Constraint Databases and Program Analysis using Abstract Interpretation

David Toman

Department of Computer Science, University of Toronto
Toronto, Ontario, Canada M5S 1A4
david@cs.toronto.edu

Abstract. In this paper we discuss a connection between two seemingly distant research areas in computer science: constraint databases and abstract interpretation. We show that while the goals of research in the respective communities are different, the used techniques are often based on similar common foundations. We substantiate this claim by showing that abstract interpretation of a standard (Algol-like) programming language with respect to its natural semantics can be equivalently thought of as querying a constraint deductive database. On the other hand the constraint database community can often benefit from the sophisticated techniques developed for computing abstract properties of programs, e.g., for query termination analysis or approximate query evaluation.

1 Introduction

Abstract Interpretation provides a canonical approach to the analysis of data-flow properties of programs [6]. It is based on semantic approximation: the domain of *concrete values* manipulated by the analyzed program is replaced by a domain of *abstract values*—descriptions of whole sets of concrete values. In addition, each elementary operation defined by the concrete semantics of the program is approximated by an operation on the domain of abstract values. This way the execution of the program with respect to the abstract domain produces an approximation of the execution with respect to the original domain of concrete values. A careful choice of the abstract domain often provides a guarantee of finiteness for such analysis.

In this paper we approach this problem from slightly different point of view: we use constraints—quantifier free formulas in a suitable theory—as abstractions of the *semantic actions* prescribed by the semantics. An immediate consequence of this choice is that the abstract values themselves can be represented using the constraints. We also choose a natural big-step operational semantics to describe the meaning of programs. This way we can think about the abstract interpretation of a program as of a deductive query evaluated over a constraint database built out of the source program by instantiating the natural semantic rules.

The main difference between the standard approaches and our proposal can be best seen in the way sequential composition is modeled: in the standard approach sequential composition in the syntax amounts to a function composition on the corresponding abstract domain. In our proposal the sequential composition is modeled by a conjunction and subsequent quantifier elimination applied on the constraints that approximate the original semantic functions involved in the composition. This is also the main difference from the *relational* abstract interpretations proposed in, e.g., [9, 17, 18, 19, 28], where constraints are often used to capture the static abstract relationships among variables rather then the dynamic behavior of the analyzed program. Note also that we are interested in analyzing general programming languages rather than (constraint) logic programs [3, 4]. There are several advantages of our approach:

- The constraint approach hides the distinction between the *forward* and the *backward* forms [8] of abstract interpretation. This is an immediate consequence of approximating the semantic functions by constraints (relations) and the bidirectional character of logic programs.
- The semantic rules instantiated with the actual text of the analyzed program yield a set of constraint Datalog clauses. Thus we can automatically generate an executable specification of an abstract interpreter from the source text of the given program.
- The choice of abstract values corresponds to the choice of an appropriate constraint class [23]. In addition, the choice of a constraint-compact class [31] guarantees finiteness of the analysis.
- The use of database technology, e.g., query optimization techniques [20, 32] or goal oriented query evaluation procedures [31], allows us to analyze relatively large programs. Indeed, the database techniques are inherently targeted towards fairly large data sets.

However, there are drawbacks as well: the common deductive database technology does not use special techniques and evaluation strategies developed for abstract interpretation, e.g., techniques for convergence speedup using *widening*. The hope is that not only these techniques can be easily incorporated in the current query evaluation algorithms, but also that they can be used in a more general applications of the constraint databases.

The main contribution of this paper lies in establishing a new connection between constraint deductive databases and abstract interpretation based on approximating the semantic actions by constraints. This connection allows us to analyze programs with respect to their semantics using techniques designed for query evaluation over constraint databases. Moreover, there is an additional benefit in showing such a connection between these two research areas:

- the use of database technology may allow the abstract interpretation techniques to be used for attacking real-life problems and to build tools for efficient program analysis based on this technology. This approach may nicely complement many of the ad-hoc database based program development tools, popular in the area of software engineering.
- the constraint database community may be able to adopt some of the sophisticated techniques developed in the area of abstract interpretation and

$$B ::= E \leq E$$
$$E ::= N \mid I \mid E + E \mid -E$$
$$C ::= I := E \mid C;C \mid \texttt{skip} \mid \texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C\ \texttt{fi} \mid \texttt{while}\ B\ \texttt{do}\ C\ \texttt{od}$$

where N stands for integer numerals and I for variable identifiers.

Fig. 1. Syntax of a Simple Imperative Language.

theory of programming languages in general, e.g., for termination analysis of queries or for finding approximate solutions. Especially the second area—finding approximate answers to queries—may provide a very fruitful addition to theory of constraint databases.

The rest of the paper is organized as follows: Section 2 provides the necessary background definitions of natural semantics, abstract interpretation, and constraint databases. Section 3 describes the connection between abstract interpretation and constraint databases. Section 4 shows simple examples of abstract interpretation based on the constraint database technology. Section 5 discusses related work and other possibilities of application of our technique in the area of abstract interpretation. Section 6 concludes the paper with comments on future research directions.

2 Background

In this section we briefly review the background notions of abstract interpretation and constraint deductive databases. To facilitate examples in this paper we define a natural big-step operational semantics for a simple imperative Algol-like language (we follow an introductory example in [12]). The syntax of this language is summarized by the grammar in Figure 1. Then we introduce the basic ideas of the standard abstract interpretation. In the end of the section we introduce deductive constraint databases and show a few elementary properties of queries over such databases.

