Hierarchical analysis of loops with relaxed abstract transformers

Michael Shell[1], Homer Simpson[2], James Kirk[3], Montgomery Scott[3], and Eldon Tyrell[4], *Fellow, IEEE*
[1]School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332 USA
[2]Twentieth Century Fox, Springfield, USA
[3]Starfleet Academy, San Francisco, CA 96678 USA
[4]Tyrell Inc., 123 Replicant Street, Los Angeles, CA 90210 USA

Circulation is an indispensable component of the program's completion of complex functions. Circulation analysis is a difficult problem in the field of program analysis. The main reason is that it is difficult to find a sufficiently accurate loop invariant to describe the behavior of the loop, especially when the loop exists. The situation becomes more complicated when linear invariants are used.

***Index Terms*—Loop analysis, Abstract Interpretation, Non-linear invariant, Hierarchical analysis, Relax transformer, Formal Method**

I. Introduction

THEstatic analysis technique based on Abstract Interpretation (AI) does not execute the program dynamically, but analyzes its source code statically, and obtains the semantics automatically, thereby verifying the correctness of the program or finding possible errors. Abstract interpretation uses the abstract semantics to approximate the concrete semantics of the program, so it can obtain an approximation of programs reachable states at any program points in a limited time and space. This approximation may contain behaviors that do not exist in the original program, so it may generate false positives. Improving the accuracy to reduce the false alarms is one of the major challenges faced by AI in practical applications. The main precision loss of AI comes from the following two aspects: Firstly, the expressiveness of specified abstract domain limits the precision of AI, such as numerical abstract domain Intervel (x=[a,b]), Octagon (....), Polyhedron (...) that commonly used can only express a linear convex space, and cannot described for nonlinear or non-convex spaces accurately. Secondly, the widening is usually introduced to speed and ensure the convergence of the iterative loop analysis, which aggressively increases the abstracted feasible program states at the loop head, and leads to precision losing heavily. In particular, using a linear abstraction domain to analyze a program with non-linear loop invariants can easily widen the values of variables to infinite boundaries, thereby introducing a large number of false alarms of run-time errors.

Figure 1 a motivating example Consider the motivating program in Fiagure 1(a), which computes the sum of all integers from -9 to 21 through a loop and stores the result in variable y. As a concrete execution of this program, the values of x and y at the program point 1 always satisfy the quadratic equation y = (x-9) * (x + 10)/2, where -10 ¡= x ¡= 20, that is the parabolic segment (L) in Figure 1 (b). From this quadratic relation, it is not difficult to conclude that the value of y always satisfies -45 ¡= y ¡= 165, so there is no integer overflow after executing the statement y = y + x at program point 2. However, the analysis using standard AI with linear abstract domain (including interval, octagon or polyhedron abstract domain, etc.) and normal widening operator can only generate meaningless invariant about variable y at program point 1, that is ? belongs to [-oo,+oo], then the analysis concludes there is an integer overflow error at point 2. And this is obviously a false alarm. In order to eliminate these alarms from non-linear programs, many work designed specific non-linear abstract domains. However, such domains usually have high time and space complexity and are not universal for all the non-linear features of programs. And most of them can only deal with polynomial invariants with a use-provided degree. For example, the work in [][] can generate the most precise invariants of our motivating example with degree 2, but if we modify the statement Y = Y + X; to Y = Y + X * X * X ;, these work will not be able to express these polynomial invariants with higher degree. And the constraints of y during the analysis will be lost after widening, then still generate integer overflow false alarms.

We proposes a new hierarchical method for loop analysis. The main idea of our method is using calculated partial invariants (only involving partial program variables) of loop to relax the transfer functions of program, then we take another analysis on relaxed program with more variables but little instability factor during widening. As a result, we can obtain more accurate loop invariant of all program variables in limited iterations of analysis. Firstly, this paper proposes and constructs a variable hierarchy graph based on variable dependencies of loop, and then builds a sliced program sequence (SPS) based on the variable hierarchy graph. Then, we get loop invariants of lower-level variables (called partial invariants) by analyzing the sliced program only involving lower-variables. Next, we use these partial invariants to relax the transfer function of the successor program involving higher-variables from SPS, in order to reach a more accurate fix-point on relaxed program by static analysis based on AI with normal widening. Then, on this basis of transformer relaxing, we also proposes a fix-point solving technology called Formula Method on single variable assignment with interval coefficient to improving the analysis of programs with exponential loop.

