# Logic Programs as Abstract Domains

Fausto Spoto
Dipartimento di Informatica
Strada Le Grazie, 15, Ca' Vignal, 37134 Verona, Italy
spoto@sci.univr.it
Ph.: +39 045 8027076 Fax: +39 045 8027982

**Abstract**

We show that logic programs provide a compact and efficient representation of some domains for the abstract interpretation of imperative programs. Logic variables express here some regularity in the abstract behaviour of commands. We show that sign analysis and class analysis can be implemented through that representation. We compare its time and space costs with those of an implementation through ground relationships. This comparison shows that our technique leads to dramatic improvements both in time and in space.

## 1 Introduction

The use of denotational semantics as the basis for the abstract interpretation of imperative languages is appealing since it leads to clean and compositional static analyses. In that context, the abstract denotation of a piece of code is a function from the abstract properties of its input to those of its output. This *relational* approach can be naively implemented through a relational database or through a ground logic program, provided the domain of abstract properties is finite. However, it could be so large that this naive approach becomes useless in practice.

In this paper, we show that logic programs with variables can be used instead of ground logic programs or relationships to represent the abstract denotation of a piece of code. The advantage is that logic variables allow for a compact and efficient implementation of dependences between the abstract input and the abstract output of the code. For instance, the abstract values of the variables which are not used by a given command can be just copied from its abstract input to its abstract output, at least for a large class of static analyses. Similarly, an assignment statement copies the abstract values of the right hand side in the abstract value of the left hand side. Those dependences can be expressed through logic programs.

We apply here that technique to the sign analysis of integer variables through the domain defined in [19], and to the class analysis of object-oriented programs through the two domains formalised in [11] through abstract interpretation, and derived from the ideas in [5] and [18], respectively. We show that our technique results in faster and cheaper analyses than traditional implementations through ground logic programs.

```
int foo(a:int,c:int)
{
  let b:int in { ...  a:=b; ...}
}
```

(a)

```
int foo(a:int,c:int)
{
  let b:int in {...  a:=b-1; ...}
}
```

(b)

Figure 1: Two simple programs.

# 2 A Motivating Example

Consider the sign analysis of the program in Figure 1(a). This means that we want to approximate the sign (positive or negative) of the variables of type `int`. We use here the domain for sign analysis defined in [19]. We refer to [19] for a formal definition of the concrete domain and of the abstraction and concretisation maps.

**Definition 1.** A *typing* $\tau$ is a map giving some type to a set of variables, which is called its *domain*, is written $\mathsf{dom}(\tau)$ and is lexicographically ordered. □

**Definition 2.** For every typing $\tau$, we define

$$S_\tau = \{\mathsf{empty}\} \cup \{\varsigma : \mathsf{dom}(\tau) \mapsto \{*, +, -, u\} \mid \varsigma(v) = * \text{ iff } \tau(v) \neq \mathtt{int}\} \ .$$

We define $\preceq$ as the minimal reflexive relation over $\{*, +, -, u\}$ such that $+ \preceq u$ and $- \preceq u$. The set $S_\tau$ is partially ordered w.r.t. $\sqsubseteq$ which is such that $\mathsf{empty} \sqsubseteq s$ for every $s \in S_\tau$ and $\varsigma_1 \sqsubseteq \varsigma_2$ if for every $v \in \mathsf{dom}(\tau)$ we have $\varsigma_1(v) \preceq \varsigma_2(v)$. □

The idea underlying the domain $S_\tau$ is that the variables of type `int` are approximated by $+$ if they are positive or zero, they are approximated by $-$ if they are negative and by $u$ if their sign is unknown. The other variables[1] are approximated by a don't care mark $*$. We write an element $\varsigma \in S_\tau \setminus \{\mathsf{empty}\}$ as a Prolog list $[\mathtt{x_1}, \ldots, \mathtt{x_n}]$ where $\mathsf{dom}(\tau) = \{v_1, \ldots, v_n\}$, $v_i$ is the $i$th variable in lexicographical order among $\{v_1, \ldots, v_n\}$ and $\varsigma(v_i) = x_i$ for every $i = 1, \ldots, n$.

