SEMANTICS-BASED DATAFLOW ANALYSIS OF LOGIC PROGRAMS

Kim MARRIOTT[†] and Harald SØNDERGAARD[‡]

[†]IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598, USA
[‡]DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 København Ø, Denmark

The increased acceptance of Prolog has motivated widespread interest in the semantics-based dataflow analysis of logic programs and a number of different approaches have been suggested. However, the relationships between these approaches are not clear. The present paper provides a unifying introduction to the approaches by giving novel denotational semantic definitions which capture their essence. In addition, the wide range of analysis tools supported by semantics-based dataflow analysis are discussed.

1 Introduction

Dataflow analysis is an essential component of many programming tools. There are two main uses of dataflow information. One is to identify errors in a program, as performed by program debuggers and type checkers; another is in program transformers such as compilers or partial evaluators, where the analysis determines the applicability of various optimisations.

Dataflow analyses may be very complex and so are often difficult to develop and prove correct. Fortunately the theory of *abstract interpretation* provides some help [1]. It aids the design and verification of a dataflow analysis by formalising the relation between analysis and semantics. The insight behind abstract interpretation is to regard a dataflow analysis as a *non-standard semantics*, that is, one in which the usual domain of values has been replaced by a domain of *descriptions* of values, and where the operators are given a corresponding non-standard interpretation. In this paper we use denotational semantics as our semantic definition formalism. For a detailed account of denotational abstract interpretation see [2].

The increasing acceptance of Prolog as a practical programming language has motivated widespread interest in dataflow analysis of logic programs and especially in abstract interpretation. Dataflow analysis of logic programs differs significantly from the analysis of conventional programs because dataflow is bi-directional (owing to unification) and control flow is more complex (owing to backtracking). Recently a number of approaches to abstract interpretation of logic programs have been suggested [3, 4, 5, 6, 7]. The relationships between these approaches, however, are not clear because the need for rigorous semantic definitions has often been ignored. The importance of such definitions is stressed by the fact that, unlike programs in more conventional languages, logic programs have many different semantics given to them: fixpoint characterisation, operational, and classical or 3-valued logic-based. Indeed these diverse semantics form the bases of different approaches to dataflow analysis seen in the literature.

The aim of this paper is to clarify the relationships between different approaches to abstract interpretation of logic programs. We do this by recasting the approaches in precise denotational definitions. The paper may also serve as an introduction to semantics-based dataflow analysis of logic programs and its applications.

Section 2 contains some preliminary definitions. In Section 3, abstract interpretation is introduced. At this stage, no assumptions are made about the programming language used. In Section 4, we give a number of new denotational definitions for definite clause semantics which capture the essences of various abstract interpretation schemes seen in the literature. We discuss and exemplify the analysis and transformation tools supported by each approach. Section 5 contains a conclusion.

The reader is expected to be familiar with logic programming [8] and denotational semantics [9]. Knowledge of abstract interpretation may be useful but is not assumed, as the necessary definitions are provided. A general introduction to abstract interpretation of declarative languages can be found in [10].

2 Preliminaries

A *preordering* is a binary relation that is reflexive and transitive, and a *partial ordering* is an antisymmetric preordering. A set equipped with a partial ordering is a *poset*. Given a poset $(X, \leq)$ and a subset Y of X, an element $x \in X$ is an *upper bound* for Y iff $y \leq x$ for all $y \in Y$. Dually we may define a *lower bound* for Y. An upper bound x for Y is a *least upper bound* for Y iff, for every upper bound x' for Y, $x \leq x'$. When it exists, we let $\bigsqcup Y$ denote the least upper bound for Y. Dually $\bigsqcap Y$ denotes the greatest lower bound for Y. A poset for which all subsets have a least upper bound and a greatest lower bound is a *complete lattice*. We denote the powerset of a set X by $\wp X$.

We let $X_{\underline{\trianglelefteq}}$ denote the poset induced by the preordering $\underline{\trianglelefteq}$ on the set X. Note that $\underline{\trianglelefteq}$ is not the ordering on the poset but a parameter in its construction. If Z is a poset with ordering $\leq$, we define $\Re Z$ to be $(\wp Z)_{\underline{\sqsubseteq}}$, where the preordering $\sqsubseteq$ is defined by $X \sqsubseteq Y$ iff $\forall x \in X \,.\, \exists y \in Y. x \leq y$. Note that $\Re Z$ consists of equivalence classes and is always a complete lattice. We use the symbol $\Re$ because of the close resemblance with the relational (or Hoare) powerdomain constructor [9].