We give only a very brief and high-level descriptions; for more detailed exposition we provide references to literature covering each of the particular topics in detail. Also, we do not want to bury a quite simple and natural idea in heaps of cumbersome notation that would be necessary to prove all the claims formally.

2.1 Natural Semantics

We define the concrete semantics of our simple imperative language using a finite set of *Natural Semantics* rules [12] of the form

$$\frac{\rho_1 \vdash e_1 \Downarrow v_1 \qquad \ldots \qquad \rho_k \vdash e_k \Downarrow v_k}{\rho \vdash op(e_1, \ldots, e_k) \Downarrow f(v_1, \ldots, v_k)}$$

$$\frac{\rho_0 \vdash E \Downarrow v}{\rho_0 \vdash I := E \Downarrow \rho_0[I \mapsto v]}$$

$$\frac{}{\rho_0 \vdash \mathtt{skip} \Downarrow \rho_0}$$

$$\frac{\rho_0 \vdash C_1 \Downarrow \rho_1 \quad \rho_1 \vdash C_2 \Downarrow \rho_2}{\rho_0 \vdash C_1; C_2 \Downarrow \rho_2}$$

$$\frac{\rho_0 \vdash B \Downarrow \mathtt{tt} \quad \rho_0 \vdash C_1 \Downarrow \rho_1}{\rho_0 \vdash \mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2\ \mathtt{fi} \Downarrow \rho_1} \qquad \frac{\rho_0 \vdash B \Downarrow \mathtt{ff} \quad \rho_0 \vdash C_2 \Downarrow \rho_2}{\rho_0 \vdash \mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2\ \mathtt{fi} \Downarrow \rho_2}$$

$$\frac{\rho_0 \vdash B \Downarrow \mathtt{ff}}{\rho_0 \vdash \mathtt{while}\ B\ \mathtt{do}\ C\ \mathtt{od} \Downarrow \rho_0} \qquad \frac{\rho_0 \vdash B \Downarrow \mathtt{tt} \quad \rho_0 \vdash C \Downarrow \rho_1 \quad \rho_1 \vdash \mathtt{while}\ B\ \mathtt{do}\ C\ \mathtt{od} \Downarrow \rho_2}{\rho_0 \vdash \mathtt{while}\ B\ \mathtt{do}\ C\ \mathtt{od} \Downarrow \rho_2}$$

The *command* part of the natural semantics of the simple imperative language from Figure 1 is defined as a transformation of store vectors—maps from variable names to integers. The expression $\rho_0 \vdash P \Downarrow \rho_1$ denotes the fact that executing P transforms the store from ρ_0 to ρ_1.

Fig. 2. Natural Semantics of the Simple Imperative Language.

where *op* is a syntactic element of the language (a term of arity k) and f is the corresponding semantic function. The assertions $\rho \vdash e \Downarrow v$ denote the fact that *executing* the expression e in the environment ρ yields the value v. We also assume that ρ_i depends only on the original ρ and the values v_j for $j < i$. The rules have to be consistent with the syntax of the language. The main idea behind the natural semantics is as follows: the rules define the meaning of every syntactic expression as a function of meanings of its (not necessarily proper) subexpressions. Execution of a program then corresponds to the construction of a derivation tree using the natural semantic rules. Figure 2 gives a natural semantics to the *command* fragment our simple imperative language. The semantics of the *expression* and *bool* components of the language can be defined analogously. Note that the semantic rules do not have to be well founded; indeed the second rule defining the `while` loop uses an assumption syntactically identical to its conclusion. The trouble with such a definition is that the meaning of the `while` loop is defined in terms of itself. Domain theory [12, 24] allows us to break this loop by defining the meaning of such a construction as a limit of its approximations. The existence and choice of the appropriate limit is discussed in length in numerous books on semantics of programming languages, e.g., [12, 24]. In general, we can compute the semantic value of a `while` loop as the (least) fixpoint obtained by unfolding the corresponding natural semantic rules. Note that given a program in our simple imperative language the instances of the natural semantic rules derived from this program yield a finite set of Horn clauses [12, 25]. This observation leads immediately to the definition of a simple interpreter for the language as a set of Horn clauses, e.g., in Prolog-II[1].

Collection Semantics. Program analysis is often interested in all the pairs of input/output values (ρ, v) associated with a particular syntactic expression (command) during the execution of the analyzed program. We can *collect* all these input/output pairs from the generated semantic tree in a form of a relation. Moreover, because the programs are finite, we can compute such a relation for every subexpression of the analyzed program. The result is often called the *collecting semantics* of the program.

2.2 Abstract Interpretation

Constructing a derivation tree immediately using the natural semantic provides no help in discovering properties of a given program: the concrete semantics completely characterizes the program and thus is not computable in general. However, if we map the *actual values* involved in the computation to a domain of *abstract values*—descriptions of sets of concrete values— with simple enough structure, we might be able to derive some dynamic properties of programs while guaranteeing finiteness of the analysis.

