Abstract Interpretation and Partial Evaluation in Functional and Logic Programming

Neil D. Jones
DIKU, University of Copenhagen
2100 Copenhagen East
Denmark
neil@diku.dk

Abstract

The functional and logic programming research communities are to a significant extent solving the same types of problems, and by similar methods. The purpose of this talk is to encourage the two communities to show greater awareness of and openness to each other's methods and solutions.

1 Semantics-Based Program Analysis and Transformation Methods

Program analysis and transformation algorithms and systems tend quickly to become very complex. One way to reduce the possibility of erroneous results is by a transformational approach: to begin with a language semantics, or a precise but not necessarily computable mathematical definition of the problem to be solved, and then to transform this into an efficient algorithm by small steps, taking care to preserve correctness along the way.

A complicating factor is the fact that Prolog, for example, has several types of semantics: bottom-up deduction of ground terms, top-down search with backtracking, various interpretations of failure, etc. Consequently the question of just what it means to "preserve semantics" can vary with the different semantic assumptions.

A well-known tricky example within logic programs concerns the optimization of programs by introducing "difference lists". While this can, for instance, reduce a program's running time from quadratic to linear, it can also very often lead to incorrect programs. Marriott and Søndergaard describe safety conditions in [17] which are sufficient to ensure that semantics is preserved.

2 Abstract Interpretation

One use of the term "abstract interpretation" is to describe a semantics-based approach to program flow analysis (in contrast to the more ad hoc approaches often taken in traditional compiler writing).

Pathbreaking work in this direction was done by Patrick and Radhia Cousot [2], and a fairly broad overview is found in [15]. The former mainly concerned imperative languages, and the latter imperative and functional languages. An early paper on semantics-based abstract interpretation of a logic programming language was [21] by Søndergaard, which put an approach developed earlier by Plaisted on firmer semantic foundations. More comprehensive survey articles in this area include [8,18] and [3].

3 Complexity of Program Analysis

Early works on determining the computational complexity of imperative programs and their analysis include [9], [10], [11]. DeBray has done recent work in a similar vein, but applied to the analysis of logic programs [4].

Much more work could be done in this area, both as concerns complexity of various analyses, and of the relative expressive power of various language features. A good example is the "logical variable", which is often used to write programs faster than a naive approach would yield. The same concept has also been added to functional languages, to give some of the power of assignment or even arrays, but without loss of referential transparency.

A question: can it be proven formally that such language features actually increase problem-solving ability? In one context, one could ask whether within the same time or space bound, more problems can be solved by programs with the new feature than by ones without it. Alternatively, one could try to find a sigle problem which can provably be solved faster by some program using the feature than by any program without it.

These are complexity-theoretical questions, but ones that concern our daily tools: programming languages and their features. A related result appears in [12], where it is shown that increasing run time by even a constant factor properly increases problem-solving ability, for a reasonable programming language.

4 Partial Evaluation and Other Program Transformations

Again, the different communities are attacking simmilar but not identical problems with similar tools. Partial evaluation of mainly imperative and functional languages is described in detail in [14], and similar methods are applied to Prolog in [19].

Preservation of semantics is again a tricky matter, and especially problems concerning negation by failure are carefully addressed in [16]. Partial evaluation of a typed logic programming language is the subject of a recent thesis by Gurr [7].

Other transformations include Wadler's "deforestation" [23] and Turchin's "supercompilation" [24]. Deforestation has mainly been used in functional

programming, but the idea has also been applied to logic programming [20].

Recent work has led to much clarification of the similarities, differences, and relative strengths of these methods [22]. The paper [13] extracts the essence of supercompilation in a way independent of programming language or data structure, and it is seen to be properly stronger than traditional projection-based partial evaluation. For eample, the transformation of [1], which required a nontrivial program rewriting, can be accomlished by completely automatic means using supercompilation [6].

In connection with logic programming: the 'configurations" manipulated by the supercompiler are very similar to the ones most often used by Prolog partial evaluators. Further, paper [5] by Glück and Sørensen shows that in a sense, supercompilation and partial deduction are essentially identical.

