Notes on Program Analysis

Alex Aiken

April 20, 1994

1 Background

One of the most difficult aspects of program compilation is designing appropriate *program analyses*. A program analysis is just a special-purpose theorem prover, designed to prove useful facts about a program. In the sense used here, program analysis is a good deal more than verifying simple syntactic properties of programs (e.g., whether all variables are declared) and a good deal less than full program verification. In practice, however, this "middle ground" is not well-defined and program analysis algorithms span the spectrum from very simple to quite elaborate, with the only real constraints being the compiler writer's imagination and the pragmatics of building efficient systems.

Before delving into a discussion of program analysis in general, we should start with a more basic question: Why have program analysis at all? That is, why is it that virtually all compilers have a particular organization—division into parsing, program analyses, program transformations and optimization, resource assignment (e.g., register allocation), code generation—and not some other organization? The honest answer to this question is "Because that's the way it is!" The standard organization of compilers, which varies relatively little among programming languages, has grown out of the experience of many language development projects over a period of decades. The FORTRAN I compiler, the first compiler ever written, already had the essential building blocks that are used today, including program analysis components. Compiler organization subsequently has been refined and formalized, but not radically altered. Today it can be said that the "standard" compiler organization is universally accepted almost without question, although radically different organizations have hardly been explored.

While the building blocks of compilers have remained fixed for a substantial period of time, the relative cost and difficulty of the major components has changed dramatically. Very early, the major intellectual and implementation problem was parsing; later the focus shifted to issues intimately tied to code generation, such as register allocation. In this latter period, which extended into the 1980's, program analysis and optimization was primarily the domain of FORTRAN compilers. Early C compilers, for example, did almost no program analysis or optimization. In fact, even in the mid 1980's it was not expected that a C compiler would perform optimizations that FORTRAN compilers had incorporated since FORTRAN I.

Today much of the research activity in the implementation of programming languages is focussed on program analysis problems. This is because program analysis is intellectually difficult, computationally expensive (analysis time is the dominant cost in many optimizing compilers), and increasingly important. There are many reasons for the increased importance of program analysis, but two in particular stand out: (1) languages are moving toward higher levels of abstraction, and (2) machine architectures are becoming increasingly difficult to program in anything but a high-level language. For (1), the trend in programming languages is toward languages with increased expressive power and less detail. Evidence of this trend is found in functional, logic, and object-oriented languages—almost all languages except the vanilla Algol-family languages. Among modern languages there is a great disparity in the level of programming abstraction provided, but the overall trend toward greater abstraction is clear. For (2), trends in hardware design, such as increased parallelism within a processor (both through pipelining and through multiple functional units), make modern machines very difficult to program by hand at the assembly language level.

The bottom line of points (1) and (2) is that the gap between the program that a programmer writes and the code that actually runs on a machine is growing, and growing fairly rapidly. As languages become higher level, machines are at the same time becoming lower level, with more detail and complexity exposed to software. To bridge the widening gap between programs and machines, compilers are becoming more sophisticated, particularly in the area of program optimization. Typically, program optimization is a source-to-source transformation where the original program is replaced by an improved program. In many cases, the actual syntactic program transformations are easy to express and implement. The difficult part of program optimization is establishing that a transformation does not change the meaning of a program. Establishing such facts is the domain of program analysis.

As mentioned above, there are other motivations for contemporary interest in program analysis. In the computing industry, a primary motivation is simple competition. Today, a significant portion of the improvement in workstation performance is due not to any improvement of the underlying hardware but instead to improvements in compiler optimization and the program analyses on which optimization relies. Note that the annual rate of improvement in hardware far outstrips the annual rate of improvement in compiler technology, but improved compiler technology still provides real additional performance benefits. As a result, the last ten years have seen dramatic improvements in the quality of C and FORTRAN compilers. Currently, no other language, excepting C++, has a large enough following to benefit from such competitive pressures.

The last, but certainly not least, motivation for studying program analysis is the desire for improved software engineering through automatic techniques for program verification. As a research area, program verification has fallen into disrepute following a period of considerable hype and few results. In fact, full program verification is well beyond the capability of any known method. However, verifying limited properties of programs is tractable and demonstrably useful. The classic example is the Hindley/Milner polymorphic type inference algorithm which can guarantee that a program is free from a large class of common programming errors.

2 Correctness of Program Analysis

We have now come to a point in the discussion where it is difficult to proceed without defining more precisely what program analysis is. Historically, this question was treated rather informally for a period of twenty years from the late '50s to the late '70s. This fact is not so surprising when one considers that the conceptual tools for reasoning about any aspect of language semantics were not developed until the early '70s and that, in fact, formal treatments of program analysis appeared shortly thereafter.