A common choice of the abstract domain is a (complete) lattice; it also often satisfies the finite ascending chain condition. The connection between the domain of concrete and abstract values is usually formalized by two maps: the *abstraction* (α) and the *concretization* (γ) that form an adjoint pair of functions [6]. That is, for every set of concrete values x, $x \subseteq \gamma(\alpha(x))$, and for every abstract value y, $y = \alpha(\gamma(y))$. This condition is the basis for showing that

1. the "execution" of a given program with respect to its natural semantics on the domain of abstract values is sound provided all the abstractions of the individual semantic rules are sound, and
2. the result obtained by abstract interpretation is the tightest possible approximation of the actual result with respect to the ordering of the abstract values.

The finite ascending chain condition is used to guarantee finiteness such analysis. For details see [6, 7, 8].

2.3 Constraint Databases

It is not surprising that constraints have been used in place of the abstract values [8, 9, 21, 25]. They provide a natural and compact descriptions of properties of values. Moreover, they are naturally equipped with the appropriate boolean

[1] First-order programming languages can be encoded using standard Prolog terms. However, higher-order languages may require co-inductive data types to represent closures [25]. These can be conveniently encoded by rational trees of Prolog-II.

structure. However, the semantic operations on the constraints prescribed by the semantics are commonly defined as external functions operating on the constraint class [9]. In our approach we use constraints to approximate the *semantic functions* defined by the natural semantics. This way the whole abstract interpretation can be reduced to a deductive query over a constraint database built out of the analyzed program.

In the remainder of this section we give a definition of constraint databases and deductive queries over such databases. First we define properties of an abstract constraint class—the domain of data stored in the constraint databases:

Definition 2.1 Let $\mathcal{C}_0$ be a finite set of satisfiable constraints. A *Constraint class* generated by $\mathcal{C}_0$ is the least set of constraints $\mathcal{C}$ containing $\mathcal{C}_0$, closed under conjunction, quantifier elimination, and renaming, that is equipped with the following computable operations:

Constraint Conjunction $\wedge^{\mathcal{C}} : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C} \cup \{\perp\}$ that for every pair of constraints C_1, C_2 computes the *conjunction* $C_1 \wedge C_2$ if the conjunction is satisfiable; otherwise returns $\perp$.

Constraint Projection $\exists^{\mathcal{C}} : V \times \mathcal{C} \rightarrow 2^{\mathcal{C}}$ that for every constraint C and every set $V \subset FV(C)$ computes the set $\{C_i\}$ (a *projection* of C to V) that satisfies the condition

$$\left(\bigvee_{C_i \in \exists^{\mathcal{C}}(V,C)} C_i\right) \equiv \exists x_1. \cdots \exists x_l.C$$

where $x_j \in FV(C) - V$ for $0 < j \leq l$. The function is well defined and always returns a *finite* subset of $\mathcal{C}$.

Constraint Subsumption $\leq^{\mathcal{C}} : \mathcal{C} \times \mathcal{C} \rightarrow$ bool that satisfies the condition

$$C_1 \leq^{\mathcal{C}} C_2 \text{ implies } C_1 \supset C_2$$

Finite sets of elements of $\mathcal{C}$ are used to represent possibly infinite relations stored in the constraint database [15]. Query evaluation over such a database is based on the operations defined on the constraint class: the first two operations are the equivalents of relational algebra join and projection. The last operation, the constraint subsumption, is used for duplicate elimination and is the key to proving termination of query evaluation algorithms. Note that the relation induced by $\leq^{\mathcal{C}}$ can be *weaker* than the $\supset$ relation. However, a *better approximation* of $\supset$ relation by the $\leq^{\mathcal{C}}$ operation reduces the number of possible duplicate answers and generally improves the efficiency of query evaluation methods. In the following text we omit the superscripts $^{\mathcal{C}}$. We also use a *strict*[2] version of the $\wedge^{\mathcal{C}}$ operation.

In the rest of the paper we base most of the examples on the following constraint class:

[2] I.e., $\perp$-preserving.

Example 2.2 (Integer Gap-Order Constraints) Let $(\mathbf{Z}, =, <)$ be linearly ordered set of integers numbers and $A \subset \mathbf{Z}$ a finite set. Then the carrier of the class of *Integer Gap-Order Constraints* is generated from the set

$$\mathcal{C}_0 = \{p < x, x = y, x < y, x < r : p, r \in A\}.$$

The appropriate constraint operations on this class are defined in [22] and are based on a graph representation of the gap-order constraints. There are several other classes of constraints used in constraint databases, e.g, the dense order constraints, the equality constraints, or the set-order constraints. For detailed exposition see [23].

The constraints are used as the *raw data* stored in the constraint database; we use finite sets of Horn clauses—Datalog programs—to express queries over such data:

Definition 2.3 A *Datalog$^{\mathcal{C}}$ program* P is a finite set of clauses of the form $A \longleftarrow D, B_1, \ldots, B_k$, where A and B_i are literals (predicate symbols with distinct variables in their arguments) and D a constraint in $\mathcal{C}$.

The individual literals are interpreted by finite sets of constraint atoms—constraint counterparts of sets of ground tuples [31]. In the rest of the paper we slightly abuse the notation and denote the constraint atoms by $A(C)$ where A is a predicate symbol and $C \in \mathcal{C}$. The evaluation of a Datalog$^{\mathcal{C}}$ program is then performed with respect to the class of constraint interpretations—finite sets of constraints associated with literals.