We make use of many different equivalence relations and often need to choose a representative for a particular equivalence class. This is obtained using the non-deterministic function $rep : \wp Z \to Z$ where $rep\, z$ is a representative of z. The lack of determinism will not affect the well-definedness of any of the subsequent semantics, because of a certain confluence in our definitions: whichever representative *rep* chooses, the overall result is the same. However, *rep* is assumed to have the property of Hilbert's epsilon: different applications of *rep* to a given equivalence class yield the same representative.

Let $(X, \leq)$ and $(Y, \preceq)$ be posets. A function $F : X \to Y$ is *monotonic* iff $x \leq x' \Rightarrow F\,x \preceq F\,x'$ for all $x, x' \in X$. A *fixpoint* for a function $F : X \to X$ is an element $x \in X$ such that $x = F\,x$. If X is a complete lattice and $F : X \to X$ is monotonic then F has a least fixpoint which we denote by $lfp\, F$. Since we only deal with monotonic functions, $X \to Y$ shall denote the set of monotonic functions from X to Y throughout this paper.

3 Abstract Interpretation

Many dataflow analyses may be regarded as mimicking the normal execution of a program by working with *descriptions* of objects rather than the objects themselves. The descriptions are usually chosen so that the mimicking execution is guaranteed to terminate, that is, the execution is finitely approximated. Since most interesting program properties are undecidable, the descriptions computed must, in general, be approximate. However, for the descriptions

to be useful, the approximation must be safe, or conservative. In this section we make precise what it means to safely approximate the normal interpretation.

Following [2] we let the semantics of a programming language be defined in two stages. The first stage maps programs to semantic equations expressed in some metalanguage. The second is an "interpretation" which maps expressions in the metalanguage to their denotations.

Our metalanguage consists of typed lambda expressions. A thorough description of the metalanguage is beyond the scope of the present paper. However we note that the "interesting" types in the language are defined by

$$t ::= p \mid t \to t \mid \Re\, t$$

where $p \in P$, the set of primitive types.

Definition. An *interpretation* (T, E) consists of a *type interpretation* T which is a mapping from types to posets and an *expression interpretation* E which is a mapping from expressions to their denotations. □

An interpretation may be specified by giving the type interpretation for primitive types and the expression interpretation for all constants in the metalanguage; the mappings for other types and expressions are then induced in the standard manner (see [2]). Our metalanguage contains a least fixpoint operator and a least upper bound operator, and for expressions containing these to be well-defined, we may require that certain posets are complete lattices.

By varying the interpretation, one may obtain different semantics from the same semantic equations. The *standard interpretation* gives the usual input/output behaviour of the program while dataflow analyses may be expressed as "non-standard" interpretations. The role of abstract interpretation is to give relationships between the standard and non-standard interpretations which guarantee that the analysis safely approximates the standard semantics. This is done by giving a family of so-called concretization functions $\gamma = \{\gamma_p\}$. If T is the standard and T' the non-standard type interpretation then for each primitive type p, γ_p gives the denotations of elements of $T'p$ in terms of elements of Tp. A sufficient (but not necessary [1]) condition to ensure termination of the dataflow analysis is that the primitive types are interpreted as finite posets.

Definition. Let T, T' be type interpretations and γ a family of (monotonic) functions such that for each primitive type p, $\gamma_p : (T'\,p) \to (T\,p)$. Define the relation $ap[\gamma]_t$ by

$$\begin{array}{ll} u\ ap[\gamma]_p\ v & \text{iff } v \le \gamma_p\, u \\ u\ ap[\gamma]_{t\to t'}\ v & \text{iff } \forall u', v' \,.\, u'\ ap[\gamma]_t\ v' \Rightarrow (u\,u')\ ap[\gamma]_{t'}\ (v\,v') \\ u\ ap[\gamma]_{\Re t}\ v & \text{iff } \forall v' \in rep\ v \,.\, \exists u' \in rep\ u \,.\, u'\ ap[\gamma]_t\ v'. \end{array}$$

Definition. An interpretation (T', E') *γ-approximates* interpretation (T, E) iff

- for each primitive type p, there is a (monotonic) function $\gamma_p : T'\,p \to T\,p$ which is one-one, and
- for each constant c of type t, $(E'\,c)\ ap[\gamma]_t\ (E\,c)$. □

In abstract interpretation it is usual to demand that the range of each function γ_p is a Moore family. This is equivalent to demanding the existence of a so-called abstraction function α [1]. Here, we do not require this restriction, and in practice many dataflow analyses do not satisfy it.

