Modular Class Analysis with DATALOG*

Frédéric Besson and Thomas Jensen

IRISA/INRIA/CNRS
Campus de Beaulieu
F-35042 Rennes, France

Abstract DATALOG can be used to specify a variety of class analyses for object-oriented programs as variations of a common framework. In this framework, the result of analyzing a class is a set of DATALOG clauses whose least fixpoint is the information analysed for. Modular class analysis of program fragments is then expressed as the resolution of *open* DATALOG programs. We provide a theory for the partial resolution of sets of open clauses and define a number of operators for reducing such open clauses.

1 Introduction

One of the most important analyses for object-oriented languages is *class analysis* that computes an (over-)approximation of the set of classes that an expression can evaluate to at run-time [1,3,12,23,24]. Class analysis forms the foundation for static *type checking* for OO programs aimed at guaranteeing that methods are only invoked on objects that implement such a method. It is also used for building a precise call graph for a program which in turn enables other optimisations and verifications. For example, the information deduced by class analysis can in certain cases be used to replace virtual method invocations by direct calls to the code implementing the method.

Existing class analyses are all whole-program analyses that require the entire program to be present at analysis time. There are several reasons for why it is desirable to improve this situation. The size of the object-oriented code bases to be analysed means that a whole-program analysis is lengthy. Having to re-analyse all the program every time a modification is made means that the analysis is of little use during a development phase. Furthermore, dealing with languages that allow *dynamic class loading* means that not all code is available at analysis time. These shortcomings must be resolved by developing more incremental and modular analyses, that can deal with fragments of programs.

Modular program analysis has been the object of several recent studies. Cousot and Cousot [10] examine the various approaches to modular program analysis and recast these in a uniform abstract interpretation framework. The essence of their analysis is a characterisation of modular program analysis as the problem of calculating approximations to a higher-order fixpoint. In this paper

* This work was partially funded by the IST FET/Open project "Secsafe".

we demonstrate how this fixpoint characterisation can be instantiated to the case of class analyses expressed using DATALOG clauses. In this case, the result of the class analysis is defined as the least solution of the set of clauses generated from the program in a syntax-directed manner. A modular analysis is a procedure that transforms a partial set of constraint into an equivalent "more resolved" set, where "more resolved" means that the number of iterations required to reach the least solution has been reduced.

The class analysis will be expressed as a translation from a simple object-oriented programming language into constraints specified using the DATALOG language. DATALOG is a simple relational query language yet rich enough to give a uniform description of a number of control flow analyses of object-oriented languages, including the set-based analysis of Palsberg and Schwartzbach. It has an efficient bottom-up evaluator that provides an implementation of the analysis of closed programs for free. The analysis of program fragments gives rise to *open* sets of DATALOG clauses, for which a number of powerful *normalisation* operators exist. Finally, the semantic theory of open logic programs provides a basis for defining abstract union operators on constraint systems corresponding to the union of two program fragments.

The contributions of the paper can be summarised as follows.

- We show how DATALOG can be used for specifying class analyses in a uniform manner. We notably show how a number of existing analyses can be expressed naturally in this framework.
- We extend this to a theory of modular class analysis in which the analysis of program fragments are modeled using *open* DATALOG programs.
- We define a number of partial resolution techniques for reducing an open DATALOG program to a solved form.

Section 2 defines a simple object-oriented programming language. Section 3 recalls basic notions of DATALOG, which is then used to define a number of class analyses in Section 4. Open DATALOG programs arising from analysis of program fragments are introduced in Section 5.1. In Section 5.2, we give a characterisation of correct modular resolution methods. We then show, in Section 6, how several modular resolution techniques fit into this framework.

2 An Object-Oriented Language

Our analysis is defined with respect to an untyped imperative class-based object-oriented language. To focus on the class analysis principles, language constructs are kept to a minimum. The precise syntax is defined in Figure 1.

A program P is made of a set of class declarations. Each class C is identified by a unique name c and define a set of methods. It may extend an existing class c'. Within a class, each method M is uniquely identified by its signature m/i where m is the method name and i the number of arguments. The method body is a sequence of instructions. The instruction x:= `new` c creates an object of class c. Instruction $x.fd$:=y assigns the value of variable y to the field fd of

$$
\begin{array}{lll}
P & ::= & \{C_1, \ldots, C_n\} \\
C & ::= & \texttt{class}\ c\{M_1, \ldots, M_n\} \mid \\
 & & \texttt{class}\ c\ \texttt{extends}\ c'\{M_1, \ldots, M_n\} \\
M & ::= & m(x_1, \ldots, x_n) \quad IL \\
IL & ::= & [I_1, \ldots, I_n] \\
I & ::= & x{:=}\ \texttt{new}\ c \mid x.fd{:=}y \mid x{:=}y.fd \mid \\
 & & x{:=}x_0.f(x_1, \ldots, x_n) \mid \texttt{ret}\ x
\end{array}
$$

Figure1. A minimalist object-oriented language

the object referenced by x. Similarly, $x{:=}y.fd$ transfers the content of field fd of y to variable x. The instruction $x{:=}x_0.f(x_1, \ldots, x_n)$ invokes the method f on the object stored in x_0 with $x_1, \ldots, x_n$ as arguments and stores the result in x. Finally, `ret` x ends the execution of an invoked method by returning the value of x. Except the last instruction that models method return, all the other instructions are different kinds of assignments: object creation, field update, field access and dynamic method invocation. Following the object-oriented conventions, in a method body, the current active object is referenced as *self*.

Execution of a program starts at the first instruction of the method `main` of class `main`. The inheritance relation between the classes of a program is acyclic. Because our language does not provide multiple inheritance, the class hierarchy is a forest (of trees). Virtual method resolution is defined by a method lookup algorithm that given a class name c and a method signature m returns the class c' that implements m for the class c. In order to take inheritance into account, the method lookup algorithm walks up the class hierarchy from class c and eventually returns the first class c' that defines a matching method m.

3 DATALOG

We recall some basic facts about DATALOG [29] that will serve as language for specifying the class analyses. Syntactically, DATALOG can be defined as PROLOG with only nullary functions symbols *i.e.,* constants. Hence, most of the definitions and properties of DATALOG programs are inherited from PROLOG. It can also be presented as a relational query language extended with recursion.

The denotation of a DATALOG program is the least set of atoms that satisfy the clauses of the program. The fundamental difference with respect to Prolog is that the least Herbrand model is computable.

