setting that allows one to express a number of dataflow analyses in a uniform way, just as we have done in the present paper through our "dataflow semantics." The various frameworks differ considerably in: (1) the degree of semantic justification provided, (2) assumptions about underlying semantics, (3) the number of operations that need to be given in an "interpretation," (4) the notion of "safety approximation" as a relation between operations, and (5) general complexity.

Abstract interpretation of logic programs was first mentioned by Melliès [29] who suggested it as a way to formalize mode analysis. An occur check analysis which was formalized as a non-standard semantics was given by Sendersgaard [35]. Some of the techniques used in the present paper can be traced back to that work. This is the case with the principle of performing unification both on call and return so as to facilitate that only local variables need be manipulated at any stage (this was referred to as a principle of "locality").

A framework for the abstract interpretation of logic programs was first given by Melliès [29]. Melliès's semantics is an operational parallel to our "lazy" semantics with the impression that this implies: success patterns are not associated with their corresponding call patterns, so success information is propagated back, not only to the atom that actually called the clause, but to all atoms that unify with the clause's head. The application that had Melliès's interest in particular was mode analysis. Debray [10] subsequently investigated this application in more detail and pointed to a problem in Melliès's application (the so-called aliasing problem, which may manifest itself as either a non-safety or a completeness problem, depending on the particular dataflow analysis).

A framework for the abstract interpretation of logic programs based on a denotational definition of SLD was given by Jones and Sendersgaard [10]. This was the first denotational approach to abstract interpretation of logic programs. The framework allowed even the base (or standard) semantics to be expressed as an instance of the dataflow semantics. This has the advantage of providing a very clear cut between a semantic definition which is precise (unlike our lax and dataflow semantics) and interpretations in which all introduced impression resides. In the present paper we have abandoned this approach only to simplify our presentation. Jones and Sendersgaard used operations "call" and "return" which in the present paper have been replaced by "call" and "return." We find this conceptually clearer.

Kanazawa and Kawamura [25] suggested a framework based on ODL/1 resolution, which essentially is SLD resolution extended with memoing, so as to avoid redundant computation. Braumärkte et al. [2, 4] suggested an AND/OR tree based framework in which an interpretation contains some add operations. This has later been simplified to some extent [5]. Neither approach uses point characterizations for standard or non-standard semantics. We have earlier discussed the relative merits of operational and denotational definitions as basis for the design of dataflow analyses. Readers should compare the two last mentioned approaches with that of this paper, both as regards conceptual complexity and completeness of the various semantic definitions.

The framework used by Winsborough [37, 38] is rather close to ours. In particular, one semantic definition (Winsborough's "total function graph semantics) [38] is almost identical to our base semantics; the difference being that it works with substitutions that are "canonized" to bar variants of a substitution from introducing redundancy (Melliès [30] used the same idea). Jacobs and Lassen [17] have studied abstract interpretation based on similar definitions.

Debray [8] has studied a framework for dataflow analysis with the point of departure that analyses must be efficient. He identifies a property of description domains ("substitution closures") and give a complexity analysis to support the claim that the corresponding class of dataflow analyses can be implemented efficiently. Our groundness analysis falls outside Debray's class, as does any dataflow analysis that attempts to maintain information about possible aliasing.

In other studies of logic program analysis [37], we have found it useful to distinguish between "bottom-up" and "top-down" analysis. This distinction is not clear-cut, but we think of a top-down semantics as one that allows for extraction of information about the SLD tree that corresponds to the execution of a program gives some query. Bottom-up analysis is not based on such a semantics to begin with, but on a T-style semantics, and therefore it cannot provide information about for example calls that will take place at runtime. Bottom-up analysis suffices for several applications, though. It is not only the conceptually simplest of the two, it also allows for efficient derivation of query-independent information about a program.

Without claiming completeness of the list or the attached references, we finally list some applications that fit into our framework, or a similar framework.

- Aliasing analysis [15].
- Compile time garbage collection [4].
- Determinacy analysis [13].
- Floundering analysis for normal logic programs [6].
- Independence analysis for AND-parallelism [11, 13, 35].
- Mode analysis [2, 12, 20, 23, 29, 36].
- Occur check analysis [30].
- Program transformation [26].
- Type inference [4, 18, 20, 24, 25].

8 Discussion

We have called our approach denotational abstract interpretation. Characteristic of our treatment is the use of a language-independent framework and parametric instantiations. In this respect it differs significantly from much of the other work that has been published about abstract interpretation of logic programs. We hope that the present treatment will be seen as a point in favor of the denotational approach developed by P. Nielson, or at least for an approach based on a powerful meta-language such as that of denotational semantics. Nielsen’s approach allows for generality.
at different levels. First, the language of denotational semantics allows for comfortable reasoning at exactly the level of abstraction called for by any particular class of applications, or dataflow analysis. Second, the proof of correctness of a particular dataflow analysis becomes almost trivial, since most of it can be conducted at the level of the meta-language case for all. Finally, most of the theory is independent of any particular programming language, since it is expressed in terms of the meta-language only.

The last point indicates that our work could be redone for related programming language paradigms without too much effort. Related languages include logic programming languages with delay mechanisms ("freeze," "wait," "when," etc.), deductive databases, constraint logic programming languages, and perhaps even concurrent constraint programming languages. Whether this is the case remains to be seen, though. Of course the types of analysis required for those languages may be different from those discussed here, but it would seem that abstract interpretation could be as versatile for those languages as it has proved to be for logic programming.

The basic hypothesis in the present work has been that of P. and R. Cousot: many important dataflow analyses in a programming language $L$ can be understood as approximations of extreme exposts in $L$'s semantics domains. It seems certain that this does not cover all interesting analyses, however, and it would be very useful to understand the limitations of the framework of abstract interpretation in a logic programming context. One problem is that there are many useful properties that are not subfolds, such as "finite" in deductive databases. In such cases our present theory allows for no better solution than approximating the relevant least expost by approximating a greatest expost from above. Another problem is that some of the domains of descriptions that have been suggested for useful dataflow analyses are neither Noetherian nor possess the dual property of being "descending chain finite" [80]. This is the case, for example, for the so-called regular trees used by Pyo and Reddy [83] for type inference. It would therefore seem that methods such as those of Heinze and Jaffar [14] and Pyo and Reddy cannot easily be captured, if at all, unless one considers incorporating some kind of "widening," as discussed by P. and R. Cousot [5].

References


Approaches to Logic Program Development

J. Christopher Hogger
Department of Computing
Imperial College
London - England

Developing logic programs has usually been associated with the use of program transformation and synthesis, driven by wholly deductive procedures. These methods suffer from various methodological and technical weaknesses.

An alternative approach to the problem of defending the integrity of program evolution is to analyze the consequences of making modifications. Again purely deductive methods are technically unsatisfactory, requiring either new logical material to be added to the program being analyzed, or else the use of induction. However, extra reasoning power can be obtained by combining deduction with loop-detection, as is implicit in the use of inference systems such as SL5-resolution.

The lecture will discuss and illustrate these ideas, and relate them to the consistency-checking methods customarily used for analysing database updates in the context of integrity constraints.