Proposition. If interpretation (T', E') γ-approximates the interpretation (T, E) then, for all closed expressions e of type t in our metalanguage, $(E'\, e)\ ap[\gamma]_t\ (E\, e)$ holds. □

Example 3.1. Assume that the standard interpretation of a primitive type p is $\wp\, Int$. In a non-standard interpretation we can interpret p as the set $D = \{\bot, odd, even, \top\}$ ordered by $d \leq d'$ iff $d = d' \vee d = \bot \vee d' = \top$. The denotation of description $d \in D$ is given by

$$\gamma\, d = \begin{cases} \emptyset & \text{if } d = \bot \\ \{n \in Int \mid n \text{ is odd}\} & \text{if } d = odd \\ \{n \in Int \mid n \text{ is even}\} & \text{if } d = even \\ Int & \text{if } d = \top. \end{cases}$$

The set $\{2, 4\}$ is approximated by the description *even*, the set $\{2, 3\}$ by $\top$, and multiplication of sets of integers is approximated by the function $mult : D \times D \to D$ where

$mult$	$\bot$	odd	$even$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
odd	$\bot$	odd	$even$	$\top$
$even$	$\bot$	$even$	$even$	$even$
$\top$	$\bot$	$\top$	$even$	$\top$

□

Usually a program is mapped to semantic equations specifying only its input/output behaviour. We call such a semantics a *base semantics*. For dataflow analysis purposes, however, we are interested in a semantics that is somewhat more complex than this. The reason is that we are looking for invariants (like "x is always positive") that hold at *program points*. A *collecting semantics* is obtained by extending the base semantics so that for each program, the semantic equations also specify the run-time states associated with each of its program points. A base semantics may then be viewed as a degenerate collecting semantics that has only one program point, namely the "end" of the program.

4 Logic Program Analysis

In this section we give several semantic definitions for definite clause logic programs and relate these to dataflow analyses given in the literature. We consider semantics in terms of successful derivations only. As a consequence, finite failure is not distinguished from nontermination. Our definitions are rigorous: even the issue of variable renaming, so often glossed over, is handled precisely.

We first define the various semantic domains needed. Then we discuss "bottom-up" and "top-down" analyses separately. Informally, a bottom-up analysis mimics the well-known T_P semantics [11], whereas a top-down analysis mimics a Prolog interpreter. We show a simple application and discuss the range of applications more generally. Finally we show how analyses may be done more efficiently.

4.1 Semantic Domains

Let *Atom* denote the set of atoms, *Body* the set of bodies, *Clause* the set of clauses, and *Prog* the set of programs. A *renaming* is a bijective substitution. We are only interested in the set *Sub* of substitutions that are renamings of idempotent substitutions. By restricting our interest to *Sub*, we do not exclude any "useful" substitutions.

We equip $Atom$ with the *standard* preordering $\underline{\lhd}$ defined by $A \underline{\lhd} A'$ iff $\exists \theta \in Sub \,.\, A = \theta\, A'$. The equivalence relation induced by this is denoted by $\sim$. This is "modulo variable renaming" equivalence. $Atom$ is assumed to have a least element $\bot$. We also make use of the *flat* preordering $\preceq$ on $Atom$, defined by $A \preceq A'$ iff $A = \bot \vee A \sim A'$. Since variable names are sometimes unimportant, we shall often use $Atom_{\underline{\lhd}}$ (or $Atom_{\preceq}$), rather than $Atom$.

We equip Sub with the *standard* preordering $\underline{\lhd}$ defined by $\theta \underline{\lhd} \theta'$ iff $\forall A \in Atom \,.\, \theta\, A \underline{\lhd} \theta'\, A$. Again, we denote the induced equivalence relation by $\sim$. This is "modulo variable renaming" equivalence. Sub is assumed to have a least element (corresponding to failure of unification), which, when applied to an atom, yields $\bot$. $Sub_{\underline{\lhd}}$ forms a complete lattice, and this explains our interest in Sub rather than the set of all substitutions. We also make use of the *flat* preordering $\preceq$ on Sub, defined by $\theta \preceq \theta'$ iff $\theta = \bot \vee \theta \sim \theta'$. The standard and flat preorderings induce the same equivalence relation on $Atom$, and on Sub as well.

Our interest in equivalence classes is motivated by the fact that we do not want to distinguish between certain elements. For technical precision we shall usually put "hats" on variables that range over equivalence classes, and the equivalence class of x shall be denoted by $[x]$. In examples, however, we omit square brackets, since these can easily be deduced. When convenient, we regard $\hat{\theta} \in Sub_{\underline{\lhd}}$ as the mapping from $Atom$ to $Atom_{\underline{\lhd}}$ defined by $\hat{\theta}\, A = [(rep\, \hat{\theta})\, A]$. This is appropriate since $\theta, \theta' \in \hat{\theta} \Rightarrow [\theta\, A] = [\theta'\, A]$.