Definition 1. *Let Π be a (finite) set of predicate symbols and V (resp. C) a set of variables (resp. constant symbols).*

- *An atom is a term $p(x_1, \ldots, x_n)$ where $p \in \Pi$ is a predicate symbol of arity n and each x_i ($i \in [1, \ldots, n]$) is either a variable or a constant ($x_i \in V \uplus C$).*

- *A clause is a formula $H \leftarrow B$ where H (the head) is an atom while the body B is a finite set of atoms.*
- *A program P is a set of clauses.*

For atom A, $Var(A)$ is the set of variables occurring in A and $Pred(A)$ is the predicate symbol of A. Var and $Pred$ are extended the obvious way to clauses and programs. An atom A is said *ground* if its set of variables is empty. A substitution $\sigma : V \rightarrow V \uplus C$ is a mapping from variables to variables or constants. A *ground* substitution maps variables to constants. We note $A\sigma$ the application of a substitution σ to an atom A.

A Herbrand interpretation I is a set of ground atoms. Given a set of predicate symbols Π and a set of constant C, the Herbrand base $HB(\Pi, C)$ is the (greatest) set of ground atoms that can be built from the predicate symbols in Π and constants in C. The least Herbrand model of a program P is a Herbrand interpretation defined as the least fixed point of a monotonic, continuous operator T_P known as the *immediate consequence operator* [2].

Definition 2. *For a program P and a Herbrand interpretation I, the operator T_P is defined by:*

$$T_P(I) = \{A \mid \exists (H \leftarrow B \in P, \sigma : V \rightarrow C). \forall (b \in B). b\sigma \in I, A = H\sigma\}$$

In the following, the fixed point operator is noted *lfp*. Hence, the least Herbrand model of a program P is $lfp(T_P)$.

4 Class Analysis in DATALOG

In this section we describe a class analysis in the form of a syntax-directed translation that maps a term L of the object-oriented language defined in Section 2 to a set of DATALOG clauses. The result of the analysis is the least fixpoint of these clauses. The analysis presented in this section deals with complete programs; the following sections will be concerned with showing how to combine separate analyses of individual classes to eventually recover the result of the analysis of the complete program.

One of the advantages of DATALOG is that it allows to specify a collection of class analyses in a common framework, thus allowing to relate a number of known analyses. To make this point clear, we first give an intuitive description of the basic analysis and then explain how it can be varied to obtain analyses of different degree of precision. Intuitively, the basic analysis works as follows. For each variable x defined in method m of class c in the program we introduce a unary predicate named $c.m.x$ that characterises the set of objects being stored in that variable. Each assignment to a variable will result in a clause defining the corresponding predicate. The heap of allocated objects is represented by a collection of binary predicates $fd(\_, \_)$, one for each field name fd used in the program. If an object o_1 of class c_1 references an object o_2 of class c_2 via field fd then $fd(c_1, c_2)$ holds. To deal with method calls, each method signature m

gives rise to a pair of predicates $m.call$ and $m.ret$ such that $m.call$ collects the arguments of all calls to methods named m while $m.ret$ collects the return values from the different calls to m.

A number of syntactic properties of a program are represented by the predicates *class*, *subclass*, *sig* and *define*. Given an object-oriented program P, we have: $class(c)$ if c is a class of P; $subclass(c, c')$ if c is a direct subclass of c' in P; $sig(m)$ if m is a method signature of P and $define(c, m)$ if class c of P defines a method of signature m. The dynamic method lookup is encoded by the predicate lk such that $lk(o, f, c)$ if a call to method f on an object of class o is resolved to the definition of f found in class c.

$$
\begin{array}{ll}
lk(c, m, c) & \leftarrow \{define(c, m)\} \\
lk(c, m, c') & \leftarrow \{notDefine(c, m), subclass(c, c''), lk(c'', m, c')\} \\
notDefine(c, m) & \leftarrow \{class(c), sig(m), \neg define(c, m)\}
\end{array}
$$

A technical note: because of the use of negation ($\neg$), the clause defining the predicate *notDefine* does not strictly comply with our definition of clauses. However, as long as negations are stratified – recursion cycles are negation free – the least Herbrand model exists.

4.1 Context-Sensitive Analyses

There are a number of places in which the precision of the basic class analysis can be fine-tuned:

- Modeling of the heap of objects. In the basic analysis, objects in the heap are simply abstracted by their class identifier, hence the abstract domain of objects is defined by $Object = Class$. Other ways of abstracting objects take into account the *creation context*. For instance, objects can be distinguished by their program point of creation, in which case we get $Objet = Class \times PP$ where $PP = Class \times Meth \times PC$ is the set of program points.
- Distinguishing different calls to the same method. The precision of an analysis can be improved by keeping track of the program point at which a method was invoked $Context = PP$. Other ways of separating the method calls is by distinguishing them according to the class of their arguments (see the Cartesian Product abstraction below).
- Distinguishing different occurrences of the same variable. The basic analysis keeps one set for each variable. All assignments to that variable will contribute to this set. Introducing one set for each occurrence of a variable can lead to more precise analysis results, see *e.g.,* [18] for the benefits obtained in the case of binding-time analysis for imperative languages.

The last type of context sensitivity is relatively straightforward to obtain by syntactic transformations so here we focus on the first two items: creation- and call-context sensitive analyses.

Context sensitivity is expressed separately by means of the predicates *object*, $objCtx$, $methCtx/n$ and *classOf*. The predicate *object* is used to define the abstract domain of objects. The predicate $objCtx$ models a function that given a

Program

$$[\![\{C_1, \ldots C_n\}]\!] = \bigcup_{i \in [1,\ldots,n]} [\![C_i]\!] \cup \left\{ \begin{array}{l} main.call(o, ctx) \leftarrow \\ \quad \left\{ \begin{array}{l} objCtx(\underline{0}, \underline{0}, \underline{0}, \underline{main}, \underline{0}, \underline{0}, o), \\ methCtx/n(\underline{0}, \underline{0}, \underline{0}, o, \underline{0}, ctx) \end{array} \right\} \end{array} \right\}$$

Class

$$\begin{array}{ll} [\![\texttt{class } c \quad Meths]\!] & = [\![Meths]\!]_c \cup \{class(\underline{c}) \leftarrow \{\}\} \\ [\![\texttt{class } c \texttt{ extends } c' \quad Meths]\!] & = [\![Meths]\!]_c \cup \left\{ \begin{array}{ll} subclass(\underline{c}, \underline{c'}) & \leftarrow \{\} \\ class(\underline{c}) & \leftarrow \{\} \end{array} \right\} \end{array}$$