Bottom-up query evaluation [1] is based on iterating a constraint variant of the immediate consequence operator:

Definition 2.4 (Immediate Consequence Operator) Let I be a $\mathcal{C}$-interpretation and P a Datalog$^{\mathcal{C}}$ program. We define

$$\begin{aligned} T_P^{\mathcal{C}}(I) = I \cup \{A(C') : \; & A \leftarrow D, B_1, \ldots, B_k \in P, \\ & B_i(C_i) \in I \text{ for all } 0 < i \leq k, D \in \mathcal{C}, \\ & C = D \wedge C_1 \wedge \ldots \wedge C_k \text{ exists, } C' \in \exists_A.C, \\ & \text{and } C' \not\leq C'' \text{ for any } A(C'') \in I\} \end{aligned}$$

where $\exists_A$ is a shorthand for $\lambda x.\exists^{\mathcal{C}}(FV(A), x)$. The variables in the constraint atoms are renamed using the variable names in the associated atoms of the clause.

Similarly we can define top-down evaluation procedures based on the constraint operations [31]. The following definition states a fundamental property of constraint classes on which the finiteness arguments for bottom-up query evaluation procedures are implicitly based [22, 29, 30, 32]:

Definition 2.5 (Constraint-compact Constraint Class) Let $\mathcal{C}$ be a class of constraints. If for every finite set of variables V and for every set $\mathcal{D} \subseteq \mathcal{C}$ such that $\forall C \in \mathcal{D}.FV(C) \subseteq V$ there is a finite subset $\mathcal{D}_{fin} \subseteq \mathcal{D}$ such that $\forall C \in \mathcal{D}.\exists C' \in \mathcal{D}_{fin}.C \leq^{\mathcal{C}} C'$. (i.e., $\mathcal{D}_{fin}$ covers $\mathcal{D}$ with respect to $\leq^{\mathcal{C}}$) then $\mathcal{C}$ is called *constraint-compact*.

Theorem 2.6 [31] Let $\mathcal{C}$ be a constraint compact constraint class and P an arbitrary Datalog$^{\mathcal{C}}$ program. Then the associated immediate consequence operator $T_P^{\mathcal{C}}$ has a finite closure ordinal.

In the following sections we use the notation

Notation 2.7 Let P be a Datalog$^{\mathcal{C}}$ program and $G(a)$ a ground atom. We say that $G(a)$ follows from P, $P \models G(a)$, if there is a constraint $C \in \mathcal{C}$ such that $G(C) \in (T_P^{\mathcal{C}})^{\omega}(\emptyset)$ and $C(a)$ is true in the constraint theory $\mathcal{C}$.

3 Abstract Interpretation as a Constraint Query

The traditional abstract interpretation often uses constraints to approximate concrete values in the analyzed program. The relation between concrete semantics and its abstract interpretation then looks as follows:

$$\text{a step} \quad \bullet \longrightarrow \bullet \quad \text{is abstracted by} \quad \boxed{\bullet} \longrightarrow \boxed{\bullet}$$

where the concrete values are denoted by bullets ($\bullet$), the constraints by boxes, and the semantic actions by arrows. Note that both the attribute-independent and the relational abstract interpretation techniques [14, 19, 28] follow this pattern; the difference between these two approaches is the arity of the constraints used to approximate the concrete values: unary vs. n-ary. In our case, rather than approximating the individual values by constraints, we approximate the semantic functions:

$$\text{a step} \quad \bullet \longrightarrow \bullet \quad \text{is abstracted by} \quad \boxed{\bullet \longrightarrow \bullet}$$

Note that in our model we can obtain the approximations of the individual values by appropriately applying the quantifier elimination procedure.

Our goal is to define an abstract interpreter for a program P with respect to the given natural semantics to be a Datalog$^{\mathcal{C}}$ program $\mathcal{A}(P)$. For simplicity, we explain the connection using the example of our simple imperative programming language and its natural semantics: the values are vectors of integers and the semantic actions are transformations of these vectors. The technique can be lifted to other settings; we briefly mention some of the other possibilities in Section 5.

Definition 3.1 Let P be an arbitrary program and $S(P)$ the abstract syntactic tree of P. We define $\mathcal{A}(P)$ to be the set of clauses obtained by instantiating the rules of the natural semantics using the elements of $S(P)$ as follows:

1. for every node $op_i \in S(P)$ we create a literal $\mathtt{op}_i(\rho, \rho')$ where ρ and ρ' are tuples of distinct variables of the same arity as the store for P (each). In the following we denote the elements of $S(P)$ by their principal operator op and a distinguished label i.

2. for every literal $\mathtt{op}_i(\rho, \rho')$ and every natural semantics rule

$$\frac{\rho_1 \vdash e_1 \Downarrow \rho_1' \qquad \ldots \qquad \rho_k \vdash e_k \Downarrow \rho_k'}{\rho \vdash op(e_1, \ldots, e_k) \Downarrow f(\rho_1', \ldots, \rho_k')}$$

having op in its conclusion and where $\rho_1 = g_1(\rho), \ldots, \rho_k = g_k(\rho, \rho_1', \ldots, \rho_{k-1}')$ we define a clause

$$\mathtt{op}_i(\rho, \rho') \leftarrow D(\rho, \rho', \rho_1, \rho_1', \ldots, \rho_k, \rho_k'), \mathtt{op}_{i_1, j_1}(\rho_1, \rho_1'), \ldots, \mathtt{op}_{i_k, j_k}(\rho_k, \rho_k')$$

where $op_{i,j}$ is the literal associated with the j-th child of op in $S(P)$ with label i and D the least constraint in $\mathcal{C}$ that approximates the union of graphs of the semantic functions $f, g_1, \ldots, g_k$ in the rule. In general we may need a disjunction of elements of $\mathcal{C}$ here or several clauses.