The seminal paper on program analysis is Cousot and Cousot [CC77]. In their view, a program analysis is an *abstract interpretation* of programs. Briefly, a programming language has a standard interpretation μ mapping programs e to values $\mu(e)$. An abstract interpretation consists of another interpretation σ that maps programs e to abstract values $\sigma(e)$. The abstract values are abstractions of the standard values; this property is formalized by a function γ that maps every abstract value a to the set of (standard) values $\gamma(a)$ that a represents. If a (standard) value v is in $\gamma(a)$, then a is said to be an abstraction of v.

Definition 2.1 Let σ, γ, and μ be given. Then the abstract interpretation of μ defined by σ and γ is *correct* if and only if for all programs e:

$$\mu(e) \in \gamma(\sigma(e))$$

The Cousots' framework formally captures the intuitive notion that a correct program analysis is conservative—i.e., a program analysis approximates and in fact may be imprecise. Abstract interpretation provides a suitable basis for justifying the correctness of program analyses. There are two problems, however. The first problem is mostly an annoyance: abstract interpretations are defined with respect to a standard semantics and there are many ways of giving a standard semantics to programs. Thus, the general ideas of abstract interpretation must be adapted depending on how the standard semantics of a particular language is defined.

The second problem is more serious: abstract interpretation is a semantic framework only. That is, abstract interpretation is useful for showing that a program analysis is correct and is not directly useful for much else. In particular, abstract interpretation doesn't assist one in designing program analyses, comparing program analyses (unless they happen to be over the same domain), or in evaluating the speed versus precision (the "price-performance") of a program analysis.

In their original paper on abstract interpretation, the Cousots did propose one algorithmic framework that gives an "off-the-shelf" procedure for constructing program analyses [CC77]. Program analyses in this style are essentially variations on a single theme. A finite abstract domain A is designed (A is usually a lattice) and the program analysis is expressed in terms of a system of recursive equations of the following form

$$x_1 = \sigma_1(X) \ldots x_n = \sigma_n(X)$$

where $X = \{x_1, \ldots, x_n\}$ is a set of variables and each σ_i is a monotonic function with signature $A^{|X|} \rightarrow A$. It is well-known that the least solution of such equations is computable [CC77]. Given that one can design a correct analysis in this framework, the implementation is straightforward and has two additional useful properties: first, the computed analysis is the best possible within the chosen parameters (i.e., it is the least solution of the equations) and second, the analysis is guaranteed to terminate. Analyses for C and FORTRAN programs based on dataflow equations are classic examples of this kind of program analysis.

The cookbook recipe "finite domains plus monotonic functions equals program analysis" has proven very popular in the research community. There are an enormous number of applications of this idea; representative examples are [Myc80, JM86, Hud87, Wad87, HY88, PBJ$^+$91]. The paradigm has become so popular, in fact, that today the term *abstract interpretation* usually refers to this specific algorithmic technique instead of a general semantic framework. For the purposes of this document, abstract interpretation is the semantic framework and not any particular analysis algorithm.

The cookbook recipe is popular because it provides a tool for constructing program analyses. It is also the right tool for some applications (e.g., [Myc80]). However, it also has serious limitations, and, except for the dataflow analyses discussed above, it is not widely used in practice. The critical limitations are:

- *Finite domains.* Using a finite domain allows one to make only finitely many distinctions. Thus, it is impossible to reason about unbounded data structures such as lists and trees without making gross approximations.[1]

- *Efficiency.* For many applications, the computational complexity of analyses in this class grows exponentially with the size of the abstract domain. Thus, in these applications the technique is viable only for small—and therefore rather coarse—abstract domains.

- *Forward vs. backward analysis.* Abstract interpretations in this class are generally either *forwards* (i.e., the analysis is carried out from inputs to outputs) or *backwards* (i.e., the analysis is from outputs back to inputs). Some program analyses require bidirectional information flow, which is difficult to express in this paradigm.

3 Paradigms for Program Analysis

The Cousots' cookbook recipe is an example of a program analysis paradigm. The shortcomings of this particular paradigm have sparked a search for other general algorithmic frameworks for program analysis. Today, there are three principal paradigms in use:

- the cookbook recipe and extensions

- type inference algorithms

- applications of formal logic

The cookbook recipe is discussed in the preceding section. There are ongoing efforts to extend the cookbook recipe to handle a wider variety of program analysis problems. For example, there is considerable interest in finding general techniques for computing fixed points in infinite domains [FH93]. This line of work has led in a number of interesting directions and continues to progress. Existing implementations of these ideas are very slow.

Type inference is receiving increasing attention as a program analysis paradigm. In this context, type inference means specifically the Hindley/Milner type inference algorithm [DM82]. This particular algorithm has been adapted in a number of ways to problems that have little or nothing

[1] The restriction to finite domains can be relaxed to finitary domains without changing the nature of the problem, but the objection that the approximations are necessarily coarse still applies.

to do with type inference. In a typical application, type inference rules are given for the programming language, with one rule for each syntactic form. The "types" capture the properties of interest (i.e., what it is that is being analyzed). The implementation of the type inference rules is done via unification of constraints, or a close variant of unification. Good examples of this paradigm are type and effect systems [GJSO91], Remy's system for inferring record types [Ŕ89], and Tofte and Talpin's algorithm for automatic storage management [TT93]. Perhaps not surprisingly, this paradigm has found the greatest number of applications in languages with higher-order functions, a language feature that can be modelled only awkwardly using the cookbook recipe.