Methods

$$[\![\{M_1, \ldots, M_n\}]\!]_c = \bigcup_{i \in [1,\ldots,n]} [\![M_i]\!]_c$$

$$[\![m(x_1, \ldots, x_n) IL]\!]_c = [\![m(x_1, \ldots, x_n)]\!]_c \cup [\![IL]\!]_{c,m}$$

$$[\![m(x_1, \ldots, x_n)]\!]_c = \left\{ \begin{array}{ll} define(\underline{c}, \underline{m/n}) & \leftarrow \{\} \\ sig(\underline{m/n}) & \leftarrow \{\} \\ c.m.ctx(ctx) & \leftarrow \left\{ \begin{array}{l} m.call(o, o_1, \ldots, o_n, ctx), \\ classOf(o, c'), lk(c', \underline{m/n}, \underline{c}) \end{array} \right\} \\ c.m.self(o, ctx) & \leftarrow \{m.call(o, o_1, \ldots, o_n, ctx), c.m.ctx(ctx)\} \\ c.m.x_1(o_1, ctx) & \leftarrow \{m.call(o, o_1, \ldots, o_n, ctx), c.m.ctx(ctx)\} \\ \ldots & \\ c.m.x_n(o_n) & \leftarrow \{m.call(o, o_1, \ldots, o_n, ctx), c.m.ctx(ctx)\} \end{array} \right\}$$

Instructions

$$[\![[I_1, \ldots, I_n]]\!]_{c,m} = \bigcup_{i \in [1,\ldots,n]} [\![I_i]\!]_{c,m,i}$$

$$\begin{array}{ll} [\![x \texttt{ := new } c']\!]_{c,m,i} & = \left\{ c.m.x(o, ctx) \leftarrow \left\{ \begin{array}{l} c.m.self(o', ctx), \\ objCtx(\underline{c}, \underline{m}, \underline{i}, \underline{c'}, ctx, o', o) \end{array} \right\} \right\} \\ [\![x.fd \texttt{ := } y]\!]_{c,m} & = \{fd(o, o') \leftarrow c.m.x(o, ctx), c.m.y(o', ctx)\} \\ [\![x \texttt{ := } x_0.f(x_1, \ldots, x_n)]\!]_{c,m,i} & = \left\{ \begin{array}{l} c.m.i.call(o_0, \ldots, o_n, ctx, ctx') \leftarrow \\ \quad \left\{ \begin{array}{l} x_0(o_0, ctx), \ldots, x_n(o_n, ctx), \\ methCtx/n(\underline{c}, \underline{m}, \underline{i}, o_o, \ldots, o_n, ctx, ctx') \end{array} \right\} \\ f.call(o_0, \ldots, o_n, ctx') \leftarrow \\ \quad \{c.m.i.call(o_0, \ldots, o_n, ctx, ctx')\} \\ c.m.x(o, ctx) \leftarrow \\ \quad \{f.ret(o, ctx'), c.m.i.call(o_0, \ldots, o_n, ctx, ctx')\} \end{array} \right\} \\ [\![ret \quad x]\!]_{c,m,i} & = \{m.ret(o, ctx) \leftarrow c.m.x(o, ctx)\} \end{array}$$

Figure2. Generation algorithm for class analysis

syntactic program point (class, method, program counter), a class to instantiate and an analysis context yields a new object. If $objCtx(c, m, i, c', ctx, self, newObj)$ holds then $newObj$ is a novel object of class c' built from the program point (c, m, i), for the call context ctx and the current object $self$.

The predicate family $methCtx/n$ models a function that given a syntactic program point (class, method, program point), the n arguments of the call and the current call context yields a novel call context. If

$$methCtx/n(c, m, i, self, o_1, \ldots, o_n, ctx, newCtx)$$

holds then $newCtx$ is a novel call context built from the program point (c, m, i), for the call context ctx and the argument objects of the call $self, o_1, \ldots, o_n$. Finally, the predicate $classOf(o, c)$ permits to obtain the class c of the object o. The predicates $objCtx$ and $classOf(o, c)$ must satisfy the following coherence constraint: $objCtx(c, m, i, c', ctx, o) \Rightarrow classOf(o, c')$.

4.2 Example Analyses

In the following we specify a number of known analyses as variations of the analysis in Figure 2. We do this by specifying the abstract domains of objects and contexts, and by defining the instantiation of the predicates $objCtx$, $methCtx$ and $classOf$. For certain analyses, we make use of a tuple notation which is not part of DATALOG. However, this extension is not a theoretical problem: such finite depth terms can be flattened and encoded by adding extra arguments to predicate symbols. To give a more uniform presentation, we keep a tuple notation.

0-CFA 0-CFA is a degenerated context sensitive analysis in which objects are abstracted by their class and where there exists a single call context identified by the constant $\underline{ctx}$. Hence, we have $Object = Class$ and $Context = \{\underline{ctx}\}$.

$$\begin{array}{ll} objCtx(c, m, i, c', ctx, c') & \leftarrow \{\} \\ methCtx/n(c, m, i, o_0, \ldots, o_n, ctx, \underline{ctx}) & \leftarrow \{\} \\ classOf(c, c) & \leftarrow \{\} \end{array}$$

1/2-CFA Some analyses of object-oriented programs deal with inheritance by copying the inherited methods into the inheriting class [23]. This syntactic unfolding adds a certain degree of call context sensitivity to an analysis because it distinguishes between a method according to which class of object it is called on. To model this effect of unfolding the inheritance mechanism, we keep as call context the class of the receiver of the current call. This is expressed by the repeated occurrence of $self$ in the definition of $methCtx$. We have $Object = Class$ and $Context = Object$.

$$\begin{array}{ll} objCtx(c, m, i, c', ctx, c') & \leftarrow \{\} \\ methCtx/n(c, m, i, self, o_1, \ldots, o_n, ctx, self) & \leftarrow \{\} \\ classOf(c, c) & \leftarrow \{\} \end{array}$$

k-l-CFA The principle of the k-l-CFA hierarchy of analysis is to keep a call string of length k and a creation string of length l. As a result, the call context is a tuple of the k call instructions that lead to the call. Similarly, an object o_1 now contains information about the object o_2 that created it, and the object o_3 that created o_2, and ... the object o_l that created the object o_{l-1}. We have $Object = Class \times PP^l$ and $Context = PP^k$.

