PAG – an efficient program analyzer generator

Florian Martin

Universität des Saarlandes, FB14 Computer Science Department, P.O. Box 151150, D-66041 Saarbrücken, Germany;
E-mail: florian@cs.uni-sb.de

Abstract. In order to produce high quality code, compilers have to perform efficiency increasing program transformations. These transformations usually depend on preceding program analyses. These analyses, known as data flow analyses or abstract interpretations, may range from “simple” intraprocedural bit vector frameworks to very complex interprocedural alias analyses. Their implementation may be difficult and expensive. By exploiting the underlying theories of abstract interpretation and data flow analysis the implementation and design of analyzers can be supported by a tool. Abstract interpretation provides the relation to the semantics of the language and allows the systematic derivation of provably correct and terminating analyses. Data flow analysis supplies many efficient algorithms, such as fixed point iterations.

The program analyzer generator PAG described in this paper attempts to offer the best of both worlds, specification languages based on the clean theory of abstract interpretation and efficient implementation methods from the theory of data flow analysis. PAG has a high level functional input language to specify data flow analyses. It offers the generation of complex data structures and is therefore not limited to bit vector problems. PAG generated interprocedural analyzers can be easily integrated into existing compilers.

PAG has successfully been used in the ESPRIT project COMPARE to generate several analyzers (including alias analysis and constant propagation) for industrial quality ANSI-C and Fortran90 compilers, and is now marketed by the spin-off company AbsInt.1 A simplified version of PAG can be interactively tested over the Web.2

Key words: Data flow analysis – Specification and generation of analyzers – Abstract interpretation – Interprocedural analysis – Compiler construction

1 Introduction

In the area of compiler construction there is a long tradition in the automatic generation of software. Generators like lex and yacc for frontend generation are used routinely to implement all kinds of language interfaces. Backend generators like beg [12] or code selector generators like iburg [14] are getting more and more popular. So far, generators that support the optimization phase have not been very successful, because the proposed generators have been either inefficient or too restricted.

We present a system to generate interprocedural analyzers, PAG [2, 4, 23], which is able to produce analyzers that can be integrated into compilers by instantiation of a well designed interface. The system is based on the theory of abstract interpretation. The philosophy of PAG is to support the designer of an analyzer by providing languages for specifying the data flow problem and the abstract domains. This simplifies the process of constructing the analyzers, supports the correctness proof, and leads to a modular structure for the specification. The user of PAG is confronted neither with the implementation details of domain functionality nor with the traversal of the control flow graph or syntax tree nor with the implementation of suitable fixed point algorithms.

Generated analyzers have been tested with imperative languages, as well as with machine code. But the analysis of object oriented and logical languages is possible as well.

2 Overview

The paper is organized as follows. After an introduction to abstract interpretation in Sect. 3 the underlying theory is briefly summarized in Sect. 4, then the different interprocedural analysis methods are discussed in Sect. 5. Section 6 gives an overview over the generation given. Section 7 presents the specification languages, and Sect. 8 a short implementation description. A performance evaluation is presented in Sect. 9. Finally, Sect. 10 describes related works and Sect. 11 draws a conclusion.

1 http://www.AbsInt.de
2 http://www.cs.uni-sb.de/~martin/pag
3 Program analysis and abstract interpretation

Program analysis is a widely used technique to determine runtime properties of a given program automatically, without actually executing it. A program analyzer takes a program as input and computes some interesting properties. Most of these properties are undecidable. Hence, correctness and completeness of the computed information are not both achievable together. In program analysis there cannot be any compromise on the correctness side; the computed information is relied upon for enabling optimizing transformations. It cannot thus guarantee completeness. The quality of the computed information, usually called its precision, should be as good as possible.

A general framework for static program analyses called abstract interpretation is described by [10]. It is semantics based, so it supports correctness proofs of program analyses. Abstract interpretation amounts to perform a program’s computation using value descriptions or abstract values in place of concrete values.

One reason for using abstract values instead of concrete ones is to ensure that analysis results are obtained in finite time. Another reason is to obtain results that describe the computations on all possible inputs.

Examples of abstract interpretations from mathematics and everyday life are “computation with residues”, “casting out of nines”, and the “sign rules” [37].

There is a natural analogy described in [18]: abstract interpretation is to formal semantics as numerical analysis is to mathematical analysis. Problems without any known analytic solution can be solved numerically, giving approximate solutions, for example a numerical result r and an error estimate e. The solution is acceptable for practical usage if e is small enough.

An approximate program analysis is safe if its results can always be depended on. Results are allowed to be imprecise as long as they are on the safe side, i.e., if a boolean variable is sometimes true, then its value is safely described by “I don’t know”, but not by “false”. Safety always depends on how the abstract values are interpreted in relation to actual computations.

In PAG, data flow analysis is used as a special case of the abstract interpretation. But some advantages are taken from the general theory of abstract interpretation, such as the strong relation to the original semantics and some iteration techniques like widening.

4 Theoretical background

4.1 Overview

A complete discussion of data flow frameworks and their characterizing properties can be found in [22], [26] gives an overview of program analysis.

The goal of program analysis is to compute properties for every program point in a given input program. These properties are described by abstract values in a certain domain D. The program points of interest are the points directly before and directly after the execution of a statement.

The basis of program analysis is the control flow graph (CFG). It represents the program to be analyzed: for each statement in the program there is a node in the CFG, and every edge represents a possible transfer of control during the execution.

The result of the analysis is a labeling of each node n of the CFG with an abstract value from D, which is valid directly before the execution of the statement corresponding to n.

To compute these values, to every edge in the CFG a function $f : D \rightarrow D$ is assigned, which describes the modification of the abstract values, whenever the control flows along this edge.

4.2 Basic definitions

Definition 1. A control flow graph (CFG) is a graph $G = (N,E,s,e)$, with a set N of nodes, a set $E \subseteq N \times N$ of edges, a start node s and an end node e, $s,e \in N$. If $(n,m) \in E$, n is called a predecessor of m ($m$ is a successor of n). s is required to have no predecessor, e to have no successor.

Additionally, it is required for the rest of the paper that CFGs are connected, i.e., there are no nodes that cannot be reached from s. An example is given in Fig. 1.

```
int fac(int n)
{
    int res;
    res=1;
    while (n>1) {
        res=res*n;
        n=n-1;
    }
    return res;
}
```

Fig. 1. A program and its CFG

Definition 2. A path $\pi$ in a CFG $(N,E,s,e)$ is a sequence of edges, beginning with a node $n_1$ and ending with some node $n_k : \pi = (n_1,n_2),(n_2,n_3),\ldots,(n_{k-1},n_k)$ with $(n_i,n_{i+1}) \in E$.

For PAG it is required that the set of abstract values forms a complete lattice. (In some cases it is possible to relax this prerequisite.)