Manuscript received December 1, 2012; revised August 26, 2015. Corresponding author: M. Shell (email: http://www.michaelshell.org/contact.html).

Finally, we introduce our framework combining hierarchical loop analysis and transformer relaxing, and implement a prototype tool. The preliminary experimental results show that our method can obtain more accurate invariants.

For our motivating example, our method can generate a linear invariants about x and y expressed by polyhedron abstract domain at the loop header x¡=20, -x¡=10; -9x-y ¡=90; -21x+y¡=210. Geometrically, the state space represented by our invariants is shown in the gray triangle region in Figure 1(b), i.e. the hyper plane consisting of lines L'(-9x-y¡=90), L" (-21x+y¡=210), and L"' (x¡=20).From Figure 1(b), we can see that although the linear abstract domain polyhedron cannot accurately express the quadratic invariants of the original program (i.e. the geometric parabola), our method can use an linear polyhedron invariants to enclose the nonlinear invariants soundly. Come back to check whether the statement y=y+x; can cause an integer overflow, as we can get the range of x is [-9,21] and y is [-270,630] from our linear invariants above at program point 2, we can prove that there is no integer overflow after executing y=y+x;.At this point check the statement y=y+x again; if it causes an integer overflow, we can know from the linear constraint above that the range of x at program point 2 is [-9,21] and the range of y is [-270,630], so that we can prove The statement y = y + x; will not produce integer overflow, i.e. we have eliminated this false alarm of integer overflow at program point 2.

Organization. Section 2 illustrates the main framework of our method and how it works on our motivating example. Section 3 presents hierarchal variable dependency graph (HVDG) of loop and program slicing by hierarchal variables. Section 4 presents the definition, soundness and some strategies of relaxing. Section 5 describes a fix-point solver based on formula method. Section 6 presents discussions on soundness and precision of our method. Section 7 presents our experiment and evaluation. Section 8 discusses related work. Section 9 gives conclusions as well as future work.

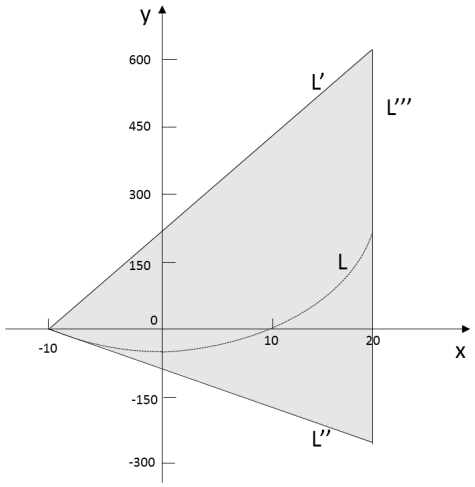

Fig. 1. Motivation Example

II. Overview

It is usual to find that there some directed dependencies between the variables in the loop of real programs. For example, loop control variables (such as loop counter) often do not depend on other variables in the loop, besides relying on themselves or some outside inputs; and many variables depend on loop control variables directly. If we use the abstract interpretation to analyze all the variables in the loop simultaneously, it is very likely that the precision loss of the analysis is huge after widening because of too many unstable factors in the adjacent iterations (for example, many variable values are changing). Especially for programs with non-linear behavior, such as our motivating example, the standard AI with linear abstraction domain always generates useless invariants due to widening by analyzing all the variables at the same time.

Based on the above observations, considering that the values of some variables in the loop depend on the values of other variables directed, we will firstly construct a hierarchical variable dependency graph for the variables involved in the loop. Then, according to the hierarchical variable dependence graph, an iterative refinement method is used to carry out hierarchical program analysis as following. First, standard AI generates the partial invariants of the underlying variables, then our method makes use of the partial invariants of underlying variables to analyze the high-level variables. The iteration continues from lower-level to higher-level variables until reaching the highest one.Specifically, when analyzing the i-th level variables, the original program is sliced using a set of variables related to that level (and lower level) to obtain a sliced version P(i) of the program. Then our method analyze the sliced program P(i), but during this analysis we use our relaxing operator to relax some expressions of P(i), and these expressions only have low-level variables whose invariants have generated by P(i-1). After relaxing, these instable factors due to direct data dependency between two adjacent iteration, and more constraints of variables are kept after widening operator, which makes our invariants more precise. One key point of our method is designing a valid Relaxing operator, which should provide relaxed transfer functions with more stable factors before widening. In the semantic equation of loop body, some transfer functions of statements have unstable expressions (i.e. their values will change between two iterations during computing the fix-point) and expressions which cannot be expressed effectively by specific abstract domain (e.g. linear polyhedron domain cannot express squared or higher-order expressions), and these expressions are called target expressions of relaxing. We extract the over approximations of these target expressions from the existing loop invariants and substitute these target expressions in the original loop statements with their over approximations, then we get a relaxed semantic equation. The above process is so-called Semantic Equation Relaxing. However, it is worth noting that the loop invariants relied on by Semantic Equation Relaxing is not necessarily an invariant involving all variables, but only these sound constraints between all the variables appearing in the target expression of relaxing, and these loop invariants is so-called