The final paradigm is to apply tools from formal logic to program analysis problems. In this approach, program analysis problems are encoded as formulas in some logic. Performing the program analysis then corresponds to analyzing the constructed formula. An excellent example of this paradigm is the analysis of *graph types* in [KS93], where the safety of pointer manipulations is reduced to a decision problem in the theory of k-successors. The theory of k-successors is decidable, so the reduction gives an effective procedure for determining whether or not all the pointer operations in a program are safe.

In many applications it is desirable to have more than a "yes" or "no" answer from a program analysis. In such cases, it is more useful to construct an explicit representation of a model of the formula (if the analysis is successful) or a counterexample to the formula (if the analysis fails). Examples in this class include program analyses that are reducible to constraint resolution problems; e.g., all of the analyses based on *set constraints* [HJ90, AW92].

The type inference and logical paradigms at least partially address the problems associated with the cookbook recipe. There is no restriction to finite domains and in fact the techniques work quite well in infinite domains. Efficiency is still a concern, but a case can be made that the trade-off between precision and speed is not as acute in many applications of type inference and logical systems. Finally, constraint resolution analyses are naturally able to express simultaneous forward and backward flow of information, thus eliminating completely the forward/backward analysis distinction of the cookbook recipe.

It is worth pointing out that the type inference paradigm is really just another application of formal logic and so is not really a distinct paradigm at all. The reason for distinguishing type inference techniques from other logical approaches is historical and may in fact not be warranted in the future. The "type inference" label refers primarily to the work of a community of researchers who have been influenced strongly by the Hindley/Milner algorithm.

Similarly, the cookbook recipe can, with a bit of a mental stretch, be thought of as yet another distinct logical framework. After all, the cookbook recipe rests on the ability to solve equations over a finite domain, and thus fits within the general sphere of constraint resolution techniques. The "cookbook recipe" label refers primarily to that body of work whose lineage can be traced to the Cousots.

4 The Future

At this point the reader may feel a bit discouraged and inclined to think that the state of knowledge in program analysis is a rather disorganized and random collection of techniques. There is a good reason for this impression—it happens to be correct.

Some of the causes of this situation are probably historical accidents. In the future, the perspective may be that certain useful frameworks were discovered first and thereby received disproportionate attention. Today there are enough interesting approaches—motivated by something other than old techniques—that the world of program analysis is beginning to look more like a spectrum of algorithms with various price/performance trade-offs and less like a family of distinct paradigms.

If this view of the future is accurate, then the next step is to obtain a more systematic understanding of program analysis techniques—the relationships between the major paradigms, what is expressible in one but not the others, and speed/precision trade-offs. With some luck, such an effort would lead to insights that would serve to unify, simplify, and hopefully improve much of what is currently known.

References

[AW92] A. Aiken and E. Wimmers. Solving systems of set constraints. In *Symposium on Logic in Computer Science*, pages 329–340, June 1992.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by contruction or approximation of fixed points. In *Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[DM82] L. Damas and R. Milner. Principle type-schemes for functional programs. In *Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, January 1982.

[FH93] A. Ferguson and J. Hughes. Fast abstract interpretation using sequential algorithms. In *Proceedings of the 3rd International Workshop on Static Analysis*, pages 45–59, September 1993. LNCS #724.

[GJSO91] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole. Report on the fx programming language. Technical report, Department of Computer Science, MIT, 1991.

[HJ90] N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 197–209, January 1990.

[Hud87] P. Hudak. A semantic model of reference counting and its abstraction. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 45–62. Ellis Horwood Limited, 1987.

[HY88] P. Hudak and J. Young. A collecting interpretation of expressions (without powerdomains). In *Proceedings of the 15th Annual ACM Symposium on the Principles of Programming Languages*, pages 107–118, 1988.

[JM86] N. D. Jones and A. Mycroft. Dataflow analysis of applicative programs using minimal fucntion graphs: Abridged version. In *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 296–306, January 1986.

[KS93] N. Klarlund and M. I. Schwartzbach. Graph types. In *Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 196–205, January 1993.

[Myc80] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of the 4th International Symposium on Programming*, pages 269–281, April 1980. In LNCS No. 83.

[PBJ+91] K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence flow graphs: An algebraic approach to program dependencies. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 67–78, January 1991.

[Ŕ89] D. Rémy. Type checking records and variants in a natural extension of ML. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 277–87, January 1989.

[TT93] M. Tofte and J. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Twenty-First Annual ACM Symposium on Principles of Programming Languages*, January 1993. To appear.

[Wad87] P. Wadler. Strictness analysis on non-flat domains (by Abstract interpretation over finite domains). In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 266–275. Ellis Horwood Limited, 1987.