In the program of Figure 1(a), the denotation of the assignment `a:=b` is a function from the abstract properties of the variables before the assignment to the abstract properties of the variables after it. Since exactly the four variables `a`, `b`, `c` and `foo` are addressable in that program point[2] and they all have type `int`, that denotation is a function $f : S_\tau \mapsto S_\tau$ with $\tau = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{foo} \mapsto \mathtt{int}\}$ which can be represented as

$$\begin{array}{rcl@{\qquad}rcl}
\mathtt{empty} & \to & \mathtt{empty} & & & \\
[+,+,+,+] & \to & [+,+,+,+] & [+,+,+,-] & \to & [+,+,+,-] \\
[+,+,-,+] & \to & [+,+,-,+] & [+,+,-,-] & \to & [+,+,-,-] \\
[+,-,+,+] & \to & [-,-,+,+] & [+,-,+,-] & \to & [-,-,+,-] \\
[+,-,-,+] & \to & [-,-,-,+] & [+,-,-,-] & \to & [-,-,-,-] \\
[-,+,+,+] & \to & [+,+,+,+] & [-,+,+,-] & \to & [+,+,+,-] \\
[-,+,-,+] & \to & [+,+,-,+] & [-,+,-,-] & \to & [+,+,-,-] \\
[-,-,+,+] & \to & [-,-,+,+] & [-,-,+,-] & \to & [-,-,+,-] \\
[-,-,-,+] & \to & [-,-,-,+] & [-,-,-,-] & \to & [-,-,-,-]
\end{array} \qquad (1)$$

[1] In [19] all variables are assumed to have type `int`. Here we generalise the domain $S_\tau$.

[2] The variable `foo` holds the return value of the function, like in Pascal.

Note that $f$ is not explicitly specified for the elements of $S_\tau$ binding some variables to $u$. This is because those values can be recovered from those provided above [12, 17, 19] (see Definition 5 later). For instance, $f([+,\mathtt{u},+,+]) = f([+,+,+,+]) \sqcup f([+,-,+,+]) = [+,+,+,+] \sqcup [-,-,+,+] = [\mathtt{u},\mathtt{u},+,+]$.

The above representation of the function $f$ can be naturally seen as a ground logic program stating an input/output relationship:

$$\mathtt{io(empty,empty)}.\quad \mathtt{io}([+,+,+,+],[+,+,+,+]).\quad \mathtt{io}([+,+,+,-],[+,+,+,-]).\quad \ldots$$

Its size grows exponentially with the number of program variables.

There is some regularity in the representation (1). For instance, $f$ does not change the values of `b`, `c` and `foo`. Moreover, it copies the value of `b` in the value of `a`. We can therefore represent $f$ *by using variables* as

$$\mathtt{empty} \rightarrow \mathtt{empty} \qquad [\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{Foo}] \rightarrow [\mathtt{B},\mathtt{B},\mathtt{C},\mathtt{Foo}] \tag{2}$$

which is smaller than (1) and whose size grows linearly with the number of program variables. This notation can be naturally seen as isomorphic to the logic program:

$$\mathtt{io(empty,empty)}. \qquad \mathtt{io}([\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{Foo}],[\mathtt{B},\mathtt{B},\mathtt{C},\mathtt{Foo}]). \tag{3}$$

From now on, an arrow notation like (2) must be seen as a more evocative notation for a logic program like (3).

