Approximating Context-Sensitive Program Information

Mathias Hedenborg[1], Jonas Lundberg[1], Welf Löwe[1], and Martin Trapp[2]

[1] Linnaeus University, Software Technology Group,
{Mathias.Hedenborg|Jonas.Lundberg|Welf.Lowe}@lnu.se
[2] Senacor Technologies AG, Martin.Trapp@senacor.com

Abstract. Static program analysis is in general more precise if it is sensitive to execution contexts (execution paths). In this paper we propose χ-terms as a mean to capture and manipulate context-sensitive program information in a data-flow analysis. We introduce *finite k-approximation* and *loop approximation* that limit the size of the context-sensitive information. These approximated χ-terms form a lattice with a finite depth, thus guaranteeing every data-flow analysis to reach a fixed point.

1 Introduction

Static program analysis is an important part of optimizing compilers and software engineering tools. These analyses predict properties of any execution of a given program, referred to as *program information*, by abstracting from its concrete execution semantics and its potential input values. Analyses can be context-sensitive or -insensitive, i.e., an analysis may or may not distinguish program information for different execution paths, i.e. for different *contexts*, e.g., the call contexts of a method. Context-sensitive analyses are, in general, more precise than their context-insensitive counterparts but also more expensive in terms of time and memory consumption.

In an iterative program, there are countably (infinitely) many contexts. Hence, merging the program information of some contexts is needed for the analysis to terminate. This, however, makes the analysis less context-sensitive, hence, less precise.

In Trapp et al. [THLL15], we focussed on capturing context-sensitive analysis information, i.e. contexts and program information for each program point, in a memory efficient way. In other words, we strived to delay merging the program information of different contexts for keeping analysis precision high. In the present paper, we discuss approximations that sacrifies precision for memory.

2 Background

This section introduces the notions and construction of χ-terms as components in static program analysis. Details are given in [THLL15].

2.1 SSA Representation

We assume the analysis to be based on a *Static Single Assignment* (SSA) graph representation [CFR$^+$91] of a program. Nodes in the SSA graph represent program points;

special ϕ-nodes represent merge points of the execution paths, i.e., contexts. Here we distinguish the program information of incoming paths by creating a χ-term connected to sub-terms, each representing the program information analyzed for the respective incoming execution path.

Figure 1 shows an example code with corresponding basic block and SSA-graph representations.

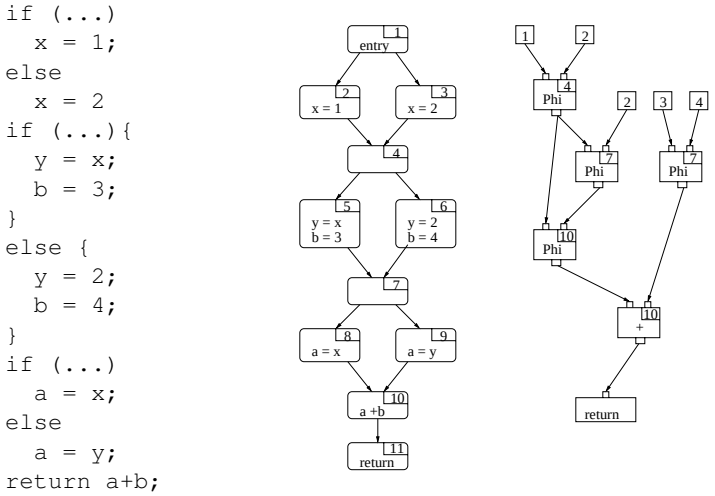

Fig. 1. A source code example with corresponding basic block and SSA graph structures.

In the figure the source code is transfered into numbered basic blocks (middle), and based on this a ϕ-node based SSA-graph have been generated (right). The ϕ-nodes will there be the merging point for different definitions of values for a given variable.

2.2 χ-terms

A χ-function is a representation of how different control-flow options affect the value of a variable. For example, we can write down the value of variable b in block 7 in Figure 1 using χ-functions as $b = \chi^7(3, 4)$. Interpretation: variable b has the value 3 if it was reached from the first predecessor to block 7 in the control-flow graph, and the value 4 if it was reached from the second predecessor block. That is, a value expressed using χ-functions (a so-called χ-term) does not only contain all possible values, it also contains which control-flow options that generated each of these values.