It is not difficult to show that the resulting program $\mathcal{A}(P)$ has the following properties:

Proposition 3.2 *Let* `main` *be the literal corresponding to the root of the syntactic tree of P. Then*

1. *if $\rho_0 \vdash P \Downarrow \rho_1$ then $\mathcal{A}(P) \models \mathtt{main}(\rho_0, \rho_1)$.*
2. *$\{(\rho_0, \rho_1) | \mathcal{A}(P) \models \mathtt{main}(\rho_0, \rho_1)\}$ is the least relation definable using constraints in $\mathcal{C}$ that contains $\rho_0 \vdash P \Downarrow \rho_1$.*

Proof: (sketch) Clearly, if $\rho \vdash P \Downarrow \rho'$ can be finitely derived using the natural semantic rules, then there is $C \in \mathcal{C}$ and a resolution proof of $\mathcal{A}(P) \models \mathtt{main}(C)$ such that $C(\rho, \rho')$ is true in the theory of $\mathcal{C}$. This can be easily shown by induction on the height of the derivation of $\rho \vdash P \Downarrow \rho'$. Moreover, constraint compactness of $\mathcal{C}$ guarantees termination of Datalog$^{\mathcal{C}}$ queries, in particular the query $\mathcal{A}(P) \models \mathtt{main}(C)$. The result then follows from equivalence between the fixpoint bottom-up evaluation and resolution on constraint compact classes [16, 31]. The minimality follows from Definition 3.1 and the properties of bottom-up query evaluation.

We apply the proposed technique to programs in our simple imperative language as follows: first we determine a finite portion of the store vector referenced in the analyzed program P. This finite store can be easily represented by a ground tuple of integers. Thus the number of attributes of literals in $\mathcal{A}(P)$ program is twice the size of the store vector.

Example 3.3 In this example we show schemes for creating Datalog$^{<Z}$ clauses from the natural semantic rules for our simple imperative language (cf. Figure 2):

Assignment. Let $I := J$ be a command in P. Using the single natural semantic rule for assignment we generate a clause

```
assigni(I1,J1,I2,J2) :- I2 = J1, J2 = J1.
```

provided I and J are the only variables in the storage vector of P; otherwise we define an identity on the rest of the variables. For more complex expressions on the right hand side of the assignment we need to approximate the result of evaluating the expression on the current store by a constraint in $\mathcal{C}$. Datalog clauses that define tha approximations of expressions are generated in a similar fashion to the clauses for commands: they define relations between the input store and a single output value.

Sequential composition. Let $C_1; C_2$ be part of P and I, J the only variables in its storage vector. Then $\mathcal{A}(C_1; C_1)$ is defined by the clause

```
seqi(I0,J0,J2,J2) :- opC1(I0,J0,I1,J1), opC2(I1,J1,I2,J2).
```

where `opC1` and `opC2` are the literals associated with C_1 and C_2. Similar technique generalizes for arbitrary storage vectors.

While-loop. The most complex case is the while-loop. Assume P contains the command `while` $I = 0$ `do` C `od`. Note that there are two natural semantic rules that define the `while` construct. Thus we generate two clauses

```
whilei(I0,J0,I1,J1) :- I0 = 0,  I1 = I0, J1 = J0.
whilei(I0,J0,I2,J2) :- I0 <> 0, opC(I0,J0,I1,J1),
                                whilei(I1,J1,I2,J2).
```

again, assuming I and J are the only variables in the store. Note that we may have to split the second clause into two cases, $0 < I$ and $I < 0$, if we require only conjunctive constraints in the resulting Datalog clauses. Note also that the boolean test in our example of the `while` construct translates immediately to order constraints. Similarly to the case of expressions, for more complex boolean expressions this may not be the case and we may need additional clauses to determine the proper abstract value of the conditional. While the value of the conditional is always true of false, the analysis also needs to determine the constraints describing the preconditions for both the cases.

The clauses for the `if` and `skip` commands are defined similarly.

4 Examples

We present several simple examples that illustrate the use of our technique in discovering abstract properties of imperative programs. We start with a simple sign analysis: the abstract properties we want to study in this case are determined by the set of unary constraints $x < 0$, $x = 0$, and $x > 0$. We can apply the analysis to the program

```
procedure sortn(a :array [1..n] of int) =
    f := 0;
    while f = 0 do
      f := 1;
      for i := 1 to n − 1 do
        if a[i] ≤ a[i + 1] then
          f := 0;
          a[i], a[i + 1] := a[i + 1], a[i]
        fi
      od
    od
```

Note that this program conforms to the grammar in Figure 1: the array $a[1..n]$ is a shorthand for n variables $a[1], \ldots, a[n]$ and similarly the `for` $i := 1$ `to` $n-1$ `do` ... `od` construction a shorthand for $n-1$ instances of its body with appropriately substituted variable names.