$$\begin{array}{ll} objCtx(c,m,i,c',ctx,o,o') & \leftarrow \left\{\begin{array}{l} o = (c'', (p_1, \ldots, p_l)), \\ o' = (c', ((c,m,i), p_1, \ldots, p_{l-1})) \end{array}\right\} \\ methCtx/n(c,m,i,o_0,\ldots,o_n,ctx,ctx') & \leftarrow \left\{\begin{array}{l} ctx = (p_1, \ldots, p_k), \\ ctx' = ((c,m,i), p_1, \ldots, p_{k-1}), \end{array}\right\} \\ classOf((c,l),c) & \leftarrow \{object((c,l))\} \end{array}$$

Cartesian Product Algorithm This kind of context sensitivity for class analysis was first discussed by Agesen [1]. A call context is built from the arguments of the call. Calls to the same method are distinguished as soon as the arguments are different. The set of call contexts of a method with n arguments is then $Context_n = Object^n$. Thus, the precision of the overall analysis depends on the object abstraction. Here, we show an instantiation where the object creation context is the program point of creation ($Object = Class \times PP$).

$$\begin{array}{ll} objCtx(c,m,i,c',ctx,o,o') & \leftarrow \left\{\ o' = (c', (c,m,i)),\ \right\} \\ methCtx/n(c,m,i,o_0,\ldots,o_n,ctx,ctx') & \leftarrow \left\{\ ctx' = (o_0, \ldots, o_n),\ \right\} \\ classOf((c,l),c) & \leftarrow \{object(c,l)\} \end{array}$$

Example 1. Consider the following contrived program

$$P = \{\ \texttt{class}\ main\{main()[self\ \texttt{:=}\ self.fd;\ \texttt{ret}\ self]\}$$

For the 0-CFA analysis, here are the generated constraints.

$$\begin{array}{ll} main/0.call(\underline{main}) & \leftarrow \{\} \\ define(\underline{main}, \overline{main/0}) & \leftarrow \{\} \\ main.main/0.self(o) & \leftarrow \{main/0.call(o), lk(o, \overline{main/0}, \underline{main})\} \\ main.main/0.self(o) & \leftarrow \{main.main/0.self(o'), \overline{fd(o', o)}\} \\ main/0.ret(o) & \leftarrow \{main.main/0.self(o)\} \end{array}$$

Next section, we detail how the clauses defining *self* can be reduced using a combination of modular resolution techniques.

5 Modular Resolution

The results in this section form the theoretical basis for analysing a class hierarchy in a compositional fashion. In this approach, each class is first analysed separately and the resulting DATALOG programs reduced towards a solved form. Then, the reduced programs are joined together and further reductions can take place.

For a class, the generation algorithm yields a set of DATALOG clauses. However, because a class is not a stand-alone program, code in one class may invoke methods defined in another class. This means that some predicate symbols appearing in the clauses modeling a single class may be either partially or totally undefined. For those predicates, other classes may enrich their definition. To make this explicit, we introduce the term *open* predicates. Being *open* is a property that depends on the scoping rules of the analysed language. For our class analysis, *open* predicate symbols arise from the analysis of method calls, method declaration, method returns and field updates. For instance, the return instruction of a method of signature m defined by a class c contributes to the definition of the $m.ret$ predicate. Because any class implementing a method m also contributes to the definition of the predicate symbol $m.ret$, its definition is kept *open* until all the program is analysed.

5.1 Open DATALOG Programs

Bossi *et al.* [4] define a compositional semantics for open logic programs. We use their definition of open programs.

Definition 3 (Bossi *et al.* [4]). *An open* DATALOG *program P^{Ω} is a (*DATALOG*) program P together with a subset Ω of its predicate symbols ($\Omega \subseteq Pred(P)$). A predicate symbol in Ω is considered to be only partially defined in P.*

The immediate consequence operator T is extended to open programs by ignoring the set of open predicates: $T_{P^{\Omega}} = T_P$.

Open clauses generated from individual classes are joined to model the analysis of the whole program. Such union of open programs requires common predicate symbols to be open. Otherwise, union is undefined. While analysing a class in isolation, it is then mandatory to declare open any predicate symbol that may be referenced elsewhere.

Definition 4. *Let $P_1^{\Omega_1}$ and $P_2^{\Omega_2}$ be open programs. Under the condition that $Pred(P_1) \cap Pred(P_2) \subseteq \Omega_1 \cap \Omega_2$, $P_1^{\Omega_1} \cup P_2^{\Omega_2}$ is defined by*

$$P_1^{\Omega_1} \cup P_2^{\Omega_2} = (P_1 \cup P_2)^{\Omega_1 \cup \Omega_2}$$

Property 1. Union of open programs is associative.

This property is essential for our purpose: the order in which analyses of classes are joined does not matter.

At this point, we are (only) able to map classes to open DATALOG clauses and join them incrementally to get the clauses modeling the whole program being analysed. Since these operations are strictly syntactic, no resolution occurs. Next sections will characterise and provide modular resolution methods.

5.2 Approximation of Open Programs

A modular resolution method maps open programs to open programs while preserving the overall correctness of the whole analysis. We formalize the notion of approximation by means of a pre-order relation $\sqsubseteq$ over open programs. This generalizes the usual containment relation over DATALOG programs [28].

Definition 5. *Let $P_1^{\Omega_1}$ and $P_2^{\Omega_2}$ be* DATALOG *open programs. $P_2^{\Omega_2}$ is an over-approximation of $P_1^{\Omega_1}$ ($P_1^{\Omega_1} \sqsubseteq P_2^{\Omega_2}$) if and only if:*

- *$\Omega_1 = \Omega_2$ and $Pred(P_1) = Pred(P_2)$*
- *for all Q^Ω such that $P_1^{\Omega_1} \cup Q^\Omega$ and $P_2^{\Omega_1} \cup Q^\Omega$ are defined, we have*

$$lfp(T_{P_1 \cup Q}) \subseteq lfp(T_{P_2 \cup Q})$$

Property 2. The relation $\sqsubseteq$ is reflexive and transitive.

The relation $\sqsubseteq$ gives rise to an equivalence relation between open programs.

Definition 6. *Let P_1^Ω, P_2^Ω be open programs. P_1^Ω is equivalent to P_2^Ω ($P_1^\Omega \equiv P_2^\Omega$) if and only if $P_1^\Omega \sqsubseteq P_2^\Omega$ and $P_2^\Omega \sqsubseteq P_1^\Omega$.*

The relevance of $\sqsubseteq$ for modular resolution lies in the following fundamental lemma. It shows that open programs remain in relation by $\sqsubseteq$ when an arbitrary open program is adjoined to them. This is the key property of modular resolution methods: whatever the unknown clauses that could be added later, the transformation preserves the correctness of the analysis.