The construction of the χ-term values and the numbering of the χ-functions is a part of a context-sensitive analysis. Every ϕ-node in an SSA graph represents a join point for several possible definitions of a single variable, say x. When the analysis reaches a block b containing a ϕ-node for x it "asks" all the predecessor blocks to give their definition of x and constructs a new χ-term $\chi^b(x_1, \ldots, x_n)$ where x_i is the χ-term value for x given by the i:th predecessor. If the i:th predecessor block does not define x

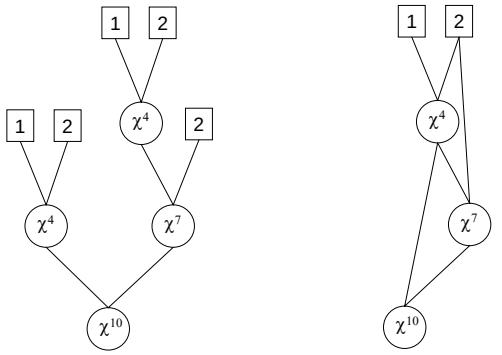

Fig. 2. Tree view of $\chi^{10}(\chi^4(1,2), \chi^7(\chi^4(1,2),2))$ and its graph representation.

by itself, it "asks" its predecessor for the value. This process continues recursively until each predecessor has presented a χ-term value for x. The process will terminate if any use of a value also has a corresponding definition.

In summary, a χ-term is a composition of χ-functions and analysis values $a, b, \ldots \in V$. Each program p has a (possibly infinite) set of χ-functions $\mathcal{X}(p)$ and each χ-function $\chi^b_j \in \mathcal{X}(p)$ is identified by a pair (b, j) where the *block number* b indicates in what basic block its generating ϕ-node is contained, and the *iteration index* j indicates on what analysis iteration over block b the χ-function was generated.

2.3 Tree and Graph Representation of χ-terms

Every χ-term can be naturally viewed as a tree. This is illustrated in Figure 2 (left) where we show the tree representation of the χ-term $\chi^{10}(\chi^4(1,2), \chi^7(\chi^4(1,2),2))$. Each edge represents a particular control flow option in this view and each path from the root node to a leaf value contains the sequence of control flow decisions required for that particular leaf value to come into play. A more compact graph representation (DAG) can easily be found by reusing identical subtrees, cf. Figure 2 (right), thus avoiding redundancies.

3 χ-term Approximations

In this section, we present two different approximations to the context-sensitive approach outlined above. We refer to these two approximations as the *finite k-approximation* and the *loop approximation*.

3.1 The Finite k-Approximation

The construction of new χ-terms is a part of the context-sensitive analysis. When the analysis reaches a ϕ-node in block b for a variable x, it constructs a new χ-term

by composing χ^b with all possible values for x. The newly constructed χ-term embodies all control-flow options that might influence the value of x at that point. The size of the χ-term representing x grows larger (without upper limit) as the analysis proceeds and more and more control-flow options influences the value of x. The finite k-approximation of χ-terms can be seen as on operation on the tree representation $G_t = \{N, E, r\}$. Whenever a new χ-term t is generated we replace all χ-terms $t_{sub} = \chi_i^b(t_1, \ldots, t_n)$ in $subterms(t)$ that has $depth(t_{sub}, t) \geq k$ with $\sqcup(t_1, \ldots, t_n)$, where $\sqcup$ is the union operator on the realted value lattice. The use of k means that the last k analysis steps have had influences on the current value. The process starts in the leafs and proceeds towards the root node. The result is a new χ-term $t^{(k)}$ with $depth(t^{(k)}) \leq k$.

3.2 The Loop Approximation

According to Trapp at al. [THLL15] we know that the analysis of a loop will generate χ-terms like $x_n^b = \chi_n^b(\ldots \chi_{n-1}^b(\ldots)\ldots)$. That is, the newly created χ-term will have a subterm with the same block number and a lower iteration index. This pattern will probably occur over and over again since each loop iteration results in a new composition of χ^b with itself. This will result in χ-terms of infinite depth and a non-terminating analysis if no measure is taken to stop the iterations. Informally, a χ-term $t = \chi_i^b(t_1, ..., t_n)$ is loop-approximated if every subterm of t that has the same block number as t is replaced by its context-insensitive approximation.

4 Result

In the previous section, we introduced two different approximations that make sense in almost any type of analysis. The loop approximation is necessary to guarantee analysis termination and k in the finite k-approximation is a precision parameter that can be seen as the size of "context memory" which decides how many previous control-flow options that each χ-term should try to remember.

By using both these approaches in the analysis phase we can handle the need of extra information to meet the demands for context sensitivity and the precision in the result of the program analysis.

In the full paper we present: a) formal definitions of both k- and loop-approximations, b) efficient algorithms for both, c) proofs showing that approximated χ-terms forms a finite value lattice (depth k) guaranteeing each analysis to reach a fixed point.

References

[CFR+91] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[THLL15] Martin Trapp, Mathias Hedenborg, Jonas Lundberg, and Welf Löwe. Capturing and manipulating context-sensitive program information. *Software Engineering Workshops 2015*, 1337:154–163, 2015.