Fig. 3. The Bubble-sort procedure *sortn*.

```
procedure p(x, y :int) =
   if x > 0
     then y := x
     else y := 0
   fi
```

The semantic rules specialized with respect to this program are as follows:

```
p(X0,Y0,X1,Y1) :- if(X0,Y0,X1,Y1).

if0(X0,Y0,X1,Y1) :- X0>0,  asg1(X0,Y0,X1,Y1).
if0(X0,Y0,X1,Y1) :- X0<=0, asg2(X0,Y0,X1,Y1).

asg1(X0,Y0,X1,Y1) :- X1 = X0, Y1 = X0.

asg2(X0,Y0,X1,Y1) :- X1 = X0, Y1 = 0.
```

Now by asking the query `?- p(X,_,_,Y)` we get the expected answers for x and y, namely $\{x > 0, y > 0\}$, $\{x = 0, y = 0\}$, and $\{x < 0, y = 0\}$. Note that in this place the traditional Abstract Interpretation would try to find the least upper bound of the results and thus loose information.

Let us consider a less trivial example: the *sortn* program in Figure 3; we have used some syntactic sugar in the syntax of this program; however, it can be easily de-sugared to an equivalent program conforming to the grammar in Figure 1.

```
sortn(A11,...,A1n,A21,...,A2n) :-
        seq1(A11,...,A1n,F1,A21,...,A2n,F2).

seq1(A11,...,A1n,F1,A21,...,A2n,F3) :-
        assign1(F1,F2),
        while1(A11,...,A1n,F2,A21,...,A2n,F3).

assign1(F1,F2) :- F2 = 0.

while1(A11,...,A1n,F1,A21,...,A2n,F2) :-
        F1 <> 0, A11 = A21, ... , A1n = A2n.
while1(A11,...,A1n,F0,A1,...,An,F) :-
        F0 = 0,
        assign2(F0,F1),
        if0(A11,A12,F1,A31,A22,F2),
        if0(A22,A13,F2,A32,A23,F3),
        if0(A23,A14,F3,A33,A24,F4),
        if0(A24,A15,F4,A34,A25,F5),
        ....
        while1(A31,...,A3n,Fn,A1,...,An,F).

assign2(F0,F12) :- F1 = 1.

if0(A1,A2,F1,B1,B2,F2) :-
        A1 <= A2, B1 = A2, B2 = A1, F2 = 0.
if0(A1,A2,F1,B1,B2,F2) :-
        A1 > A2,  B1 = A1, B2 = A2, F2 = F1.
```

Fig. 4. $\mathcal{A}(sortn)$.

Analysis with order constraints. In this case we want to discover order relationships among the elements of the input and output arrays $a[1..n]$, rather than the signs of values stored in the individual variables $a[i]$. Note that the algorithm in Figure 3 does not terminate if the input array contains two equal elements. We have intentionally chosen this version of the algorithm to show that our technique is not confined to analyzing terminating programs only.

The abstract interpreter derived from the *sortn* program is presented in Figure 4 in a slightly simplified but equivalent form: we have removed all the attributes of literals that do not match variables appearing in the syntactic subtree rooted by that particular literal. The resulting Datalog$^{<Z}$ program is equivalent to the original one as the eliminated variables were only passed around without any changes. We can ask the following queries about the abstract properties of the *sortn* program:

– The query about the output values of the *sortn* procedure,

```
?- sortn(_,...,_,A1,...,An),
```

tells us about the relationship between the elements stored in the output array independently of the input. The result is hardly surprising: the individual variables $a[i]$ are related by the constraint $a[1] < a[2] < \ldots < a[n]$[3].

– We can also ask a similar question about the input values: the query

```
?- sortn(A1,...,An,_,...,_)
```

gives us the answer $a[i] \neq a[j]$ for all $0 < i < j \leq n$ (represented as a disjunction of conjunctions of order constraints).

– Moreover, we can ask about relationships between the input and the output values, e.g. `?- sortn(A,...,_,_,...,B)`. The result is $a < b$ or $a = b$, i.e., the last element of the output array is at least as big as the first element of the input array.

The results of the first and second query can be obtained using standard abstract interpretation techniques [9, 10, 11]; the first query corresponds to a *forward* analysis while the second one to a *backward* analysis. However, we do not know about any abstract interpretation analysis proposal that can come up with the result of the third query. Also, our approach combines these three analyses in a single one. Note that the gap-order constraints are very well suited for the analysis of the *sortn* program as it does not contain any complex arithmetic expressions. The method itself can be applied to arbitrary programs. However, the precision of the answers may vary.

We also need to point out that the inherently *flat* nature of queries can not be immediately generalized to analysis of inductively defined values, e.g., lists or trees. Thus our technique can not directly analyze, e.g., a list-sorting procedure. However, in this case we face problems similar to those faced by the standard abstract interpretation technology and we can reuse solutions proposed there, e.g., limiting the depth of terms to a fixed number.

Analysis with set-inclusion constraints. Similarly we can analyze the program for *variable dependencies* using set-inclusion constraints, e.g., what other variable values does a given variable value potentially depend on? By analyzing the *sortn* program using set-inclusion constraints we discover that each of the output variables may depend on each of the input ones. We can also combine these two analyses. Then we can discover that, e.g., if the input array is sorted then each cell of the output array depends only on its value in the input array and thus the *sortn* computes an identity.