Given atoms A and A', the function mgu yields the equivalence class of most general unifiers, that is, $mgu\, A\, A' = \bigsqcap\{\hat{\theta} \in Sub_{\underline{\lhd}} \mid \hat{\theta}\, A = \hat{\theta}\, A'\}$. We assume that the set of variables Var is partitioned into the sets V_P, V_0, V_1, ... Only variables from the set V_P may appear in a program. For every natural number i, a function $rep_i : (Atom_{\preceq} \cup Atom_{\underline{\lhd}}) \to Atom$ is defined as follows: $rep_i\, \hat{A}$ is an atom $A \in \hat{A}$ containing only variables from V_i. This enables us to handle variable renaming in a simple way.

4.2 Bottom-up Analysis

The following semantics is based on the well-known T_P semantics in which the denotation of a program is the set of atoms which the program "makes true" [11]. Our semantics differs from the T_P semantics in that these atoms need not be ground. More precisely, the denotation of a program is the element of $\Re\, Atom_{\underline{\lhd}}$ all of whose elements are made true.

Definition. The *bottom-up semantics* has domain

$$Sem = \Re\, Atom_{\underline{\lhd}}$$

semantic functions

$$\mathcal{P} : Prog \to Sem$$
$$\mathcal{C} : Clause \to Sem \to Sem$$
$$\mathcal{B} : Body \to Sem \to \Re\, Sub_{\underline{\lhd}}$$

and auxiliary function

$$apply : \Re\, Sub_{\underline{\lhd}} \to Atom \to \Re\, Atom_{\underline{\lhd}}.$$

It is defined by

$$
\begin{array}{ll}
\mathcal{P}\,[\![P]\!] & = S \,\text{where rec}\, S = \bigsqcup\{\mathcal{C}\,[\![C]\!]\,S \mid C \in P\} \\
\mathcal{C}\,[\![A \leftarrow B]\!]\,S & = \mathit{apply}\,(\mathcal{B}\,[\![B]\!]\,S)\,A \\
\mathcal{B}\,[\![A_1, \ldots, A_n]\!]\,S & = [\{\sqcap\{\hat{\theta}_1, \ldots, \hat{\theta}_n\} \mid \hat{\theta}_1 \in \Theta_1 \wedge \ldots \wedge \hat{\theta}_n \in \Theta_n\}] \\
& \quad \text{where}\, \Theta_i = \{\mathit{mgu}\, A_i\, (\mathit{rep}_i\, \hat{A})) \mid \hat{A} \in S\} \\
\mathit{apply}\, \hat{\Theta}\, A & = [\{\hat{\theta}\, A \mid \hat{\theta} \in \mathit{rep}\, \hat{\Theta}\}].
\end{array}
$$
□

The denotation of a body is a set of (equivalence classes of) substitutions, whereas that of a clause is a set of (equivalence classes of) atoms. The reason is that substitutions are carriers of binding information between atoms, and it is necessary to refer to the particular variables in the clause. Seen from outside, however, a clause's variables have no impact: they are universally quantified by the clause.

We now give an example program and its denotation. The program is somewhat contrived — it has been chosen so as to illuminate certain aspects of the semantics.

Example 4.1. The program

$$
\begin{array}{l}
p(x) \leftarrow p(x) \\
p(x) \leftarrow q(x) \\
q(x) \\
q(a)
\end{array}
$$

denotes $\{p(x), q(x)\}$. Note that $q(a)$ is not in the denotation as we use $\Re\, \mathit{Atom}_{\underline{\lhd}}$, which is "downwards closed", so $q(x)$ subsumes $q(a)$. Similarly $p(a)$ is not in the denotation. □

The bottom-up semantics is a base semantics. In some applications it is useful to associate program points with clauses in a program and for each of these, to record the clause instances used in successful derivations.

We now sketch an example dataflow analysis for type inference based on the bottom-up semantics. In our setting, type inference may be viewed as finding a description for the set of atoms that a program makes true. Our descriptions are based on "Sato-Tamaki k-atoms", where k is a natural number [12]. These atoms have depth no greater than k and different variables as level k subterms, and the idea is that such an atom "describes" its set of instances. We denote the set of Sato-Tamaki k-atoms by Sat^k.

The relation between Sat^k and Atom is simply given by the inclusion function $\gamma : \mathit{Sat}^k \hookrightarrow \mathit{Atom}$. We redefine

$$
\begin{array}{ll}
\mathit{Sem} & = \Re\, \mathit{Sat}^k_{\underline{\lhd}} \\
\mathit{apply}\, \hat{\Theta}\, A & = [\{[\mathit{prune}\, (\mathit{rep}\, (\hat{\theta}\, A))] \mid \hat{\theta} \in \mathit{rep}\, \hat{\Theta}\}]
\end{array}
$$

where $\mathit{prune}\, A$ is the atom obtained by replacing every level k subterm of A by a fresh variable.