Conclusion

This selection of results indicates a strong common ground between the automatic manipulation and analysis of functional and logic programs; consequently it will be in both communities' interests to become more aware of each others' work.

Acknowledgements

This research was supported in part by the Danish Research Council grant to the DART project, and by the Esprit Basic Research Action 6809, Semantique.

References

[1] **C. Consel and O. Danvy,** Partial Evaluation of Pattern Matching in Strings, *Information Processing Letters* 30 (January 1989) 79–86.

Summary: It is shown that the efficient Knuth-Morris-Pratt algorithm for string matching can be obtained from a naive matching algorithm by using partial evaluation.

[2] **P. Cousot and R. Cousot,** Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, *Fourth ACM Symposium on Principles of Programming Languages* 238-252, 1977.

[3] **P. Cousot and R. Cousot,** Abstract interpretation and application to logic programs, *Journal of Logic Programming* 13 (2-3) 103-179, 1992.

[4] **S. DeBray,** On the complexity of dataflow analysis of logic programs, *Journal of Logic Programming* 13 (2-3) 103-179, 1992.

[5] **R. Glück and M.H. Sørensen,** Partial Deduction and Driving are Equivalent, in *International Symposium on Programming Language Implementation and Logic Programming,* Springer-Verlag, 1994.

Summary: Partial deduction and driving are two methods used for program specialization in logic and functional languages, respectively. We argue that both techniques achieve essentially the same transformational effect by unification-based information propagation. We show their equivalence by analyzing the definition and construction principles underlying partial deduction and driving, and by giving a translation from a functional language to a definite logic language. We discuss residual program generation, termination issues, and related other techniques developed for program specialization in logic and functional languages.

[6] **R. Glück and A. V. Klimov,** Occam's Razor in Metacomputation: the Notion of a Perfect Process Tree, in *Static Analysis. Proceedings,* edited by G.Filè P.Cousot, M.Falaschi and A.Rauzy, pages 112–123, Springer-Verlag, 1993.

Summary: We introduce the notion of a perfect process tree as a model for the full propagation of information in metacomputation. The concept of a simple supercompiler based on perfect driving coupled with a simple folding strategy is explained and an example demonstrates that specializing a naive pattern matcher with respect to a fixed pattern obtains the efficiency of a matcher generated by the Knuth, Morris and Pratt algorithm.

[7] **C. Gurr,** A Self-applicable Partial Evaluator for the Logic Programming Language Gödel, Ph.D. thesis, University of Bristol, 1994.

[8] **N. D. Jones and Harald Søndergaard,** A Semantics-Based Framework for the Abstract Interpretation of Prolog, in *Abstract Interpretation of Declarative Languages,* edited by S. Abramsky and C. Hankin, pages 123–142, Ellis Horwood, Chichester, England, 1987.

Summary: A semantics-based technique is developed for the static analysis of prolog programs.

[9] **N. D. Jones and S.S. Muchnick,** Complexity of flow analysis, inductive assertion synthesis, and a language due to Dijkstra, in *Program Flow Analysis,* edited by S. Muchnick and N. Jones, pages 380–393, Prentice Hall, New Jersey, 1981.

[10] **N. D. Jones and S. Muchnick**. Even simple programs are hard to analyze. *Journal of the ACM*, 24(2):338–350, 1977.

[11] **N. D. Jones and S. Muchnick**. Complexity of finite memory programs with recursion. *ACM*, 25(2):312–321, 1978.

[12] **N. D. Jones,** Constant Time Factors *Do* Matter, in *STOC '93. Symposium on Theory of Computing,* edited by Steven Homer, pages 602–611, ACM Press, 1993.

Summary: It has long been disconcerting that partial evaluation only gives linear speedups, whereas the *constant speedup theorem* from Turing machine based complexity theory says that constant factors make no difference at all in computing power. In this paper the constant speedup theorem is shown false for a simple imperative programming language $\mathcal{L}$ manipulating tree-structured data — that the collection of all sets recognizable in linear time by $\mathcal{L}$-programs, contains an infinite hierarchy ordered by constant coefficients. Further results concern nondeterministic linear time sets, and an efficient version of Kleene's Second Recursion Theorem.