Lemma 1. *Let $P_1^{\Omega_1}$, $P_2^{\Omega_2}$ and Q^Ω be open programs. If $P_1^{\Omega_1} \sqsubseteq P_2^{\Omega_2}$ and $P_1^{\Omega_1} \cup Q^\Omega$ is defined, then we have*

$$P_1^{\Omega_1} \cup Q^\Omega \sqsubseteq P_2^{\Omega_2} \cup Q^\Omega$$

Proof. To begin with, we observe that $P_2^{\Omega_2} \cup Q^\Omega$ is defined. This follows trivially from the definition of $\sqsubseteq$ ($P_1^{\Omega_1} \sqsubseteq P_2^{\Omega_2}$).

Now, consider an arbitrary open program $Q'^{\Omega'}$. To prove the lemma, we show that if $(P_1^{\Omega_1} \cup Q^\Omega) \cup Q'^{\Omega'}$ and $(P_1^{\Omega_1} \cup Q^\Omega) \cup Q'^{\Omega'}$ are defined then the fixpoints are ordered by set inclusion: $lfp(T_{(P_1 \cup Q) \cup Q'}) \subseteq lfp(T_{(P_1 \cup Q) \cup Q'})$. Because union of open programs is associative (Prop 1), we have $(P_1^{\Omega_1} \cup Q^\Omega) \cup Q'^{\Omega'} = P_1^{\Omega_1} \cup (Q \cup Q')^{\Omega \cup \Omega'}$ and $(P_2^{\Omega_2} \cup Q^\Omega) \cup Q'^{\Omega'} = P_2^{\Omega_2} \cup (Q \cup Q')^{\Omega \cup \Omega'}$. Exploiting that $P_1^{\Omega_1} \sqsubseteq P_2^{\Omega_2}$, we conclude that $lfp(T_{P_1 \cup (Q \cup Q')}) \subseteq lfp(T_{P_1 \cup (Q \cup Q')})$. As a result, by associativity of union of sets, the lemma holds.

Based on the relation $\sqsubseteq$ on open programs, we define the notion of correct and exact resolution methods.

Definition 7. *A correct (resp. exact) resolution method $\mathcal{R}^\subseteq$ (resp. $\mathcal{R}^=$) is such that for all open program P^Ω, we have*

$$P^\Omega \sqsubseteq \mathcal{R}^\subseteq(P^\Omega) \qquad P^\Omega \equiv \mathcal{R}^=(P^\Omega)$$

It should be noted that equivalence of DATALOG programs is undecidable [28]. This is of little concern to us because we are only interested in *transformations* that are guaranteed either to preserve equivalence or to yield a safe approximation.

Finally, theorem 1 formally states that correct resolution methods as defined above are indeed correct for the global analysis.

Theorem 1. *Let $P = \bigcup_{i\in[1,\dots,n]} P_i^{\Omega_i}$ a union of open programs. For any collection of correct resolution methods $\mathcal{R}_1^{\subseteq}$, ... ,$\mathcal{R}_n^{\subseteq}$ the following holds :*

$$lfp(T_{\bigcup_i P_i^{\Omega_i}}) \subseteq lfp(T_{\bigcup_i \mathcal{R}_i^{\subseteq}(P_i^{\Omega_i})})$$

Proof. Because $\mathcal{R}_i^{\subseteq}$ are correct resolution methods (Def 7), we have for any $i \in [1,\dots,n]$ that $P_i^{\Omega_i} \sqsubseteq \mathcal{R}_i^{\subseteq}(P_i^{\Omega_i})$. Because unions are defined, by recursively applying Lemma 1, we obtain that $\bigcup_i P_i^{\Omega_i} \sqsubseteq \bigcup_i \mathcal{R}_i^{\subseteq}(P_i^{\Omega_i})$. By definition of $\sqsubseteq$, since $T_{P^\Omega} = T_P$, the theorem holds: $lfp(T_{\bigcup_i P_i^{\Omega_i}}) \subseteq lfp(T_{\bigcup_i \mathcal{R}_i^{\subseteq}(P_i^{\Omega_i})})$.

6 Modular Resolution Methods

The previous section showed that resolution methods can be applied locally to parts of a DATALOG program. Here, we exhibit a number of resolution methods and relate them to some of the traditional techniques for modular analysis listed by Cousot and Cousot in [10].

6.1 Fixpoint Based Methods

One type of resolution techniques rely on the fixpoint computation of the least Herbrand model of DATALOG programs. These techniques require the ability to re-construct a DATALOG program from a Herbrand interpretation.

Definition 8. *The* extensional database *built from a Herbrand interpretation I is the program $edb(I)$ defined by $edb(I) = \{e \leftarrow \{\} \mid e \in I\}$.*

Worst Case Worst case analysis is a usual modular resolution method in which the worst assumption is made for the unknowns: they are assigned the top element of the lattice. The worst case consists in computing the fixpoint of this enriched system. For an open program P^Ω, we add an empty clause per open predicate symbol.

$$W_\Omega = \{p(x_1,\dots,x_n) \leftarrow \{\} \mid p \in \Omega, x_i \text{ are distinct variables}\}$$

Property 3. For any Herbrand interpretation I, we have $T_{W_\Omega}(I) = HB(\Omega, C)$

Because a modular resolution method must yield a program, the least Herbrand model of $P \cup W_\Omega$ is computed and translated back to a program. Theorem 2 states the correctness of this approximate resolution method.