Our type inference is now defined using exactly the same set of semantic equations as in the bottom-up semantics. Calling the resulting semantic function for programs $\mathcal{P}'$, we have by the Proposition of Section 3 that $\mathcal{P}'\, \mathit{ap}\, \mathcal{P}$ holds. That is, the type inference is correct with respect to the underlying semantics. Finally our definition can be seen to give a terminating dataflow analysis as $\mathit{Sat}^k_{\underline{\lhd}}$ is finite.

Example 4.2. Under $\mathcal{P}'$ the program

$$member(u, u.v)$$
$$member(u, x.v) \leftarrow member(u, v)$$

denotes $\{member(u, y.z)\}$. This information tells us that whenever the second clause is used in a successful derivation, v is bound to a term of form $y.z$. The program can therefore be specialised to

$$member(u, u.v)$$
$$member(u, x.y.z) \leftarrow member(u, y.z)$$

without changing its denotation [13]. □

The bottom-up base semantics is similar to that given in [7]. A difference is that the semantics in [7] is more complex because it applies to the broader class of *normal* logic programs [8] and so must deal with negation. However, the semantics in [7] is given in terms of ground atoms only and therefore does not allow as expressive description schemes as does the present.

The main application area of bottom-up analysis is type inference. Type information is useful for a number of tools, notably for type checking [14, 15] and program specialisation as in Example 4.2 [12, 13].

4.3 Top-Down Analysis

In this section we discuss the notion of top-down analysis and compare it with bottom-up analysis. We particularly stress the difference in application areas. Basically the difference is that whereas a bottom-up analysis gives success pattern information, a top-down analysis can also give call pattern information.

In the top-down semantics, the denotation of a program P is a mapping from atoms to sets of atoms. The atom A is mapped to the instances $\theta\, A$ such that there is an SLD-derivation from P and A with answer θ [8].

For two reasons, the top-down semantics is less abstract than the bottom-up. Firstly, it assumes a left-to-right computation rule. This is an advantage for certain Prolog analyses, because it allows the analyses to exploit the fact that the computation rule is known. Secondly, it uses the domain $\Re\, Atom_{\preceq}$ rather than $\Re\, Atom_{\underline{\lhd}}$. This reflects that, in the bottom-up case, we did not care about recording *instances* of atoms already recorded. Now, however, we want to record *all* atoms that appear during a computation, not just the maximal ones. For example, if it is discovered that a clause head $p(x)$ is sometimes instantiated to $p(x)$ and sometimes to $p(a)$, then we want to record both instances, not just the more general $p(x)$. The reason is that the applications we have in mind are the kind of analyses typically used in compilers. A compiler needs detailed information about call patterns and might generate incorrect code if the fact (in this example) that x is sometimes bound to a ground term was concealed.

Definition. The *top-down semantics* has domain

$$Sem = Atom_{\preceq} \rightarrow \Re\, Atom_{\preceq}$$

and semantic functions

$\mathcal{P} : \mathit{Prog} \to \mathit{Sem}$
$\mathcal{C} : \mathit{Clause} \to \mathit{Sem} \to \mathit{Sem}$
$\mathcal{B} : \mathit{Body} \to \mathit{Sem} \to \mathit{Sub}_{\preceq} \to \Re\, \mathit{Sub}_{\preceq}.$

It is defined by

$$\begin{aligned}
\mathcal{P}\,[\![P]\!] &= S \,\text{where rec}\, S = \bigsqcup\{\mathcal{C}\,[\![C]\!]\,S \mid C \in P\} \\
\mathcal{C}\,[\![A \leftarrow B]\!]\,S\,\hat{A} &= \mathit{apply}\,(\mathcal{B}\,[\![B]\!]\,S\,(\mathit{mgu}\,A\,(\mathit{rep}_0\,\hat{A})))\,A \\
\mathcal{B}\,[\![\,]\!]\,S\,\hat{\theta} &= [\{\hat{\theta}\}] \\
\mathcal{B}\,[\![A_1 \ldots A_i]\!]\,S\,\hat{\theta} &= \bigsqcup\{\,[\{\hat{\theta}' \sqcap \mathit{mgu}\,A_i\,(\mathit{rep}_i\,\hat{A}) \mid \hat{A} \in S\,(\hat{\theta}'\,A_i)\}] \\
&\qquad \mid \hat{\theta}' \in \mathit{rep}\,(\mathcal{B}\,[\![A_1 \ldots A_{i-1}]\!]\,S\,\hat{\theta}\,)\}.
\end{aligned}$$

□

The auxiliary function $\mathit{apply} : \Re\, \mathit{Sub}_{\preceq} \to \mathit{Atom} \to \Re\, \mathit{Atom}_{\preceq}$ is defined as before.