Consider now the program in Figure 1(b). The denotation of the assignment `a:=b-1` can be represented by the following logic program:

$$\mathtt{empty} \rightarrow \mathtt{empty} \qquad [\mathtt{A},-,\mathtt{C},\mathtt{Foo}] \rightarrow [-,-,\mathtt{C},\mathtt{Foo}] \qquad [\mathtt{A},+,\mathtt{C},\mathtt{Foo}] \rightarrow [\mathtt{u},+,\mathtt{C},\mathtt{Foo}] \tag{4}$$

Note that program (4) is slightly more complex than (2), but is still much more compact than an exhaustive representation. Moreover, its size still grows linearly with the number of program variables.

# 3 The Framework of Analysis

We assume familiarity with logic programming [1] and abstract interpretation [3, 4]. Given a function $f$, the notation $f[t/n]$ represents the function $f$ modified in such a way that it binds $t$ to $n$. The domain of $f[t/n]$ is that of $f$ expanded with $n$.

We briefly present here a simplified version of the framework of [19] for the localised static analysis of imperative programs. Its distinguishing feature is that the resulting static analyses have a cost in time and space proportional to the number of program points of interest. This is why we speak of a *localised* analysis framework. Moreover, that framework is based on a denotational semantics and is thus compositional. Note that we simplify the framework of [19] here, in the sense that we consider just input/output denotations without information at internal program points. However, the extension of what we are going to describe to the full-featured *watchpoint semantics* of [19] does not introduce any technical problem.

From the point of view of abstract interpretation, the framework of [19] is appealing since it is based on the compilation of the high-level constructs (conditionals,

$$[op]^a : A_{\tau_1,\cdots,\tau_n,\tau},\ \text{if } op : (\Sigma_{\tau_1} \times \cdots \times \Sigma_{\tau_n}) \mapsto \Sigma_\tau$$
$$\circ^a : \left(A_{\tau_1,\ldots,\tau_n,\tau'} \times A_{\tau,\tau_1} \cdots \times A_{\tau,\tau_n}\right) \mapsto A_{\tau,\tau'}$$

$$[op]^a = \lambda d.\alpha^D(op(\gamma^D(d))) \qquad T \circ^a (T_1,\ldots,T_n) = \lambda d.T(T_1(d),\ldots,T_n(d)) \ .$$

Figure 2: Signature and implementation of the combinators.

loops, function calls) into a very small set of *combinators* which work over *denotations*. Thus, an abstract semantics is obtained by simply using abstract denotations and the abstract combinators induced by the theory of abstract interpretation. Just one abstract domain (that of abstract denotations) is used.

We take from [19] the following (simplified) definitions and results. The concrete states $\Sigma_\tau$ are maps from variables to values. Their formal definition can be found in [19] (and has been made more concrete in [11]) and is not relevant to this paper.

An element of an abstract domain is (*gamma-*)*union-reducible* if its concretisation if the union of the concretisations of the elements which precede it in the partial ordering. This is a stronger notion than that of *join-reducible* element given in [12].

**Definition 3.** For $\tau \in Typing$, let $\langle D_\tau, \sqsubseteq\rangle$ be a complete lattice and $\alpha^{D_\tau}$ and $\gamma^{D_\tau}$ the abstraction and concretisation maps of a Galois insertion from $\langle \wp(\Sigma_\tau), \subseteq\rangle$ to $\langle D_\tau, \sqsubseteq\rangle$. We say that $d \in D_\tau$ is ($\gamma$-)*union-reducible* if

$$\gamma(d) = \cup_{d' \sqsubset d} \gamma(d') \ .$$

Otherwise, we say that $d$ is *union-irreducible*. The set of the union-irreducible elements of $D_\tau$ is denoted by $\mathsf{ui}(D_\tau)$. □