Theorem 2. *The worst case resolution of an open program P^Ω is defined by $WC = edb(lfp(T_{P \cup W_\Omega}))$. It is a correct resolution method.*

$$P^\Omega \sqsubseteq WC^\Omega$$

Proof. Consider an arbitrary open program $Q^{\Omega'}$ and suppose that $P^\Omega \cup Q^{\Omega'}$ and $WC^\Omega \cup Q^{\Omega'}$ are defined. We show that $lfp(T_{P \cup Q}) \subseteq lfp(T_{WC \cup Q})$. To begin with, we prove the equality

$$lfp(T_{P \cup W_\Omega \cup Q}) = lfp(T_{WC \cup Q})$$

To do this, we apply several rewriting steps. First, we exploit the distributivity of T and the property 3 to rewrite the left-hand side to $lfp(\lambda I.T_P(I) \cup H \cup T_Q(I))$ where H denotes the Herbrand base generated from the predicate symbols in Ω ($H = HB(\Omega, C)$). In general, the fixpoint operator does not distribute with respect to $\cup$. However, in our case, at each iteration step the computation of $T_P(I) \cup H$ does not depend on Q. Intuitively, potential interactions are captured by H. As a result, we have $lfp(\lambda I.T_P(I) \cup H \cup T_Q(I)) = lfp(\lambda I.lfp(\lambda I.T_P(I) \cup H) \cup T_Q(I)$. We rewrite now the right-hand side of the equality ($lfp(T_{WC \cup Q})$). By definition of WC and distributivity of T, we have $lfp(T_{WC \cup Q}) = lfp(\lambda I.T_{edb(lfp(T_{P \cup W_\Omega}))}(I) \cup T_Q(I))$. Since we have $T_{edb(I_1)}(I_2) = I_1$, we obtain $lfp(\lambda I.T_{edb(lfp(T_{P \cup W_\Omega}))}(I) \cup T_Q(I)) = lfp(\lambda I.lfp(T_{P \cup W_\Omega}) \cup T_Q(I))$. The proof of the equality follows because $lfp(T_{P \cup W_\Omega}) = lfp(\lambda I.T_P(I) \cup H)$.

Given this equality, it remains to show that $lfp(T_{P \cup Q}) \subseteq lfp(T_{P \cup W_\Omega \cup Q})$. It is a direct consequence of the monotonicity of lfp and T. Hence, the theorem holds.

In practice, the worst case may lead to a very imprecise global analysis. However, if the original clauses are kept, some precision can be recovered *a posteriori* by techniques such as restarting the iteration [5] or iterating separate worst case analyses ([10] Sect. 8.2).

Partial Fixpoint Rather than a worst case assumption, we can make the assumption that free predicate symbols will never be given a definition and open predicate symbols will never be enriched by additional clauses. The minimal fixpoint of the program can be computed but will only constitute a correct resolution as long as the assumption is valid *i.e.,* nothing is added to the program. However, we obtain an exact resolution method if we *enrich* an open program with this least fixpoint.

Theorem 3. *Let P be an Ω-open program, $P^\Omega \equiv (P \cup edb(lfp(T_P)))^\Omega$.*

The practical interest of this method highly depends on the clauses in P. Indeed, if the partial fixpoint is empty, nothing is gained. Otherwise, the precomputed partial fixpoint will speed-up the convergence of future fixpoint iterations.

6.2 Unfolding and Minimization

Unfolding is a standard transformation of logic programs. Here we present exact resolution methods based on unfolding. Definition 9 recalls the unfolding operator *unf* as present in the literature, see *e.g.,* Levi [20].

Definition 9. *Let P and Q be* DATALOG *programs that do not share variable names*[1]*. The unfolding of P with respect to Q is a program defined by*

$$\begin{aligned} unf(P,Q) = \{H\sigma \leftarrow \textstyle\bigcup_{i\in[1,\ldots,n]} D'_i\sigma \mid \exists(H \leftarrow \{B_1,\ldots,B_n\} \in P). \\ \exists B'_1 \leftarrow D'_1 \in Q, \ldots, B'_n \leftarrow D'_n \in Q. \\ \sigma = mgu((B_1,\ldots,B_n),(B'_1,\ldots,B'_n))\} \end{aligned}$$

where mgu is the most general unifier operator.

Simple Unfolding In the context of logic programs, unfolding has been studied extensively (see *e.g.,* [20,11]). A well-known result is that a DATALOG program and its unfolding by itself have the same least Herbrand model.

$$lfp(T_P) = lfp(T_P^2) = lfp(T_{unf(P,P)})$$

To cope with open DATALOG programs, following Bossi *et al.* [4], we model incomplete knowledge by adjoining *tautologic* clauses.

Definition 10. *The tautological clauses Id of a set S of predicate symbols is defined by $Id_S = \{p(x_1,\ldots,x_n) \leftarrow p(x_1,\ldots,x_n) \mid p \in S\}$ where $x_1, \ldots, x_n$ are distinct variables.*

To get an exact resolution method, we unfold a DATALOG program with respect to itself enriched with *tautological* clauses for open predicates symbols.

Theorem 4. *Let P^Ω be an open program. $P^\Omega \equiv unf(P, P \cup Id_\Omega)$*

Minimization Syntactically distinct DATALOG programs may denote the same T operator. Obviously, such programs are equivalent.

$$T_{P_1} = T_{P_2} \Rightarrow P_1 \equiv P_2$$

For DATALOG programs, there exists a minimization procedure *Min* that computes a normal form such that $T_P = T_{Min(P)}$. Such minimization was first proposed by Chandra and Merlin [6] for conjunctive queries and then extended by Sagiv and Yannakakis [27] to sets of clauses. The minimization yields a reduced clause for which redundant atoms have been eliminated. The idea of the algorithm is to merge variables names to obtain a subset of the initial body. As an example (due to Chandra and Merlin and rephrased in DATALOG terms), the clause $H(x) \leftarrow \{R(x,v,w), R(x,z,w), S(u,w), S(u,v), S(y,w), S(y,v), S(y,z)\}$ is minimised to $H(x) \leftarrow \{R(x,v,w), S(u,w), S(u,v)\}$ by the substitution $[y \mapsto u, z \mapsto v]$. The immediate consequence operator T is stable with respect to such transformations. Hence, minimisation is an exact modular resolution method.

[1] To lift this restriction, a renaming is needed.

Iterative Unfolding The two previous transformations are complementary. Minimization procedure gives a normal form to non-recursive programs but ignore recursive dependencies while unfolding is one step unrolling of recursions. Iterating these transformations until fixpoint yields an exact modular resolution method.

Theorem 5. *Let P^Ω be an open-program, $P^\Omega \equiv lfp(\lambda q.Min(unf(P, q \cup Id_\Omega)))^\Omega$*

If unfolding were considered without minimization, the iteration would diverge for all recursive program. To improve convergence, it is desirable to normalize programs at each iteration step. If the open predicates are exactly the free variables (*i.e.,* there are no partially defined predicates) the fixpoint program is non-recursive (at each iteration step, it is only expressed in terms of the free variables). Naugthon [21] proved that there exists a non-recursive equivalent program if and only if the iteration converges. Such programs are said to be *bounded*. Unfortunately, boundedness is not decidable [14] even for programs with a single recursive predicate.

