On the Concept of Variable Roles and its Use in Software Analysis

Yulia Demyanova, Helmut Veith, Florian Zuleger
Vienna University of Technology

Abstract—Human written source code in imperative programming languages exhibits typical patterns for variable use, such as flags, loop iterators, counters, indices, bitvectors, etc. Although it is widely understood by practitioners that these patterns are important for automated software analysis tools, they are not systematically studied by the formal methods community, and not well documented in the research literature. In this paper, we introduce the notion of variable roles on the example of basic types (int, float, char) in C. We propose a classification of the variables in a program by variable roles which formalises the typical usage patterns of variables. We show that classical data flow analysis lends itself naturally both as a specification formalism and an analysis paradigm for this classification problem. We demonstrate the practical applicability of our method by predicting membership of source files to the different categories of the software verification competition SVCOMP 2013.

I. INTRODUCTION

Programs written in imperative programming languages, such as C, Java, Perl, Python, share typical patterns of variable use, like flags, loop iterators, counters, indices, bitvectors, temporary variables, and so on. Experienced programmers have informal knowledge of these patterns, to which we refer as variable roles. For example, from the piece of C code

```
while (i<n) a[i++]=0;
```

it can be deduced that i is a loop iterator and an array index. Similarly, from the statement

```
x&=y
```

we can infer that x is a bitvector.

In common programming languages, there is no direct mapping from data types to roles - multiple roles can be associated with the same type. For example, in C, the type int can be used to store such different values as boolean, file descriptor, bitvector, and character literal. Moreover, it is not clear how to extend standard type systems for languages like C to express roles like array index, counter, and loop iterator. Additionally, one variable can have several roles simultaneously, like the variable i in the loop example above. In type systems, in contrast, one variable must be assigned one and only one type. Therefore, roles can not be considered simply as refined types.

Information about variable roles is implicitly contained in the structure of the source code, thus the roles can often be inferred by syntactic analysis. This can be done by analysing the expressions or statements of a given kind, for example, matching array indices in array subscripts. Alternatively, roles can be inferred by searching for code patterns, for example, $t=x; x=y; y=t;$ is a typical pattern for a temporary variable t.

The notion of a variable role has two dimensions. In general, variable roles represent heuristics, which means that they can be systematically studied and analysed, but they need to be treated as auxiliary heuristic information. Thus, variable roles can guide a verification tool, but the soundness of a formal analysis must not depend on variable roles. Certain variable roles, however, provide sound information, which can be relied upon during verification, and thus these roles can be viewed as types. We will explore these two dimensions of variable roles in future work.

In this paper we define 14 variable roles with a standard data-flow analysis. Our definition serves at the same time as an algorithm to compute the roles. In order to choose the roles, we have manually investigated 5.2 KLOC of C code from the cBench benchmark [1]. We assigned roles to the variables of basic types such as int, float and char. When choosing the roles, we were inspired by typical programming patterns for variable use in real life programs. We have chosen the roles in such a way that a small number of roles is able to classify each occurring program variable in the programs we considered. We have implemented a prototype of a tool which maps basic-type variables in C programs to sets of roles.

As this short paper is reporting work in progress, we are currently exploring applications for variable roles. An important natural application is the use of variable roles to create abstractions in software verification or choose abstract domains through a better understanding of the program. For example, in C programs integer variables are used to store boolean flags, because there is no boolean type. When creating an abstraction for a C program, we know that the predicate $x==0$ provides sufficient information about a boolean variable x. However, most state-of-the-art verification papers consider a program as a logical formula and either ignore such implicit information, or treat it as undocumented heuristics. For example, the ASTREE static analyser [2] relies heavily on human insight for selecting the right domain. Variable roles could be used for automating this process, e.g., to suggest the use of octagon or polyhedra domains for variables which occur only in linear operations, BDDs for boolean variables, etc. This will save a verification tool from enumerating all possible domains. After submission of this paper, we learnt about current work in this direction by the developers of the CPAchecker verification tool [3].

Another important application area of our method is to classify source files, for example, from benchmarks for different verification competitions, according to the relative number of occurrences of variable roles in them. To demonstrate...
int x, y, n = 1 0;  
...  
y = 2 x;  
while (x)  
{  
  n++3;  
  x = 4 x & 5 (x-1);  
}  

a) bitvector, counter, iterator  
b) character, file descriptor, linear  

Fig. 1: Different patterns of use of integer variables

We will use the C programs of Figure 1 to informally introduce variable roles, whose formal definitions are given later in the section. In the programs we have assigned labels to the statements and expressions to which we refer from the text.

The program in Figure 1a calculates the number of non-zero bits of the variable x. In every loop iteration, a non-zero bit of x is set to zero and the counter n is incremented. The loop continues until all bits are set to zero. Although the variables x and n are declared of the same type int, they are used differently. For a human reading the program, the statements n = 0 and n++ in the loop body signal that n is a counter. Indeed, n is used to count the number of loop iterations. On the other hand, the value of the variable x as an integer is not used in calculations, but rather individual bits in its binary representation matter.

We define the roles by restricting the operations in which a variable occurs. We require that a bitvector occurs in at least one bitwise operation (bitwise AND, OR or XOR), like the variable x in expression 5. We require that a counter variable only changes its value in an increment or decrement statement or gets assigned zero. The variable n, which is assigned in statements 1 and 3, satisfies these constraints.