[3] The actual answer may also contain constraints implied by the above answer, e.g., $a[1] + 1 < a[3]$ or $a[2] + 3 < a[6]$.

4.1 Shortcomings of the Proposed Method

Precision of Constraint Approximation. In our proposal we approximate the semantic functions in the natural semantic rules by elements of the chosen constraint class. We do not need any external functions operating on the constraint class for this purpose. This way the properties of the constraint class completely determine the properties of the abstract interpretation, e.g., its termination characteristic and precision. In addition, these properties are independent of the actual semantic rules.

The other advantage of our proposal is in the bidirectional character of the analysis. In the *sortn* examples we used exactly the same abstract program $\mathcal{A}(P)$ for both the forward and backward analyses. In contrast, the standard use of *functions* to approximate the semantic actions a priori determines the direction of the analysis.

The disadvantage of this approach is that using external functions to approximate the semantic rules might give us more tight overall approximation as the expressive power of the constraint class itself is often quite limited.

Size of the output. The other problem of our proposal is the size of the output of the analysis: indeed, in the second example in Section 4 the query

```
?- sortn(A1,...,An,_,...,_)
```

produces output exponential in the arity of the query. This is due to splitting the constraint $a[i] \neq a[j]$ to a disjunction of constraints $a[i] < a[j] \vee a[i] > a[j]$, represented during the query evaluation by a two element set. This way the whole result, presented in DNF, is exponential in its arity.

However, if we have used a richer constraint class, e.g., a class that contains the $a[i] \neq a[j]$ constraint as an atomic constraint, we could have avoided this problem and the size of the answer would be quadratic in the arity. However, this approach brings up other problems: we would have to compute unions of the answers produced by the individual clauses of our Datalog program explicitly and to allow the constraint operations to operate on such unions[4].

Slow convergence. The above problem is intimately connected with the approaches that try to speed up the convergence of the fixpoint computation. The main idea is to use a widening operator [5] to approximate disjunctions of constraints by a single more general constraint. This way, besides potentially avoiding the exponential bowup, we can *move up* in the subsumption ordering faster and thus we may reduce the number of iterations needed to reach the fixpoint. Also such a approach may allow us to use constraint classes that are not closed under disjunctions. However, this way we also partially sacrifice the precision of the analysis: up to this point we used the most precise analysis with respect to the chosen constraint class. The only places we approximated were the functions in the semantic rules.

[4] These problems are very similar to problems connected with query evaluation in the N1NF approaches.

5 Related Work

We will not attempt to survey the vast amount of literature on abstract interpretation and constraint deductive databases. The rigorous development of the abstract interpretation area starts with the seminal work [6], many references can be found in [2, 8], or more recently in [25, 26], where the connection of natural semantics and abstract interpretation was studied in detail. For survey of the research in the area of constraint databases we refer to an excellent survey [23]. In the remainder of this section we want to point out two other possibilities of application of constraint technology to abstract interpretation:

Higher-order Languages. In the whole paper we have only considered an imperative (first-order) language. However, the abstract interpretation techniques can also be applied to higher-order programming languages, e.g., PCF [27]. Here the situation is more complicated: besides abstractions for ground types we also need to abstract function types [25]. This goal is often achieved by defining a domain of values based on *closures* which in turn lead to a natural nesting of such closures inside each other. In [25] an abstract interpretation technique was developed to analyze such programs. Similar effect can be achieved using set-order constraints [29].

Small Step Semantics. We have also assumed that the semantics of the given programming language is given by a set of natural semantic rules. However, the abstract interpretation technology can be applied to a small-step operational semantics as well [21, 26]. The abstract interpretation defined in [21] uses the set-order constraints to specify a closure analysis of untyped λ-calculus.

6 Conclusion and Future Work

In this paper we made a first step towards combining results in the areas of constraint databases and abstract interpretation based program analysis. We have introduced a new constraint approach to abstract interpretation.

However, the paper points out only the most obvious similarities of the two research areas. Underneath, there are many open questions that need to be studied in depth. Some of these questions were stated in Section 5; these were mostly connected with the applications of constraint database technology to other areas of program analysis, e.g., to higher order programming languages or to different semantics. However, there are still more fundamental questions to be answered, e.g.:

1. Our Abstract Interpretation method uses a considerably more complex domain of abstract values: besided the usual joins and meets of abstract values the structure also contains the *cylindrification operations* (projections) [13] used to eliminate variables from constraints. How can we benefit from this additional structure of the abstract values? How do these additional operations affect the abstract interpretation itself?

2. The usual abstract interpretation does not collect the results in finite sets (relations), but rather approximates such sets by their least upper bound. Clearly, we could modify our query evaluation mechanism to use this strategy: we can equip the constraint class with an additional operation that computes the needed least upper bounds. How does such a modification affect:

1. the termination characteristic of the constraint deductive queries?
2. the complexity of query evaluation?
3. the precision of answers?

The third question is especially intriguing from the point of view of the constraint database research: it may lead to a natural theory of approximate query evaluation.

3. There are many domains of abstract values proposed in the abstract interpretation literature. Can these be meaningfully used as constraint classes in general applications of constraint databases?