partial loop invariant for relaxing.

Fig. 2. Framework of our method Figure 2 shows the framework of our method. From this framework, it can be seen that our method of generating loop invariants mainly contains the following four processes: building a hierarchical variable dependency graph; program slicing based on variable hierarchy; semantic equation relaxing based on partial invariants, and fix-point solver by Kleene iteration and Formula Method. These processes will be described in detail in the following sections.

Now we will illustrate our method by analyzing the motivating example. Firstly, we analyze the data and control dependencies between these all the variables involved in the loop body, i.e. x and y. And we find that variable x only depends on the value of x itself and the variable y depends on the values of x and y. So we divide these two variables of x and y into two hierarchies, and x is placed in the first hierarchy and y is placed in the second hierarchy, as shown in Figure 3(a). Next, the program is first sliced using the first hierarchy variable x to get program P(0), as shown in Figure 3(b). Since P(0) only involves the lowest level variable, no relaxation is needed. Then we use the invariant generation technique based on abstract interpretation to analyze P(0), and the partial invariant of the variable x at the program point 2 can be obtained as x=[-9,21]. Next, we analyze the second-level variables and slice the program using the first and second-level variables x,y to get P(1), as shown in Figure 3(c). Note that P(1) is the original program. Then, we relax the assignment of the second hierarchy variable in P(1) (that is, the statement y=y+x;//y=y+x*x*x; at the program point 2) with partial invariant generated by analyzing P(0), i.e. we replace the first level variable x with its interval range [-9,21] in statement y=y+x. The semantic equation after relaxing is equivalent to the program P(1') shown in Figure 3(d). Note that now the second level variable y only depends on itself and no longer depends on x. Applying the standard AI with polyhedron abstract domain to analyze the program P(1), our method can obtain a linear invariant about the x and y at the loop header as following: -x+20¿=0, x+10¿=0, 9x+y+90¿=0, 21x-y+210¿=0 Since this program only has two variable hierarchies, the above linear invariants are the output invariants of our method. And this invariant is obviously more precise than the standard AI with polyhedron without our method.

Fig. 3. Sliced and relaxed versions of the original program

III. Variable Layering and Cyclic Slicing Based on Variable Dependent Layered Graphs

Hierarchal Variable Dependency GraphHVDG

A. Variable Dependency

Definition of VD

B. Variable Dependency Graph

1) Some Definitions

Hasse Diagram

2) Construct VDG

We will introduce how to construct the VDG.

C. Variable layering

How to layer variables based on Hasse Diagram.

D. Slicing based on variable Hierarchy

How to slice the loop by vh.

IV. Relaxing transformers based on partial loop invariant

A. Definition of Relax and its soundness

Definition of Relax Soundness of Relax

B. Relaxing Strategy

1) All variables relaxing strategy

2) Templated relaxing strategy

(1)Linearization Relaxing of single assignments
(2)Relational Relaxing of multiple assignments
(3)Relaxing of single variable assignments

V. Computing Fix-point by Formula Method

A. Formula Method of single variable assignments

B. Loop counter calculation by Formula Method

C. Fix-point solver combining Kleene iteration and Formula Method

VI. Discussion

Talk about the soundness and precision of our method.

VII. Experiment and Evaluation

VIII. Conclusion

The conclusion goes here.

Appendix A
Proof of the First Zonklar Equation

Appendix one text goes here.

Appendix B

Appendix two text goes here.

Acknowledgment

The authors would like to thank...

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.

Michael Shell Biography text here.

John Doe Biography text here.

Jane Doe Biography text here.