**Example 4.** The union-irreducible elements of the domain $S_\tau$ of Definition 2 are $\mathsf{empty}$ and those $\varsigma \in S_\tau$ such that $\varsigma(v) \neq u$ for every $v \in \mathsf{dom}(\tau)$. If, instead, $\varsigma(v) = u$ for exactly one $v \in \mathsf{dom}(\tau)$, it can be shown that (for $\gamma^{S_\tau}$, see [19])

$$\gamma(\varsigma) = \gamma(\varsigma[+/v]) \cup \gamma(\varsigma[-/v]) \cup \gamma(\mathsf{empty}) = \cup_{s \sqsubset \varsigma} \gamma(s) \ .$$

**Definition 5.** Given $n \geq 1$ and $\tau_1,\ldots,\tau_n,\tau \in Typing$, let $A_{\tau_1,\ldots,\tau_n,\tau}$ be

$$\left\{ a \in (D_{\tau_1} \times \cdots \times D_{\tau_n}) \mapsto D_\tau \;\middle|\; \begin{array}{l} a \text{ is monotonic and if } d_i \text{ is union-reducible then} \\ a(d_1,\ldots,d_i,\ldots,d_n) = \bigsqcup^{D_\tau}_{d' \sqsubset d_i} a(d_1,\ldots,d',\ldots,d_n) \end{array} \right\},$$

i.e., *denotations* in $A_{\tau_1,\ldots,\tau_n,\tau}$ are identified by the union-irreducible elements. This set is a complete lattice w.r.t. the pointwise extension of $\sqsubseteq$. □

The fact that denotations are uniquely identified by the union-irreducible elements is similar to the use of component-wise additive denotations [12].

**Proposition 6 ([19]).** *An abstract semantics for the properties expressed by $D_\tau$ can be defined through the combinators on denotations of Figure 2.* □

$$\mathsf{nop}^s_\tau(\varsigma) = \varsigma \qquad (\mathsf{get\_int}^i_\tau)^s(\varsigma) = \begin{cases} \varsigma[+/res] & \text{if } i \geq 0 \\ \varsigma[-/res] & \text{if } i < 0 \end{cases}$$

$$(\mathsf{get\_var}^v_\tau)^s(\varsigma) = \varsigma[\varsigma(v)/res] \qquad (\mathsf{put\_var}^v_\tau)^s(\varsigma) = \varsigma[\varsigma(res)/v]|_{-res}$$

$$=^s_\tau(\varsigma_1)(\varsigma_2) = \begin{cases} \varsigma_2[-/res] & \text{if } \varsigma_1(res) \neq \varsigma_2(res) \\ \varsigma_2[u/res] & \text{otherwise} \end{cases} \qquad +^s_\tau(\varsigma_1)(\varsigma_2) = \begin{cases} \varsigma_2[\varsigma_1(res)/res] & \text{if } \varsigma_1(res) = \varsigma_2(res) \\ \varsigma_2[u/res] & \text{otherwise} \end{cases}$$

$$(\mathsf{restrict}^{vs}_\tau)^s(\varsigma) = \varsigma|_{-vs} \qquad (\mathsf{expand}^{v:t}_\tau)^s(\varsigma) = \varsigma[+/v]$$

$$\mathsf{is\_true}^s_\tau(\varsigma) = \begin{cases} \mathsf{empty} & \text{if } \varsigma(res) = - \\ \varsigma[+/res] & \text{otherwise} \end{cases} \qquad \mathsf{is\_false}^s_\tau(\varsigma) = \begin{cases} \mathsf{empty} & \text{if } \varsigma(res) = + \\ \varsigma[-/res] & \text{otherwise} \end{cases}$$

$$\cup^s_\tau(\mathsf{empty})(x) = \cup^s_\tau(x)(\mathsf{empty}) = x \qquad \cup^s_\tau(\varsigma_1)(\varsigma_2) = \lambda v \in \mathsf{dom}(\tau).\begin{cases} \varsigma_1(v) & \text{if } \varsigma_1(v) = \varsigma_2(v) \\ u & \text{otherwise.} \end{cases}$$

Figure 3: Some abstract operations over the domain of signs.

