Science of Computer Programming 74 (2009) 989–1020

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Mathematics for reasoning about loop functions

Ali Mili *, Shir Aharon, Chaitanya Nadkarni

College of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102-1982, United States

ARTICLE INFO

Article history:
Received 2 May 2008
Received in revised form 10 September 2009
Accepted 15 September 2009
Available online 25 September 2009

Keywords:
Functional extraction
Loop function
Sub-goal induction theorem
Mills' theorem

ABSTRACT

The criticality of modern software applications, the pervasiveness of malicious code concerns, the emergence of third-party software development, and the preponderance of program inspection as a quality assurance method all place a great premium on the ability to analyze programs and derive their function in all circumstances of use and all its functional detail. For C-like programming languages, one of the most challenging tasks in this endeavor is the derivation of loop functions. In this paper, we outline the premises of our approach to this problem, present some mathematical results, and discuss how these results can be used as a basis for building an automated tool that derives the function of while loops under some conditions.

1. Introduction and premises

Modern software applications can be characterized by high size and complexity, high levels of criticality, and heightened security concerns. Furthermore, modern software paradigms rely heavily on third-party software artifacts, which preclude process quality controls and shift the burden of verification and validation to product controls. The combination of these premises places a high premium on the ability to analyze program functions to arbitrary levels of thoroughness and precision. Such a capability would have broad applications in many fields of software engineering, such as:

- Support for code inspections.
- Support for reverse engineering of legacy code.
- Support for code analysis: if we know the function of a loop, we can answer questions of the form: *Does this loop refine specification R* (for some relational specification *R*)?

In this paper, we consider the problem of extracting the function of a while loop in a C-like programming language, of the form (`while t {B;}`) where *t* is a total boolean function. The premises that characterize our approach to the derivation of loop functions can be summarized as follows:

- *Closed-form functions*. We acknowledge that the characterization of a closed-form representation is not clear-cut, but we wish to exclude obvious non-closed forms, such as transitive closures, recursive definitions, and existential quantification over the number of iterations. In essence, this means that we must bridge the inductive gap between the function of the loop body (which describes what happens in a single iteration) and the function of the loop (which describes what function the whole loop computes).

* Corresponding author.
E-mail address: mili@cis.njit.edu (A. Mili).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.09.009

- *Deriving the loop function by successive approximations*. As a divide-and-conquer discipline, the loop function is derived progressively, by accumulating information on the loop behavior as more and more features of the loop are analyzed and captured. This is a crucial feature of our approach, as it makes it possible to derive the function of arbitrarily large loops with limited overhead, by analyzing small segments of their source code at a time.
- *Providing substitutes for the loop function*. Our approach provides a continuum of analysis capability, whereby even when we cannot derive the function of a loop in all its detail, we can still make provable statements about its functional properties.
- *A refinement based approach*. The ordering properties and the lattice properties of the refinement ordering are at the core of the divide-and-conquer strategy that we apply. The refinement ordering gives us a framework in which we can cast our arguments and our algorithms.

2. Sample loop functions

In order to help the reader gain a clear idea of our goal, and to reflect the current capability of our proposed approach, we present below a set of four simple loops, along with the functions that our algorithm computes for them. We use C++ syntax to represent the original loop, and we use mathematical (relational) notation to represent the loop function. Though the examples are fairly simple, the algorithm is not limited, in the sense that we can evolve it to deal with a wide range of data types and operations, by merely adding new knowledge (programming knowledge, domain knowledge), as we discuss in the sequel (Section 5). The function of the loop is obtained from the source text (in C++) by a three-step transformation; the intermediate representations of each loop are given in Section 6.

2.1. A numeric example

The first example involves numeric computations. We introduce a number of constants, making this a family of programs (according to the values of constants), rather than a single program (because for some values of the constants the shape of the loop changes). We consider the following C++ program:

```
#include <iostream>
using namespace std;

int x, t, i, v, w, y, z;   //  program variables
                           //  we assume i>=0;
const int a = ..;          //  program constants
const int b = ..;
const int c = ..;
const int d = ..;
const int e = ..;

int main ()
{
while (i != 0)
   { v = v + a*t;
     z = z + c*x;
     w = w + e*y;
     x = x+a;
     y = y+b;
     t = t*d;
     i = i-1;
   }
}
```

We are interested in deriving the function that this loop defines between its initial states (values of x, y, z, t, v, w and i prior to the execution of the loop) and its final states (values of these variables when execution terminates). The function of this loop is actually more complicated than could appear, because of the variety of configurations of the constants in this program (a, b, c, d, e). For example, if constant a is zero, then variables v and x are preserved, and the expression for z becomes a multiplication rather than the sum of an arithmetic series. Likewise, if constant d is equal to 1 then variable t is preserved and the expression for v becomes a multiplication rather than the sum of a geometric series. Our algorithm produces the function of this loop as a union of several terms, one for each possible configuration of the values of the program constants. For the sake of simplicity, we present below a set of three representative terms. The first one reflects the case where all the