be changed to allow setting the value to NULL so that a link may remain *fixed value* even if the element pointed to is destroyed.

In the original role set the role of an array is the role of its elements. For example, an array is *gatherer* if it contains 12 *gatherers* that calculate the total sales of each month from daily sales given as input. Moreover, there was a special role for arrays, *organizer*. These ideas apply also to other data structures. Furthermore, a new role, *container*, is needed for data structures that store data that can be added and removed. This new role covers the role of *modifier* introduced in the functional programming study.

The most problematic role in the studies was *transformation*: it was not easily recognized and it was often suggested in place of other roles. Of course, many variables are the result of computation from other variables, but *transformation* was intended to be used in specific cases where the value has no separate existence of its own but is stored in a variable, e.g., because the result of some computation is needed several times in the code. A new look at the role definitions revealed that most *transformations* are actually *temporaries*—the result is needed only for a short time. If the result has a longer existence, then the variable may be a *most-recent holder* and the variable holding the original value is then *temporary*—if its value is not needed afterwards—or they both can be *most-recent holders*. Therefore, the role of *transformation* is redundant and can be dropped.

Table 5 lists the resulting revised role set that applies to procedural, object-oriented and functional novice-level programming. In order to get a paradigm independent terminology, the term *data item* is used instead of variable. A data item is a variable or value parameter in procedural programming; a variable, attribute, value parameter or single attribute object in object-oriented programming; and the recursive behavior of a parameter or return value in functional programming.

4.2 Paradigm-specific Considerations

In large programs it can be hard to extract the behavior of a variable when the lines affecting its value are dispersed. In object-oriented programming, data flow is decentralized and all lines affecting a variable are not necessarily in the same method—in fact they can be in different classes—and control flow is harder to discern than in procedural programming (Corritore and Wiedenbeck, 1999). This does not mean, however, that roles are less important in object-oriented programming. On the contrary, explicit role information in the form of comments written by the author of a program might help expert programmers in program comprehension.

In functional programming, values of parameters cannot be changed within a function. In procedural and object-oriented programming this is possible and therefore recursion adds a second dimension in the lifetime of a parameter: on one hand the parameter has some role within the function, while on the other hand it has a role describing its behavior through recursive calls. If the parameter is not

Table 5: The revised role set for novice-level programming.

| Role | Informal description |
|--------------------|---|
| Fixed value | A data item that does not get a new proper value after its initialization. |
| Stepper | A data item stepping through a systematic, predictable succession of values. |
| Walker | A data item traversing in a data structure. |
| Most-recent holder | A data item holding the latest value encountered in going through a succession of unpredictable values, or simply the latest value obtained as input. |
| Most-wanted holder | A data item holding the best or otherwise most appropriate value encountered so far. |
| Gatherer | A data item accumulating the effect of individual values. |
| Follower | A data item that gets its new value always from the old value of some other data item. |
| One-way flag | A two-valued data item that cannot get its initial value once the value has been changed. |
| Temporary | A data item holding some value for a very short time only. |
| Organizer | A data structure storing elements that can be rearranged. |
| Container | A data structure storing elements that can be added and removed. |

changed within the function, its internal role is *fixed value* and its behavior over recursion will usually be more interesting. We have seen cases where a parameter is assigned new values within a function resulting in internal behavior different from recursive behavior and thus giving two roles to the parameter. These functions have proved to be hard to understand, and it seems to be a bad habit to overload a parameter with several roles.

Some participants pointed out in their comments that they had problems in embedding the role concept into object-oriented thinking. This may be due to the fact that expert object-oriented programmers are used to analyze programs in terms of patterns that represent stereotypical class hierarchies and work at a higher abstrac-

tion level than variables. On the other hand, many participants reported that the role concept was natural and provided a new way of thinking about variables. Thus the role concept seems to fit object-oriented programming and problems in relating roles to patterns should be rare if roles are introduced in the first programming courses. Some participants also complained about “non-standard” terminology: in object-oriented programming “iterator” is the common name for many *walkers* and in functional programming “accumulator” is used for *gatherer*. We hope that a common role terminology could unify terminology in different paradigms.

The studies also revealed major differences in the programming problem types used in different programming paradigms. Procedural programming textbooks stress functionality: programs compute new values and interact with users with simple interfaces that provide users with meaningful results. Object-oriented textbooks tend to stress data modeling on one hand and language features on the other hand. Message passing structures may be complex but their final effects may be simple actions and computations that are trivial from the perspective of the application area. Finally, functional programming textbooks are interested in data manipulation techniques. These differences could also be identified in the role distributions in the textbooks that were studied. Thus the selection of a programming paradigm does not only affect the way programs are seen to work, but also what tasks are seen as important.

5 Conclusion

Programming is a difficult skill to learn and methods to improve teaching are needed. The concept of roles of variables can be used as a pedagogical technique to teach how the constructs of a programming language work together to implement the solution of a problem. Results of using roles in teaching elementary programming indicate that the introduction of roles improves program comprehension and program writing skills. In this paper, we were interested to find out a set of roles that computer science educators could accept as intuitive. Moreover, we wanted to see whether the programming paradigm—procedural, object-oriented, or functional—has an effect on the acceptability of the roles.