Example 4.3. The program from Example 4.1 denotes

$$\begin{aligned}
\{\quad & p(x) \mapsto \{p(x), p(a)\}, \\
& p(a) \mapsto \{p(a)\}, \\
& q(x) \mapsto \{q(x), q(a)\}, \\
& q(a) \mapsto \{q(a)\} \qquad\quad \}.
\end{aligned}$$

□

Usually in dataflow analysis based on top-down semantics one uses a collecting semantics that records, for each clause, those instances of its head that were used in derivations from a query. This information cannot be obtained from a collecting semantics based on the bottom-up semantics. The reason is that the bottom-up semantics only contains information about atoms that succeed whereas the top-down semantics also contains information about all calls regardless of whether they succeed, finitely fail or do not terminate.

Typically dataflow analyses based on the top-down approach are used in compilers to allow optimisation of the target code for clause invocations. Analyses that have been developed in an abstract interpretation framework include mode analysis [3, 15, 16], live variable analysis [15], determinacy analysis [17], and occur check analysis [18].

Mode analysis aims at finding a program's call patterns. Many Prolog compilers expect a programmer to annotate programs with mode information, like "in any call of $p(x)$, x will be bound to a ground term," to facilitate the generation of better target programs. Programmers, however, can easily provide wrong mode information, leading to incorrect target programs, so the information is better provided by an automatic (correct) analysis.

Live variable analysis helps a compiler generate target programs with better storage management, and determinacy analysis can help it in terms of both efficiency (creating fewer backtrack points) and better storage handling in target programs.

For efficiency reasons, many Prolog systems omit occur checks during unification. It is well-known that this makes the system unsound as a theorem prover. Occur check analysis determines cases where unification can safely be performed without occur checks, which may help a compiler generate efficient programs without sacrificing soundness.

The information discussed here is useful not only for compilers, but also for other program transformers, like partial evaluators [19] and data structure transformers [20], as well as for parallelizers that expose the potential parallelism in logic programs [21, 22].

4.4 Efficient Analysis

The semantic definitions given so far are abstract and somewhat removed from a practical implementation. To make the transition from a dataflow semantics to its implementation easier, we may wish to "bend" the semantics towards a particular implementation strategy. To exemplify this we give a "query-directed" semantics similar to the top-down semantics but which captures the widely used implementation technique of *tabulation*.

A naive implementation of the top-down semantics as the basis for abstract interpretation has one drawback: it is not query-directed. That is, the analysis pertaining to a particular query is only obtained after performing the analysis for *all* queries. The idea behind the query-directed semantics is to remove this inefficiency: only the denotations actually needed for a particular query are calculated (without loss of generality we assume that a query is an atom). This is done by simultaneously computing the set of atoms which may be called *and* the denotation of these atoms.

This technique of making a semantic definition input-directed is quite general. Suppose we have a fixpoint characterisation $F = \mathit{lfp}\,\mathcal{F}$ of a function $F : X \to Y$, but that we are only interested in knowing $F\,x$ for a particular (or a few) $x \in X$. Rather than computing $\mathit{lfp}\,\mathcal{F}$ and then applying this to x, the idea is to "push" x into $\mathcal{F}$ to obtain a new functional $\mathcal{F}'$ such that $(\mathit{lfp}\,\mathcal{F}')\,x = (\mathit{lfp}\,\mathcal{F})\,x$ and $\mathit{lfp}\,\mathcal{F}'$ is more efficiently computable than $\mathit{lfp}\,\mathcal{F}$.

Definition. The *query-directed semantics* has domains

$$\begin{aligned}\mathit{Use} &= \Re\,\mathit{Atom}_{\preceq}\\ \mathit{Sem} &= \mathit{Atom}_{\preceq} \to \Re\,\mathit{Atom}_{\preceq}\\ \mathit{Out} &= \mathit{Use} \times \mathit{Sem}\end{aligned}$$

semantic functions

$$\begin{aligned}&\mathcal{P} : \mathit{Prog} \to \mathit{Use} \to \mathit{Sem}\\ &\mathcal{C} : \mathit{Clause} \to \mathit{Out} \to \mathit{Out}\\ &\mathcal{B} : \mathit{Body} \to \mathit{Sem} \to \mathit{Sub}_{\preceq} \to \mathit{Use} \times \Re\,\mathit{Sub}_{\preceq}\end{aligned}$$

and auxiliary function

$$\mathit{entry} : \mathit{Atom} \to \Re\,\mathit{Sub}_{\preceq} \to \mathit{Sem}.$$