The operations *op* of Figure 2 work over concrete states and implement the basic constructs of the language (assignment, access to variables, arithmetic operations, comparisons, tests, or even object-oriented operations like field access). Our abstract combinators need their optimal abstract counterparts $\alpha^D op \gamma^D$. Figure 3 defines them for the case of sign analysis (Definition 2). To better understand the operations in Figure 3, we recall that in conditionals a non-negative integer stands for true, a negative integer for false. Moreover, the variable $res$ is an *accumulator* which holds intermediate results. If not otherwise stated, those operations should be implicitly considered strict on $\mathsf{empty}$. Note that they are defined on the union-irreducible elements only (Definition 5). The operations in Figure 3 are taken from [19].

# 4 Logic Programs as Abstract Domains

In Section 2 we have shown that logic programs can be used to represent abstract denotations (Definition 5) in a compact way. The idea is that a variable stands for all its instantiations compatible with its *type*.

**Definition 7.** Let $\tau \in Typing$. Assume that we are able to represent every element of $D_\tau$ with a ground term. In the following, an element will stand for its representation and vice versa. Assume that we have a set $D^*_\tau$ of (possibly non-ground) terms with variables of different types. We say that the term $d^* \in D^*_\tau$ is union-irreducible if it has an instance $d \in \mathsf{ui}(D_\tau)$. Otherwise, $d^*$ is said to be union-reducible. The set of the union-irreducible elements of $D^*_\tau$ is denoted by $\mathsf{ui}(D^*_\tau)$. □

**Definition 8.** Let $\tau \in Typing$ and consider the terms in $D^*_\tau$. We assume that for every type $T$ for the variables in those terms we have a non-empty set $P(T)$ of terms. A substitution $\sigma$ is *legal* if $\sigma(v) \in P(T)$ where $T$ is the type of $v$. The set of legal substitutions is denoted by $\Lambda$. Its grounding subset by $\Lambda^g$.

Given $d^* \in D^*_\tau$, we define its set of *legal ground instantiations* $\xi(d^*) = \{d^*\sigma \mid \sigma \in \Lambda^g\}$. We require that $D^*_\tau$ is closed w.r.t. $\xi$ and that if $d^* \in \mathsf{ui}(D^*_\tau)$ then $\xi(d^*) = \{d \in \mathsf{ui}(D_\tau) \mid d \text{ is an instance of } d^*\}$. □

**Definition 9.** Let $n \geq 1$ and $\tau_1, \ldots, \tau_n, \tau \in Typing$. A $(\tau_1, \ldots, \tau_n, \tau)$-*clause* is $l^*_1, \ldots, l^*_n \rightarrow r^*$ where the *tail* $l^*_1, \ldots, l^*_n$ is such that $l^*_i \in \mathsf{ui}(D^*_{\tau_i})$ for every $i =$

$1, \ldots, n$, and the *head* $r^*$ is such that $r^* \in D^*_\tau$. Moreover, we require that the variables in $r^*$ occur in $l^*_1, \ldots, l^*_n$.

The $\xi$-notation is extended to tails as $\xi(l^*_1, \ldots, l^*_n) = \{l^*_1\sigma, \ldots, l^*_n\sigma \mid \sigma \in \Lambda^g\}$. □

**Example 10.** Consider the domain $S_\tau$ of Definition 2. We have already introduced the ground terms `empty` and $[\mathtt{x_1}, \ldots, \mathtt{x_n}]$ ($x_i \in \{*, +, -, \mathtt{u}\}$) to represent its elements. We define a set $Vars$ of variables, all of the same type, and the set

$$S^*_\tau = \{\mathtt{empty}\} \cup \left\{ [\mathtt{x_1}, \ldots, \mathtt{x_n}] \;\middle|\; \begin{array}{l} \mathtt{x_i} \in \{*, +, -, \mathtt{u}\} \cup Vars \text{ for } i = 1, \ldots, n \\ \mathtt{x_i} = * \text{ if and only if } \tau(v) \neq \mathtt{int} \\ \text{where } v \text{ is the } i\text{th variable in } \mathsf{dom}(\tau) \end{array} \right\} . \quad (5)$$