The outcome of the studies is encouraging because CS educators accepted the concept of roles as intuitive and found it easy to assign roles consistently. The early introduction of data structures in object-oriented and functional programming led to small changes in one role definition and to the introduction of two new roles into the role set originally developed for procedural programming. Moreover, one role was abolished because its overlap with other roles caused confusion among educators. The resulting new role set (Table 5) consists of eleven roles—two of which

are specific to data structures, while the rest apply mainly to individual data items. This set is sufficient to cover practically all variables in elementary programming and it can be used for teaching in all three programming paradigms.

The suitability of the same roles in various programming paradigms supports the view that roles are a unifying concept for studying the dynamic aspects of programs. Its focus on data behavior reveals the ultimate purpose of programs—to provide meaningful information to human users—and separates it from control issues which vary from one paradigm to another and whose main purpose is to direct the execution of statements that make the individual data manipulation steps.

Research on the role concept has so far proved evidence that roles enhance learning elementary procedural programming. Moreover, role-based program animation has been found to elaborate role knowledge so that students can apply it fluently in programming. In future we plan to study in more detail these phenomena in both procedural and object-oriented programming.

Acknowledgments

This work was supported by the Academy of Finland under grant number 206574.

References

- Ben-Ari, M. and Sajaniemi, J. (2004). Roles of variables as seen by CS educators. In *Proceedings of the Ninth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'04)*, pages 52–56. ACM Press.
- Byckling, P., Gerdt, P., and Sajaniemi, J. (2005). Roles of variables in object-oriented programming. Accepted to the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2005) Educators Symposium, San Diego, California, USA, October 2005.
- Byckling, P. and Sajaniemi, J. (2005). Using roles of variables in teaching: Effects on program construction. In Romero, P., Good, J., Bryant, S., and Chaparro, E. A., editors, *Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2005)*, pages 278–303. University of Sussex, U.K.
- Corritore, C. and Wiedenbeck, S. (1999). Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *International Journal of Human-Computer Studies*, 50:61–83.

- Détienne, F. (2002). *Software Design—Cognitive Aspects*. Springer-Verlag.
- Ehrlich, K. and Soloway, E. (1984). An empirical investigation of the tacit plan knowledge in programming. In Thomas, J. C. and Schneider, M. L., editors, *Human Factors in Computer Systems*, pages 113–133, Norwood, NJ. Ablex Publishing Company.
- Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S. (2004). The TeachScheme! project: Computing and programming for every student. *Computer Science Education*, 14(1):55–77.
- Foley, R. W. (1991). *Introduction to Programming Principles Using Turbo Pascal*. Chapman&Hall.
- Green, T. R. G. and Cornah, A. J. (1985). The programmer’s torch. In *Human-Computer Interaction - INTERACT’84*, pages 397–402. IFIP, Elsevier Science Publishers (North-Holland).
- Hansen, M. R. and Rischel, H. (1999). *Introduction to Programming Using SML*. Addison-Wesley.
- Jones, W. B. (1982). *Programming Concepts – A Second Course*. Prentice-Hall.
- Kölling, M., Quig, B., Patterson, A., and Rosenberg, J. (2003). The BlueJ system and its pedagogy. *Computer Science Education*, 13(4):249–268.
- Kuittinen, M. and Sajaniemi, J. (2004). Teaching roles of variables in elementary programming courses. In *Proceedings of the Ninth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE’04)*, pages 57–61. ACM Press.
- Kulikova, Y. (2005). Roles of variables in functional programming. Unpublished Master’s Thesis, Department of Computer Science, University of Joensuu, Finland.
- Michaelson, G. (1995). *Elementary Standard ML*. UCL Press.
- Paulson, L. C. (1996). *ML for the Working Programmer*. Cambridge University Press, 2nd edition.
- Peltomäki, J. and Malmirae, P. (1999). *Java-ohjelmoinnin peruskirja (Basic Book in Java Programming)*. Teknolit Oy.
- 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. In *Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, pages 37–39. IEEE Computer Society.
- Sajaniemi, J. (2005). Roles of variables home page. http://www.cs.joensuu.fi/~saja/var_roles/. (Accessed June 10th, 2005).
- Sajaniemi, J. and Karjalainen, M. (1985). *Suppea johdatus Pascal-ohjelmointiin (A Brief Introduction to Programming in Pascal)*. Epsilon ry, Joensuu, Finland.
- Sajaniemi, J. and Kuittinen, M. (2005). An experiment on using roles of variables in teaching introductory programming. *Computer Science Education*, 15:59–82.
- Sajaniemi, J. and Navarro Prieto, R. (2005). Roles of variables in experts' programming knowledge. In Romero, P., Good, J., Bryant, S., and Chaparro, E. A., editors, *Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2005)*, pages 145–159. University of Sussex, U.K.
- Schulte, C., Magenheimer, J., Niere, J., and Schäfer, W. (2003). Thinking in objects and their collaboration: Introducing object-oriented technology. *Computer Science Education*, 13(4):269–288.
- Ullman, J. (1998). *Elements of ML Programming*. Prentice-Hall.
- Wikla, A. (2003). *Ohjelmoinnin perusteet Java-kielellä (Basics of Programming in Java)*. OtaDATA.