It is defined by

$$\begin{aligned}&\mathcal{P}\,[\![P]\!]\,\hat{U} = S\\ &\quad \text{where rec}\,(\hat{U}', S) = \bigsqcup\{\mathcal{C}\,[\![C]\!]\,(\hat{U} \sqcup \hat{U}', S) \mid C \in P\}\\ &\mathcal{C}\,[\![A \leftarrow B]\!]\,(\hat{U}, S) =\\ &\quad \bigsqcup\{\,(\hat{U}', \mathit{entry}\,A'\,\Theta)\\ &\qquad \text{where}\,(\hat{U}', \Theta) = \mathcal{B}\,[\![B]\!]\,S\,(\mathit{mgu}\,A\,A')\\ &\qquad \text{where}\,A' = \mathit{rep}_0\,\hat{A}\\ &\qquad \mid \hat{A} \in \mathit{rep}\,\hat{U}\}\\ &\mathcal{B}\,[\![\,]\!]\,S\,\hat{\theta} = (\bot_{\mathit{Use}}, [\{\hat{\theta}\}])\\ &\mathcal{B}\,[\![A_1 \ldots A_i]\!]\,S\,\hat{\theta} = (\,\hat{U} \sqcup (\mathit{apply}\,\hat{\Theta}\,A), \hat{\Theta}'\,)\\ &\quad \text{where}\,\hat{\Theta}' =\\ &\quad \bigsqcup\{[\{\hat{\theta}' \sqcap \mathit{mgu}\,A_i\,(\mathit{rep}_i\,\hat{A}) \mid \hat{A} \in S\,(\hat{\theta}' A_i)\}] \mid \hat{\theta}' \in \mathit{rep}\,\hat{\Theta}\}\\ &\qquad \text{where}\,(\hat{U}, \hat{\Theta}) = \mathcal{B}\,[\![A_1 \ldots A_{i-1}]\!]\,S\,\hat{\theta}\\ &\mathit{entry}\,A\,\hat{\Theta}\,\hat{A} = \text{if}\,A \in \hat{A}\,\text{then}\,\mathit{apply}\,\hat{\Theta}\,A\,\text{else}\,[\emptyset].\end{aligned}$$

□

Example 4.4. The denotation of the program given in Example 4.1 maps the initial query set $\{p(a)\}$ to

$$\begin{array}{ll}\{ & p(x) \mapsto \bot, \\ & p(a) \mapsto \{p(a)\}, \\ & q(x) \mapsto \bot, \\ & q(a) \mapsto \{q(a)\} \quad \}.\end{array}$$

The queries $p(x)$ and $q(x)$ are both mapped to $\bot$, since their successful instances are not needed to find those of $p(a)$. □

The query-directed semantics may be used as a basis for the tabulation methods of [5, 6, 16]. In these a table containing an entry for each predicate symbol is constructed. Each entry contains a description of the calls to the predicate, and for each call, a description of the answers. The query-directed semantics also bears close resemblance to the *minimal function graph* semantics of [22, 23]. Finally, the transformation from a non-query-directed definition into a query-directed definition may be viewed as a generalisation of the magic-set transformation used in deductive databases [24].

5 Conclusion

The purpose of this paper has been to explain dataflow analysis as non-standard interpretation, and to present in a uniform way different approaches to abstract interpretation of logic programs. To do this we have given a series of denotational definitions that capture these approaches and provide a rigorous basis for comparison.

The reader may wonder why so many different semantic bases for dataflow analysis of logic programs have been developed. We feel that part of the reason is that different kinds of analysis call for different degrees of "abstraction" of the underlying semantics. A semantics $\mathcal{P}$ is *as abstract as* $\mathcal{P}'$ iff $\mathcal{P}'[\![P]\!] = \mathcal{P}'[\![P']\!] \Rightarrow \mathcal{P}[\![P]\!] = \mathcal{P}[\![P']\!]$ for all programs P and P'. A more abstract semantics usually has a simpler definition, and so will allow for a simply designed dataflow analysis. The semantics, however, should not be so abstract that information useful in the analysis is factored out. Knowledge about the computation rule, for instance, can make information about call patterns more precise.

Our bottom-up semantics is more abstract than our top-down semantics. This is because the latter takes into account a left-to-right computation rule, and because it uses the flat ordering in the generation of the lifted domains, rather than the standard ordering. An even less abstract top-down semantics might keep a record of the derivation history. Such a semantics would be useful for analysis of memory requirements or determinacy.

Acknowledgements

We would like to thank Graeme Port and an anonymous referee for their constructive comments about draft versions of the paper. The paper was written at the Department of Computer Science, University of Melbourne. The first author was supported by an ARGS grant given to that department's MIP project. The second author was supported in part by the Danish Research Academy.