If $\tau = \{a, b, c, d \mapsto \mathtt{int}\}$, the set $S^*_\tau$ contains the non-ground union-irreducible term $[+, -, \mathtt{X}, \mathtt{Y}]$, as well as the non-ground union-reducible term $[+, \mathtt{u}, \mathtt{X}, \mathtt{Y}]$. Moreover, $[+, -, \mathtt{X}, \mathtt{Y}] \to [\mathtt{u}, \mathtt{Y}, +, +]$ is a $(\tau, \tau)$-clause. □

**Example 11.** A set of legal substitutions for a term in $S^*_\tau$ (Equation (5)) is

$$\{\sigma \mid \text{for every } v \in Vars \text{ we have } \sigma(v) \in Vars \cup \{+, -\}\} ,$$

i.e., we do not allow variables to be bound to `u`. This is because a set of legal substitutions must not lose union-irreducibility (Definition 8). □

**Definition 12.** The set $A^*_{\tau_1, \ldots, \tau_n, \tau}$ of *compact denotations* is made of sets $\{l_1 \to r_1, \ldots, l_m \to r_m\}$ of $(\tau_1, \ldots, \tau_n, \tau)$-clauses such that

- $\xi(l_i) \cap \xi(l_j) = \emptyset$ for every $i, j \in 1, \ldots, m$ with $i \neq j$ (tails are disjoint),
- $\cup_{i=1,\ldots,m} \xi(l_i) = \{d_1, \ldots, d_n \mid d_i \in \mathsf{ui}(D_{\tau_i}),\ i = 1, \ldots, n\}$ (tails are exhaustive).

A compact denotation is mapped into a ground one by $\xi : A^*_{\tau_1, \ldots, \tau_n, \tau} \mapsto A_{\tau_1, \ldots, \tau_n, \tau}$:

$$\xi(a^*) = \{(l^*_1, \ldots, l^*_n \to r^*)\sigma \mid l^*_1, \ldots, l^*_n \to r^* \in a^* \text{ and } \sigma \in \Lambda^g\} .$$

Having defined this notion of *compact* representation, the next step needed to define a static analysis consists in finding elements of $A^*$ representing the abstract counterparts of the *op* operations (denotations) used in Figure 2 (namely, in the case of sign analysis, we look for elements of $A^*$ representing the operations of Figure 3). Moreover, we need an operation $\circ^*$ which mimics over $A^*$ what $\circ^a$ does over $A$.

Figure 4 shows some of those abstract operations in the case of sign analysis assuming that $\tau$ is such that $\mathsf{dom}(\tau) = \{a, b, c\}$ for all operations except for $\mathsf{put\_var}^s$, $=^s$ and $\mathsf{is\_true}^s$, for which we assume that $\mathsf{dom}(\tau) = \{a, b, c, res\}$, and except for $\cup^s$, for which we assume that $\mathsf{dom}(\tau) = \{a, b\}$. All variables are assumed to have type `int`. Note that the tails of the clauses are union-irreducible (Definition 9).

More generally, those operations are implemented by the domain specific operations whose signature is given in Figure 5 (last four lines).