If necessary, termination can be enforced by means of a widening operator [9]. Divergence comes from the fact that, in a clause body, the length of atoms/variables dependancies cannot be bounded. Typically, we have sets of atoms like $\{x(o_1, o_2), x(o_2, o_3), \ldots, x(o_{n-1}, o_n)\}$. The idea of the widening proposed by Codish *et al.* [8,7] is to limit the length of such dependencies. They choose to trigger widening as soon as a dependency chain goes twice through the same predicate symbol at the same argument position. Each occurrence of the variable responsible for this (potential) cyclic dependency is then renamed to a fresh variable. This is a conservative transformation that ensure convergence.

Example 2. To illustrate unfolding, normalization and widening transformations, we extract an open set of clauses from Example 1.

$$P = \left\{ \begin{array}{l} self(o) \leftarrow \{main.call(o), lk(o, \underline{main}, \underline{main})\} \\ self(o) \leftarrow \{self(o'), fd(o', o)\} \end{array} \right\}$$

The set of open predicate symbols is $\{main.call, lk, fd\}$. By iterative unfolding (no minimization applies), an infinite ascending chain of programs is computed.

$$\begin{array}{ll} P_0 & = \{\} \\ P_1 & = \{self(o) \leftarrow \{main.call(o), lk(o, \underline{main}, \underline{main})\}\} \\ P_2 & = \left\{ \begin{array}{l} self(o) \leftarrow \{main.call(o), lk(o, \underline{main}, \underline{main})\} \\ self(o_2) \leftarrow \{main.call(o_1), lk(o_1, \underline{main}, \underline{main}), fd(o_1, o_2)\} \end{array} \right\} \\ \ldots & \\ P_{n+1} & = P_n \cup \left\{ self(o_n) \leftarrow \left\{ \begin{array}{l} main.call(o_1), lk(o_1, \underline{main}, \underline{main}), \\ fd(o_1, o_2), \ldots fd(o_{n-1}, o_n) \end{array} \right\} \right\} \end{array}$$

P_3 is widened by renaming the first occurrence of o_2 by α and the second by β.

$$\triangle(P_2) = \left\{ \begin{array}{l} self(o) \leftarrow \{main.call(o), lk(o, \underline{main}, \underline{main})\} \\ self(o_2) \leftarrow \{main.call(o_1), lk(o_1, \underline{main}, \underline{main}), fd(o_1, \alpha), fd(\beta, o_2)\} \end{array} \right\}$$

At the next iteration step, widening is also triggered and we obtain.

$$\triangle(P_3) = \left\{ \begin{array}{l} self(o) \leftarrow \{main.call(o), lk(o, \underline{main}, \underline{main})\} \\ self(o_2) \leftarrow \{main.call(o_1), lk(o_1, \underline{main}, \underline{main}), fd(o_1, \alpha), fd(\beta, o_2)\} \\ self(o) \leftarrow \left\{ \begin{array}{l} main.call(o_2), lk(o_1, \underline{main}, \underline{main}), \\ fd(o_1, \alpha), fd(\beta, \gamma), fd(\delta, o_2) \end{array} \right\} \end{array} \right\}$$

The last clause is minimized by renaming $[\beta \mapsto \delta, \gamma \mapsto o_2]$ and we obtain the following program

$$\left\{ \begin{array}{l} self(o) \leftarrow \{main.call(o), lk(o, \underline{main}, \underline{main})\} \\ self(o_2) \leftarrow \{main.call(o_1), lk(o_1, \underline{main}, \underline{main}), fd(o_1, \alpha), fd(\beta, o_2)\} \end{array} \right\}$$

which is stable by the iteration step. While the first clause is directly inherited from the initial clauses, the second is computed by the resolution process. Widening is responsible for the introduction of the fresh variables α and β. Without widening, there would be a path via *fd* fields between α and β objects.

7 Related Work

The use of logic languages such as DATALOG to specify static analyses is not new but seems to have stayed within the imperative and logic programming realm. The approach called *abstract compilation* evolves around the idea of translating a program into a set of Horn clauses that can then combined with particular queries to obtain information about the program being analysed. Reps proposed to use logic databases in order to have a demand-driven dataflow-analysis for free [25]. Hill and Spoto studied the use of logic programs to compactly model abstract denotations of programs [17]. They cite an experiment with class analysis but do not provide details.

Nielson and Seidl have proposed to use Alternation-Free Least Fixed Point Logic (ALFP) as a general formalism for expressing static analyses (in particular 0-CFA control-flow analysis [22]) in the Flow Logic framework. Hansen [16] shows how to encode a flow logic for an idealized version of Java Card. The ALFP logic is more expressive than DATALOG but as shown here, this added expressiveness does not seem required in the particular case of class analysis. It is unknown to us how our resolution techniques for open programs carry over to this more powerful logic.

The notion of context-sensitive analyses has been around for a long time. The article by Hornof and Noyé [18] describes various types of context-sensitivity for static analysis of imperative languages. For object-oriented languages, DeFouw *et al.* [15] describe a parameterized framework to define context-sensitive class analyses, but in a somewhat more operational setting than here. Jensen and Spoto [19] classify a number of low-cost, context-insensitive class analyses such as Rapid Type Analysis and 0-CFA in a language-independent abstract interpretation framework.

In their survey of modular analysis methods, Cousot and Cousot [10] write that modular analysis consists in computing a parameterized fixpoint

$$\lambda(p_1, \ldots, p_n).lfp(\lambda p'.f(p')(p_1, \ldots, p_n))$$

but note that a direct approach is not in general feasible. By restricting attention to a particular type of analysis that can be expressed in the simple specification language DATALOG we have been able to re-express this fixpoint computation as the solution of a set of open DATALOG clauses and to propose a number of resolution techniques that can help in this resolution.

Flanagan and Felleisen [13] developed a *componential set based analysis* for Scheme. Each module gives rise to a constraint set separately minimized under a notion of observational equivalence. For this particular constraint language, equivalence under observational equivalence is decidable (though computationally expensive). It means that a normal form for constraints exists . This is not the case for DATALOG, and hence the resolution techniques are not in general guaranteed to yield a normal form.

Rountev *et al.* [26] define a framework to refine the result of a whole program analysis by applying a more precise analysis on program fragments. Basically, whole program information is used to abstract the behaviour at the boundaries of the fragment. The pros of this technique is that the fragment is only analysed in contexts relvant for the whole program being analysed. The cons is that the fragment is to be re-analysed as other parts of the program change.