The program in Figure 1b reads a decimal number from a text file and stores its numeric representation in the variable val. In contrast, the variables fd and c are used to store the output of the library functions open() and read() respectively. The difference between the two variables is that c is later used in calculations, while fd is only passed to the function read() as a black box because its value does not directly affect the result of the computations. One can conjecture that c is a character, because it is passed as an input to the function isdigit(), which checks whether its parameter is a decimal digit character. Even though isdigit() is declared to take a parameter of type int, the documentation states that the parameter is a character to be tested, cast to int [5].

We define character, file descriptor and linear roles as follows. We require that a character variable is assigned at least once a character literal (e.g., c = ’a’) or another character variable, or is used in a standard C function for manipulating characters (e.g., c = getchar() or isdigit(c)). A file descriptor is required to be used in a standard C function for manipulating files (e.g., fd = open(path, flags) or read(fd, &c, 1)). A linear variable can be assigned only linear combinations of linear variables. The variables c and val, assigned in statements 3 and 2, respectively, satisfy the latter constraint.

II. FORMALISATION OF VARIABLE ROLES

A. Examples

We will use the C programs of Figure 1 to informally introduce variable roles, whose formal definitions are given later in the section. In the programs we have assigned labels to the statements and expressions to which we refer from the text.

The program in Figure 1a calculates the number of non-zero bits of the variable x. In every loop iteration, a non-zero bit of x is set to zero and the counter n is incremented. The loop continues until all bits are set to zero. Although the variables x and n are declared of the same type int, they are used differently. For a human reading the program, the statements n = 0 and n++ in the loop body signal that n is a counter. Indeed, n is used to count the number of loop iterations. On the other hand, the value of the variable x as an integer is not used in calculations, but rather individual bits in its binary representation matter.

We define the roles by restricting the operations in which a variable occurs. We require that a bitvector occurs in at least one bitwise operation (bitwise AND, OR or XOR), like the variable x in expression 5. We require that a counter variable only changes its value in an increment or decrement statement or gets assigned zero. The variable n, which is assigned in statements 1 and 3, satisfies these constraints.

The program in Figure 1b reads a decimal number from a text file and stores its numeric representation in the variable val. In contrast, the variables fd and c are used to store the output of the library functions open() and read() respectively. The difference between the two variables is that c is later used in calculations, while fd is only passed to the function read() as a black box because its value does not directly affect the result of the computations. One can conjecture that c is a character, because it is passed as an input to the function isdigit(), which checks whether its parameter is a decimal digit character. Even though isdigit() is declared to take a parameter of type int, the documentation states that the parameter is a character to be tested, cast to int [5].

We define character, file descriptor and linear roles as follows. We require that a character variable is assigned at least once a character literal (e.g., c = ’a’) or another character variable, or is used in a standard C function for manipulating characters (e.g., c = getchar() or isdigit(c)). A file descriptor is required to be used in a standard C function for manipulating files (e.g., fd = open(path, flags) or read(fd, &c, 1)). A linear variable can be assigned only linear combinations of linear variables. The variables c and val, assigned in statements 3 and 2, respectively, satisfy the latter constraint.

B. Definition of the analysis

We define variable roles using classical intraprocedural dataflow analysis [6]. In this section we use the notation as follows. Var denotes the set of program variables, and Num denotes all scalar constant literals (e.g., 0, 0.5, ’a’). S, E and B denote the set of program statements, arithmetic and boolean expressions respectively. For the elements of these sets we use the same names in the lowercase version (e.g., var for a program variable).

For a program s ∈ S the result of analysis R is computed using the function ResR, which is defined as follows:

\[ Res^R = Init^R \cup gen^R(s), \]
and the analysis yields the result
operation. After that, the result set does not change anymore,
bitwise AND operation. At expression 5, the variable $x$ is also
added to the result set because $x$ occurs in a bitwise
operation. After that, the result set does not change anymore,
and the analysis yields the result \{x\}, as shown in Figure 3a.

## III. IMPLEMENTATION AND EXPERIMENTS

We used the clang compiler [7] to implement a prototype of
a tool, which assigns a subset of variable roles to every basic-
type variable. The current implementation does intraprocedural
analysis and does not include a pointer analysis. We replace
all function calls (e.g., c=getchar()) and pointer derefer-
ences (e.g., n=ptr.n.arr[1]) with fresh variables. For example,
the statement c=getchar() would be rewritten as c=t1, and in the LINEAR analysis the variable t1 would
not be excluded from the result set, but rather assigned the
role “unresolved assignment”, which is a trade-off between
soundness and precision.

We ran two experiments. In the first one we computed the
relative number of the occurrences of each role in every
category. We calculated it by summing up the numbers in all
files of a category and normalising them by the total number of
variables in these files. The results for the categories “Control
The authors use static analysis to extract two types of statements – the sound statements which follow from the requirements of safety, non-redundancy and reachability of the code (e.g., "a pointer is not null") and hypotheses which follow from the statistics of observations (e.g., "calls to functions f() and g() should be paired").

Rondon et al. [12] use predicate abstraction over a fixed set of predicates to infer so called liquid types, i.e. types refined with a conjunction of propositional predicates (e.g., \( x > 0 \land x < 5 \)). We consider this approach to be complementary to ours, because it does not use any information from the source code other than the transition relation, and concentrates on arithmetic properties of variables.

Variable names and comments as an additional source of knowledge about a program have been systematically studied in program comprehension. The Latent Semantic Indexing technique [13] allows to query the program source code using words in natural language, based on the number of occurrences of the words in variable names and comments. A study has been made of the naming rules for variables in real-word programs [14], and of expanding abbreviated identifiers to full words [15]. We plan to use these techniques in future work.

ACKNOWLEDGMENT

This work is supported by the Austrian National Research Network S11403-N23 (RISE) of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) through grants PROSEED and ICT12-059.

REFERENCES
