Perspectives on Program Analysis

Flemming Nielson
Computer Science Department, Aarhus University, Denmark

To guide the research efforts in the area of program analysis it is necessary to provide a taxonomy of the various approaches (identifying strengths and weaknesses), and to explore the links to programming languages and theoretical computer science.

In the past program analysis was mainly a tool for the compiler writer, but in the future it shows promise of being an important ingredient in ensuring the acceptable behaviour of software components roaming around on information networks. To guide the research efforts it is necessary with an appraisal of the current technology and to investigate its links to programming languages and theoretical computer science.

Taxonomy

The *flow based* approach [4; 8] to program analysis includes the traditional data flow analysis techniques as well as the more recent control flow analysis techniques. In this approach the focus is on discovering *intensional* properties like use-definition chaining (associating uses of variables with the corresponding assignments or "definitions") and closure analysis (associating function applications with the corresponding labels of the functions applied). This allows to discover aspects of the *extensional* properties: constant propagation (determining that a variable is always a constant at a given use), and neededness analysis (whether or not a function actually uses its argument). On the positive side, this approach often gives rise to a rather efficient implementation; often it is performed by generating constraints that can then be solved efficiently, without being influenced by the syntax of the program being analysed. On the negative side, the semantic correctness of the analysis is seldom established and therefore there is often no formal justification for the program transformations for which the information is used.

The *semantics based* approach [1; 5] is often based on domain theory in the form of abstract domains modelling sets of values, projections, or partial equivalence relations. The approach tends to focus more directly on discovering the extensional properties of interest: for constant propagation it might operate on sets of values with constancy corresponding to singletons, and for neededness analysis it might perform a strictness analysis and use the strictness information for neededness (or make use of the "absence" notion from projection analysis and attempt to discover the difference). On the positive side, this usually gives rise to provably correct analyses, although there are sometimes complications (due to deciding what information to stick onto the program points) when formally justifying the program transformations. On the negative side, the implementations are often computationally too costly to be of general use.

The *inference based* approach [6; 7] includes general logical techniques as well as annotated type and effect systems built on top of Hindley-Milner type inference. In spirit the approach is close to that of the semantics based one except that logical formulae are used to denote the semantic entities. Consequently it shares many of the positive and negative aspects of the semantics based approach. A potential advantage over the semantics based approach is that the shape of inference trees seems promising for sticking information onto program points and thereby facilitates formally justifying the program transformations.

The *abstract interpretation based* approach [2; 3] is in a sense an intermediary between the flow based and semantics based approaches. It operates equally well on property spaces modelling intensional information as on property spaces modelling extensional information. On the positive side, it allows to give a systematic account of the design of flow based and semantics based program analyses by "calculating" the analyses rather than merely specifying them; additionally it offers general techniques like "widening operators" for the approximation of fixed points. On the negative side, the available technology is not very programming language dependent: good examples of widening operators are hard to find in the literature, and the approach is often based on a rather low-level notion of operational semantics.

Programming Languages

Program analysis has a number of applications but none as dominating as improving the quality of the code generated by compilers. It is important to consider whether or not the results of the analysis, and any imprecisions in the information obtained, is of relevance to the programmer or not. If it is only of relevance to the compiler internals there is no harm in having the analysis operate on the intermediate language of the compiler. If the information is also of relevance to the programmer, then the use of an intermediate language may make it impossible to present the information to the programmer. Consequently he may be unable to understand why the program cannot be implemented efficiently and how to modify the program. Similarly, there may be small tricks (like don't curry functions needlessly) that will help a certain analysis technology to produce more precise results or to do so less costly.

Another use of program analysis technology is to influence the design of programming languages with a view to producing languages that may be analysed efficiently and precisely. This is related to the impact semantics has had on programming language design: it is partly descriptive, explaining what a construct really does, and partly prescriptive, advising against constructs with intricate semantics. The current interest in safe systems on the information network, motivates producing software that can positively and automatically be shown to be free of unwanted behaviour; examples include guaranteeing that data will not be modified unless properly authorised, and that certain protocols of communication are always obeyed.

To increase its impact on programming languages it would be advantageous to develop general tools for program analysis (like `lex` and `yacc` for parsing) that could then be integrated into applications.

Theoretical Computer Science

One branch of Theoretical Computer Science is devoted to the *semantics* of computation. The semantics of program analysis is certainly an interesting field for applying ideas from domain theory (including denotational semantics) as well as logical systems inspired by linear logic, domain logics, and modal logic. However, it is not always clear that the demands posed by program analysis are simple variations of the demands posed by the semantics of programing languages; an example being the frequent identification of the computation ordering (as in domain theory) with the approximation ordering (as in the subset ordering). Perhaps there is a need for the demands of program analysis to more fundamentally influence the thinking of semantics, rather than to see program analysis as merely an interesting field for applications.

Another branch of Theoretical Computer Science is devoted to the study of *algorithms* and their complexities. Program analysis is largely a consumer of results and techniques in this area: to ensure that the program analyses may be efficiently implemented, and to warn against needlessly improving the precision of analyses that are already too costly (say worse than exponential); the latter seems the rule rather than the exception for a number of semantics based analyses. At the same time there are a number of places in program analysis where notions from complexity may facilitate the development of a richer theory; one example being a notion of complexity for widening operators.

To increase its link with theoretical computer science it would be helpful to identify the open problems and the areas most in demand for further research.

REFERENCES

[1] G. L. Burn, C. Hankin, and S. Abramsky. Strictness Analysis for Higher-Order Functions. *Science of Computer Programming*, 7:249–278, 1986.

[2] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. 4th POPL*, pages 238–252. ACM Press, 1977.

[3] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. 6th POPL*, pages 269–282. ACM Press, 1979.

[4] S. Jagannathan and S. Weeks. A unified treatment of flow analysis in higher-order languages. In *Proc. POPL '95*, pages 393–407. ACM Press, 1995.

[5] N. D. Jones and F. Nielson. Abstract Interpretation: a Semantics-Based Tool for Program Analysis. In *Handbook of Logic in Computer Science* vol. 4. Oxford University Press, 1995.

[6] P. Jouvelot and D. K. Gifford. Algebraic reconstruction of types and effects. In *Proc. POPL '90*, pages 303–310. ACM Press, 1990.

[7] H. R. Nielson and F. Nielson. Higher-Order Concurrent Programs with Finite Communication Topology. In *Proc. POPL '94*, pages 84–97. ACM Press, 1994.

[8] F. Nielson and H.R. Nielson. Infinitary Control Flow Analysis: A Collecting Semantics for Closure Analysis. To appear in *Proc. POPL '97*. ACM Press, 1997.