

Static Program Analysis

Anders Møller and Michael I. Schwartzbach

September 9, 2020

Copyright © 2008–2020 Anders Møller and Michael I. Schwartzbach

Department of Computer Science
Aarhus University, Denmark

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>.

Contents

Preface	iii
1 Introduction	1
1.1 Applications of Static Program Analysis	1
1.2 Approximative Answers	3
1.3 Undecidability of Program Correctness	6
2 A Tiny Imperative Programming Language	9
2.1 The Syntax of TIP	9
2.2 Example Programs	12
2.3 Normalization	13
2.4 Abstract Syntax Trees	14
2.5 Control Flow Graphs	14
3 Type Analysis	17
3.1 Types	18
3.2 Type Constraints	20
3.3 Solving Constraints with Unification	22
3.4 Record Types	27
3.5 Limitations of the Type Analysis	30
4 Lattice Theory	33
4.1 Motivating Example: Sign Analysis	33
4.2 Lattices	34
4.3 Constructing Lattices	36
4.4 Equations, Monotonicity, and Fixed-Points	39
5 Dataflow Analysis with Monotone Frameworks	47
5.1 Sign Analysis, Revisited	48
5.2 Constant Propagation Analysis	53
5.3 Fixed-Point Algorithms	55

5.4	Live Variables Analysis	59
5.5	Available Expressions Analysis	63
5.6	Very Busy Expressions Analysis	66
5.7	Reaching Definitions Analysis	67
5.8	Forward, Backward, May, and Must	69
5.9	Initialized Variables Analysis	71
5.10	Transfer Functions	72
6	Widening	75
6.1	Interval Analysis	75
6.2	Widening and Narrowing	77
7	Path Sensitivity and Relational Analysis	83
7.1	Control Sensitivity using Assertions	84
7.2	Paths and Relations	85
8	Interprocedural Analysis	93
8.1	Interprocedural Control Flow Graphs	93
8.2	Context Sensitivity	97
8.3	Context Sensitivity with Call Strings	98
8.4	Context Sensitivity with the Functional Approach	101
9	Control Flow Analysis	105
9.1	Closure Analysis for the λ -calculus	105
9.2	The Cubic Algorithm	106
9.3	TIP with First-Class Function	107
9.4	Control Flow in Object Oriented Languages	111
10	Pointer Analysis	113
10.1	Allocation-Site Abstraction	113
10.2	Andersen's Algorithm	114
10.3	Steensgaard's Algorithm	116
10.4	Interprocedural Points-To Analysis	117
10.5	Null Pointer Analysis	118
10.6	Flow-Sensitive Points-To Analysis	121
10.7	Escape Analysis	123
11	Abstract Interpretation	125
11.1	A Collecting Semantics for TIP	125
11.2	Abstraction and Concretization	131
11.3	Soundness	137
11.4	Optimality	143
11.5	Completeness	145
11.6	Trace Semantics	149
	Bibliography	153

Preface

Static program analysis is the art of reasoning about the behavior of computer programs without actually running them. This is useful not only in optimizing compilers for producing efficient code but also for automatic error detection and other tools that can help programmers. A static program analyzer is a program that reasons about the behavior of other programs. For anyone interested in programming, what can be more fun than writing programs that analyze programs?

As known from Turing and Rice, all nontrivial properties of the behavior of programs written in common programming languages are mathematically undecidable. This means that automated reasoning of software generally must involve approximation. It is also well known that testing, i.e. concretely running programs and inspecting the output, may reveal errors but generally cannot show their absence. In contrast, static program analysis can – with the right kind of approximations – check all possible executions of the programs and provide guarantees about their properties. One of the key challenges when developing such analyses is how to ensure high precision and efficiency to be practically useful. For example, nobody will use an analysis designed for bug finding if it reports many false positives or if it is too slow to fit into real-world software development processes.

These notes present principles and applications of static analysis of programs. We cover basic type analysis, lattice theory, control flow graphs, dataflow analysis, fixed-point algorithms, widening and narrowing, path sensitivity, relational analysis, interprocedural analysis, context sensitivity, control-flow analysis, several flavors of pointer analysis, and key concepts of semantics-based abstract interpretation. A tiny imperative programming language with pointers and first-class functions is subjected to numerous different static analyses illustrating the techniques that are presented.

We take a *constraint-based approach* to static analysis where suitable constraint systems conceptually divide the analysis task into a front-end that generates constraints from program code and a back-end that solves the constraints to produce the analysis results. This approach enables separating the analysis

[Click here to download full PDF material](#)