4. In the deductive databases community a considerable effort has been directed towards including negation in the otherwise positive Horn clauses. Can negation (stratified, well-founded, stable, etc.) be used to enhance the capabilities of abstract interpretation, e.g., in the case of coinductive types and greatest fixpoints [6, 25]?

References

1. Abiteboul, S., Hull, R., Vianu, V. Foundations of Databases. Addison–Wesley, 1995.
2. Abramski, S., Hankin, C. (eds.) Abstract Interpretation of Declarative Languages. Ellis Horwood, 1987.
3. Codish, M., Demoen, B. Analysing Logic Programs Using 'Prop'-ositional Logic Programs and a Magic Wand. Proc. *1994 International Logic Programming Symposium*, MIT Press, 114–129, 1994.
4. Codognet P., File, G. Computations, Abstractions, and Constraints in Logic Programming. Proc. *IEEE Conference on Computer Languages*, 155–164, 1992.
5. Cousot, P., Cousot, R. Static Determination of Dynamic Properties of Programs. in Proc. *2nd Int. Symposium on Programming*, 1976.
6. Cousot, P., Cousot, R. Abstract Interpretation: a unified lattice model for static analysis of programs. In *Proc. 4th ACM Symp. on Principles of Programming Languages*, 238–252, 1987.
7. Cousot, P., Cousot, R. Systematic design of program analysis frameworks. In *Proc. 6th ACM Symp. on Principles of Programming Languages*, 269–282, 1989.
8. Cousot, P., Cousot, R. Abstract Interpretation and Application to Logic Programs. *J. Logic Programming*, 13:103–179, 1992.
9. Cousot, P., Halbwachs, N. Automatic Discovery of Linear Restraints Among Variables of a Program. Proc. *ACM Symp. on Principles of Programming Languages*, 84–96, 1978.

10. Granger, P. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*,165–199, 1989.
11. Static analysis of linear congruences among variables of a program. Tech. Rep. LIX, Paris, France, 1990.
12. Gunter C.A. Semantics of Programming Languages. MIT Press, 1992.
13. Henkin, L., Monk, J.D., Tarski, A. Cylindric Algebras. Part I, II. Studies in Logic and Foundations of Mathematics, vols. 64, 115, North Holland 1971, 1985.
14. Jones, N.D., Muchnick, S.S. Flow analysis and optimization of LISP-like structures. In Proc. *6th ACM Symp. on Principles of Programming Languages*, 244–256, 1979.
15. Kanellakis, P. C., Kuper, G. M., Revesz, P.Z . Constraint Query Languages. Journal of Computer and System Sciences 51(1):26-52, 1995.
16. Lloyd, J. Foundations of Logic Programming. Springer-Verlag, 1987.
17. Marriott, K., Sondergaard, H. Analysis of Constraint Logic Programs. Proc. NACLP 1990, 531-547.
18. Marriot, K., Stuckey, P. J. Approximating interactions between linear arithmetic constraints. Proc. *1994 International Logic Programming Symposium*, MIT Press, 1994, 571–585.
19. Mycroft, A., Jones, N.D. A relational framework for abstract interpretation. In *Programs as Data Objects*, LNCS 217, 156–171, 1985.
20. Naughton, J.F., Ramakrishnan, R., Sagiv, Y., Ullman, J.D. Argument Reduction by Factoring. in Proc. *Intl. Conf. on Very Large Data Bases*, 1989.
21. Palsberg, j. Closure Analysis in Constraint Form. In Proc. *CAAP'94*, 276–290, 1994. (to appear in ACM Transactions on Programming Languages and Sytems).
22. Revesz, P. Z. A Closed Form Evaluation for Datalog Queries with Integer (Gap)-Order Constraints. *Theoretical Computer Science*, vol. 116, no. 1, 117–149, 1993.
23. Revesz, P. Constraint Query Languages. A Survey. (unpublished manuscript), 1995.
24. Schmidt, D.A. Denotational Semantics: A methodology for Language Development. Allyn and Bacon, 1986.
25. Schmidt, D.A. Natural-semantics-based abstract interpretation. In Proc. *Static Analysis Symposium*, LNCS 983, 1–18, 1995.
26. Schmidt, D.A. Abstract Interpretation of Small-Step Semantics. Presented at 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages, Stockholm, June 1996.
27. Scott, D.S. An Alternative to CUCH, ISWIM, OHWY. unpublished manuscript, 1969.
28. Sondergaard, H. An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction. Proc. ESOP 1986, 327-338.
29. Srivastava, D., Ramakrishnan, R., Revesz, P. Z. Constraint Objects. Proc. *Intl. Workshop on Principles and Practice of Constraint Programming*, 218–228, 1994.
30. Toman, D., Chomicki, J., Rogers D. S. Datalog with Integer Periodicity Constraints. Proc. *1994 International Logic Programming Symposium*, MIT Press, 1994, 189–203.
31. Toman, D. Top-Down beats Bottom-Up for Constraint Based Extensions of Datalog. Proc. *1995 International Logic Programming Symposium*, MIT Press, 1995, 98-112.
32. Ullman J. D. Principles of Database and Knowledge-base Systems, Vol. 1,2. Computer Science Systems, 1989.

This article was processed using the LaTeX macro package with LLNCS style