[13] **N. D. Jones,** The Essence of Program Transformation by Partial Evaluation and Driving, in *Logic, Language and Computation, a Festschrift in honor of Satoru Takasu,* edited by Masahiko Sato N. D. Jones, Masami Hagiya, pages 206–224, S-V, April 1994.

Summary: A new abstract framework is developed for program specialization algorithms that is more general than the projection-based partial evaluation methods formalised in the author's earlier 'Re-examination' paper. Turchin's 'driving' transformation, known to yield more efficient specialised programs, is in this paper for the first time put on a solid semantic foundation which is not tied to any particular programming language or data structure. The new approach includes both projection-based methods and driving in a natural and simple way, and eases formulating correctness of residual code generation.

[14] **N. D. Jones, C. Gomard, P. Sestoft,** *Partial Evaluation and Automatic Program Generation,* Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).

Summary: This book provides a broad coverage of basic and advanced topics in partial evaluation. A wide spectrum of languages are treated including imperative, functional (first-order, higher-order), and logic programming languages.

[15] **N. D. Jones and F. Nielson,** Abstract Interpretation: a Semantics-Based Tool for Program Analysis, in *Handbook of Logic in Computer Science,* Oxford University Press, 1994. 121 pages.

Summary: This is a broad overview of Abstract Interpretation, to be a large chapter (around 100 pages) in the above-mentioned handbook. It consists of three main parts: an Introduction with motivation and Descriptions of the main methods used in the field; a mathematical development of the logical relations approach with several applications;

and short descriptions of a broad spectrum of Semantics-Based Program Analyses.

[16] **J.W. Lloyd and J.C. Shepherdson,** Partial evaluation in logic programming, in *Journal of Logic Programming*, 11(3,4) 1994.

[17] **K. Marriot and H. Søndergaard,** Prolog program transformation by the introduction of difference lists, *Proc. Int. Computer Science Conf. 88,* 206-213, IEEE, Hong Kong, 1988.

[18] **K. Marriot and H. Søndergaard and N.D. Jones,** Denotational Abstract Interpretation of Logic Programs, *ACM Transactions on Programming Languages and Systems* (1994). Accepted.

Summary: A framework is established for the static analysis of prolog programs. The framework is quite general and is solidly based in a formal semantics, thus enabling rigorous correctness proofs of various analyses. The methods used include an adaption of logical relations as applied by Nielson and Nielson.

[19] **T.Æ. Mogensen and A. Bondorf,** Logimix: a self-applicable partial evaluator for Prolog, in *Proceedings of LOPSTR 92. Workshops in Computing,* edited by Kung-Kiu Lau and Tim Clement, Springer-Verlag, January 1993. ISBN: 3-540-19806-7.

Summary: A self-applicable partial evaluator for a large subset of full Prolog is presented. Great care is taken to preserve the operational semantics of the partially evaluated programs, including the effects of non-logical predicates and side effects

[20] **M. Proietti and A. Pettorossi,** Unfolding - definition - folding, in this order for avoiding unnecessary variables in logic programs, in *Proceedings of PLILP 91,* LNCS 528, pp. 347-358, 1991

[21] **H. Søndergaard,** An application of abstract interpretation in logic programs: Occur check reduction, in *ESOP Proceedings,* Springer-Verlag Lecture Notes in Computer Science vol. 86, 327-338, 1986.

[22] **M.H. Sørensen, R. Glück, N. D. Jones,** Towards Unifying Partial Evaluation, Deforestation, Supercompilation, and GPC, in *ESOP Proceedings,* Springer-Verlag Lecture Notes in Computer Science, 1994.

[23] **V.F. Turchin**, The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3), pp. 292–325, July 1986.

[24] **P. L. Wadler**, Deforestation: transforming programs to eliminate trees. European Symposium On Programming (ESOP). Lecture Notes in Computer Science 300, pp. 344-358, Nancy, France, Springer-Verlag, 1988.