References

[1] Cousot, P. & R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, *Proc. 4th Ann. ACM Symp. Principles of Programming Languages*, Los Angeles, California (1977) 238–252.

[2] Nielson, F., Strictness analysis and denotational abstract interpretation, *Information and Computation* **76**, 1 (1988) 29–92.

[3] Mellish, C., Abstract interpretation of Prolog programs, *Abstract Interpretation of Declarative Languages* (ed. S. Abramsky & C. Hankin), Ellis Horwood (1987) 181–198.

[4] Jones, N. & H. Søndergaard, A semantics-based framework for the abstract interpretation of Prolog, *Abstract Interpretation of Declarative Languages* (ed. S. Abramsky & C. Hankin), Ellis Horwood (1987) 123–142.

[5] Kanamori, T. & T. Kawamura, Analyzing Success Patterns of Logic Programs by Abstract Hybrid Interpretation, ICOT TR-279, ICOT, Tokyo, Japan (1987).

[6] Bruynooghe, M., A Framework for the Abstract Interpretation of Logic Programs, Report CW 62, Dept. of Computer Science, University of Leuven, Belgium (1987).

[7] Marriott, K. & H. Søndergaard, Bottom-up abstract interpretation of logic programs, *Proc. 5th Int. Conf. Symp. Logic Programming* (ed. R. Kowalski & K. Bowen), MIT Press (1988) 733–748.

[8] Lloyd, J., *Foundations of Logic Programming*, Springer-Verlag (1987).

[9] Schmidt, D., *Denotational Semantics*, Allyn & Bacon (1986).

[10] Abramsky, S. & C. Hankin, An introduction to abstract interpretation, *Abstract Interpretation of Declarative Languages* (ed. S. Abramsky & C. Hankin), Ellis Horwood (1987) 9–31.

[11] Apt, K. & M. van Emden, Contributions to the theory of logic programming, *JACM* **29** (1982) 841–862.

[12] Sato, T. & H. Tamaki, Enumeration of success patterns in logic programs, *Theoretical Computer Science* **34** (1984) 227–240.

[13] Marriott, K., L. Naish & J.-L. Lassez, Most specific logic programs, *Proc. 5th Int. Conf. Symp. Logic Programming* (ed. R. Kowalski & K. Bowen), MIT Press (1988) 909–923.

[14] Kanamori, T. & K. Horiuchi, Type inference in Prolog and its application, *Proc. 9th Int. Joint Conf. Artificial Intelligence* (ed. A. Joshi), Los Angeles, California (1985) 704–707.

[15] Bruynooghe, M. *et al.*, Abstract interpretation: towards the global optimization of Prolog programs, *Proc. 4th Int. Symp. Logic Programming*, San Francisco, California (1987) 192–204.

[16] Debray, S. & D. S. Warren, Automatic mode inference of logic programs, *Journal of Logic Programming* **5**, 3 (1988) 207–229.

[17] Debray, S. & D. S. Warren, Detection and optimization of functional computations in Prolog, *Proc. 3rd Int. Conf. Logic Programming* (ed. E. Shapiro), LNCS 225, Springer-Verlag (1986) 490–504.

[18] Søndergaard, H., An application of abstract interpretation of logic programs: occur check reduction, *Proc. ESOP 86* (ed. B. Robinet & R. Wilhelm), LNCS 213, Springer-Verlag (1986) 327–338.

[19] Gallagher, J., M. Codish & E. Shapiro, Specialisation of Prolog and FCP programs using abstract interpretation, *New Generation Computing* **6** (1988) 159–186.

[20] Marriott, K. & H. Søndergaard, Prolog program transformation by introduction of difference-lists, *Proc. Int. Computer Science Conf. '88*, IEEE Computer Society, Hong Kong (1988) 206–213.

[21] Debray, S., Static analysis of parallel logic programs, *Proc. 5th Int. Conf. Symp. Logic Programming* (ed. R. Kowalski & K. Bowen), MIT Press (1988) 711–732.

[22] Winsborough, W., Automatic, Transparent Parallelization of Logic Programs at Compile Time, TR 88-14, Dept. of Computer Science, University of Chicago, Illinois (1988).

[23] Jones, N. & A. Mycroft, Dataflow analysis of applicative programs using minimal function graphs, *Proc. 13th Ann. ACM Symp. Principles of Programming Languages*, St. Petersburg, Florida (1986) 296–306.

[24] Bancilhon, F. *et al.*, Magic sets and other strange ways to implement logic programs, *Proc. 5th ACM Symp. Principles of Database Systems*, Cambridge, Massachusetts (1986) 1–15.