Bossi *et al.* [4] defined a compositional semantics for (open) logic programs for which the semantics domain is defined by *syntactic* objects: sets of clauses. Based on this semantics, Codish *et al.* [7] proposed a framework for the modular analysis of logic programs. Alike this line of work, our modular analysis does not require to give a semantics to open object-oriented programs. Our presentation does not even explicit a semantics for open DATALOG programs but formalizes what safe approximations are through the $\sqsubseteq$ relation. Anyway, at the end, our approaches converge since iterative unfolding of clauses (Section 6.2) is the semantics of open programs proposed by Bossi *et al.*

8 Conclusions

We have demonstrated the use of DATALOG as a specification language for class analysis of object-oriented languages, by showing how a variety of context-sensitive analyses can be expressed as instances of a common framework. For closed programs, this provides a straightforward implementation of the analysis through a bottom-up evaluator for DATALOG. We have also shown its use for developing modular program analyses. Analysis of program fragments gives rise to DATALOG programs with partially or undefined predicates. Such programs can be reduced using a number of iteration and normalisation operators, all expressible in the DATALOG framework.

As noted in the section on resolution by unfolding and minimisation (Section 6.2), the resolution might in certain cases need a widening operator to

enforce convergence. The next step in our work is to design suitable widenings in order to be able to experiment with the analysis of realistic code fragments. Another issue that needs to be treated formally is how to take into account the structuring mechanisms and scoping rules of the programming language (such as the visibility modifiers and package structure in Java) when determining what predicates can be considered closed and what must be kept open. A precise modeling of this is important since the more predicates can be considered closed, the more resolution can take place.

References

1. O. Agesen. Constraint-Based Type Inference and Parametric Polymorphism. In B. Le Charlier, editor, *Proc. of the 1st International Static Analysis Symposium*, volume 864 of *LNCS*, pages 78–100. Springer-Verlag, 1994.
2. K. R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 493–574. Elsevier, Amsterdam, 1990.
3. D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proc. of OOPSLA'96*, volume 31(10) of *ACM SIGPLAN Notices*, pages 324–341, New York, 1996. ACM Press.
4. A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. A Compositional Semantics for Logic Programs. *Theoretical Computer Science*, 122(1-2):3–47, 1994.
5. M. G. Burke and B. G. Ryder. A critical analysis of incremental iterative data flow analysis algorithms. *IEEE Transactions on Software Engineering*, 16(7):723–728, 1990.
6. A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proc. of the 9th ACM symposium on Theory of computing*, pages 77–90, 1977.
7. M. Codish, S. K. Debray, and R. Giacobazzi. Compositional analysis of modular logic programs. In *Proc. of the 20th ACM symposium on Principles of programming languages*, pages 451–464. ACM Press, 1993.
8. M. Codish, M. Falaschi, and K. Marriott. Suspension analysis for concurrent logic programs. *ACM Transactions on Programming Languages and Systems*, 16(3):649–686, 1994.
9. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proc. of the International Workshop Programming Language Implementation and Logic Programming,*, volume 631 of *LNCS*, pages 269–295. Springer, 1992.
10. P. Cousot and R.Cousot. Modular static program analysis, invited paper. In R.N. Horspool, editor, *Proc. of the 11th International Conference on Compiler Construction*, volume 2304 of *LNCS*, pages 159–178, Grenoble, France, April 2002. Springer.
11. F. Denis and J-P Delahaye. Unfolding, procedural and fixpoint semantics of logic programs. In *Proc. of the 8th Annual Symposium on Theoretical Aspects of Computer Science*, volume 480 of *LNCS*, pages 511–522, Hamburg, Germany, February 1991. Springer.

12. A. Diwan, J. E. B. Moss, and K. S. McKinley. Simple and Effective Analysis of Statically Typed Object-Oriented Programs. In *Proc. of OOPSLA'96*, volume 31(10) of *ACM SIGPLAN Notices*, pages 292–305, New York, 1996. ACM Press.
13. C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Transactions on Programming Languages and Systems*, 21(2):370–416, 1999.
14. H. Gaifman, H. Mairson, Y. Sagiv, and M. Y. Vardi. Undecidable optimization problems for database logic programs. In *Proc. Symposium on Logic in Computer Science*, pages 106–115, Ithaca, New York, jun 1987. IEEE Computer Society.
15. D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6):685–746, 2001.
16. R. R. Hansen. Flow logic for carmel. Technical Report Secsafe-IMM-001, IMM, Technical U. of Denamrk, 2002.
17. P. M. Hill and F. Spoto. Logic Programs as Compact Denotations. Proc. of the Fifth International Symposium on Practical Aspects of Declarative Languages, PADL '03, 2003.
18. L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: flow, context, and return sensitivity. *Theoretical Computer Science*, 248(1–2):3–27, 2000.
19. T. Jensen and F. Spoto. Class analysis of object-oriented programs through abstract interpretation. In F. Honsell and M. Miculan, editors, *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS'01)*, pages 261–275. Springer LNCS vol .2030, 2001.
20. G. Levi. Models, unfolding rules and fixpoint semantics. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proc. of the 5th International Conference and Symposium on Logic Programming*, pages 1649–1665, Seatle, 1988. ALP, IEEE, The MIT Press.
21. J. F. Naughton. Data independent recursion in deductive databases. *Journal of Computer and System Sciences*, 38(2):259–289, April 1989.
22. F. Nielson and H. Seidl. Control-flow analysis in cubic time. In *Proc. of European Symp. on Programming (ESOP'01)*, pages 252–268. Springer LNCS vol. 2028, 2001.
23. J. Palsberg and M. I. Schwartzbach. *Object-Oriented Type-Systems*. John Wiley & Sons, 1994.
24. J. Plevyak and A. Chien. Precise Concrete Type Inference for Object-Oriented Languages. In *Proc. of OOPSLA'94*, volume 29(10) of *ACM SIGPLAN Notices*, pages 324–340. ACM Press, October 1994.
25. T. Reps. Demand interprocedural program analysis using logic databases. In *Applications of Logic Databases*, pages 163–196, Boston, MA, 1994. Kluwer.
26. A. Rountev, B. G. Ryder, and W. Landi. Data-flow analysis of program fragments. In *Proc. of the 7th international symposium on Foundations of software engineering*, pages 235–252. Springer-Verlag, 1999.
27. Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM*, 27(4):633–655, 1980.
28. O. Shmueli. Decidability and expressiveness aspects of logic queries. In *Proc. of the 6th ACM symposium on Principles of database systems*, pages 237–249. ACM Press, 1987.
29. J. D. Ullman. *Principles of database and knowledge-base systems, volume 2*, volume 14 of *Principles of Computer Science*. Computer Science Press, 1988.