Consider the definition of $\circ^*$ now. If we have compact representations for the denotations $T, T_1, \ldots, T_n$, Figure 2 suggests that a compact representation for $T \circ^a (T_1, \ldots, T_n)$ can be obtained through some sort of folding of clauses. But the inputs of a representation are union-irreducible elements, while $T_1(d), \ldots, T_n(d)$ are not

| | |
|---|---|
| $\mathsf{nop}^s_\tau$ | $\mathtt{empty} \to \mathtt{empty} \quad [\mathtt{A},\mathtt{B},\mathtt{C}] \to [\mathtt{A},\mathtt{B},\mathtt{C}]$ |
| $(\mathsf{get\_int}^i_\tau)^s$, $i \geq 0$ | $\mathtt{empty} \to \mathtt{empty} \quad [\mathtt{A},\mathtt{B},\mathtt{C}] \to [\mathtt{A},\mathtt{B},\mathtt{C},+]$ |
| $(\mathsf{get\_int}^i_\tau)^s$, $i < 0$ | $\mathtt{empty} \to \mathtt{empty} \quad [\mathtt{A},\mathtt{B},\mathtt{C}] \to [\mathtt{A},\mathtt{B},\mathtt{C},-]$ |
| $(\mathsf{get\_var}^v_\tau)^s$<br>assuming that $v \equiv a$ | $\mathtt{empty} \to \mathtt{empty}$<br>$[\mathtt{A},\mathtt{B},\mathtt{C}] \to [\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{A}]$ |
| $(\mathsf{put\_var}^v_\tau)^s$<br>assuming that $v \equiv a$, | $\mathtt{empty} \to \mathtt{empty}$<br>$[\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{Res}] \to [\mathtt{Res},\mathtt{B},\mathtt{C}]$ |
| $=^s_\tau$ | $\mathtt{empty},\mathtt{empty} \to \mathtt{empty}$<br>$\mathtt{empty},[\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{Res}] \to \mathtt{empty}$<br>$[\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{Res}],\mathtt{empty} \to \mathtt{empty}$<br>$[\mathtt{A}_1,\mathtt{B}_1,\mathtt{C}_1,+],[\mathtt{A}_2,\mathtt{B}_2,\mathtt{C}_2,-] \to [\mathtt{A}_2,\mathtt{B}_2,\mathtt{C}_2,-]$<br>$[\mathtt{A}_1,\mathtt{B}_1,\mathtt{C}_1,-],[\mathtt{A}_2,\mathtt{B}_2,\mathtt{C}_2,+] \to [\mathtt{A}_2,\mathtt{B}_2,\mathtt{C}_2,-]$<br>$[\mathtt{A}_1,\mathtt{B}_1,\mathtt{C}_1,\mathtt{Res}],[\mathtt{A}_2,\mathtt{B}_2,\mathtt{C}_2,\mathtt{Res}] \to [\mathtt{A}_2,\mathtt{B}_2,\mathtt{C}_2,\mathtt{u}]$ |
| $(\mathsf{restrict}^{vs}_\tau)^s$<br>assuming $vs = \{a\}$ | $\mathtt{empty} \to \mathtt{empty}$<br>$[\mathtt{A},\mathtt{B},\mathtt{C}] \to [\mathtt{B},\mathtt{C}]$ |
| $\mathsf{is\_true}^s_\tau$ | $\mathtt{empty} \to \mathtt{empty}$<br>$[\mathtt{A},\mathtt{B},\mathtt{C},-] \to \mathtt{empty} \quad [\mathtt{A},\mathtt{B},\mathtt{C},+] \to [\mathtt{A},\mathtt{B},\mathtt{C},+]$ |
| $\cup^s_\tau$ | $\mathtt{empty},\mathtt{empty} \to \mathtt{empty} \quad \mathtt{empty},[\mathtt{A},\mathtt{B}] \to [\mathtt{A},\mathtt{B}]$<br>$[\mathtt{A},\mathtt{B}],\mathtt{empty} \to [\mathtt{A},\mathtt{B}] \quad [\mathtt{A},\mathtt{B}],[\mathtt{A},\mathtt{B}] \to [\mathtt{A},\mathtt{B}]$<br>$[\mathtt{A},+],[\mathtt{A},-] \to [\mathtt{A},\mathtt{u}] \quad [\mathtt{A},-],[\mathtt{A},+] \to [\mathtt{A},\mathtt{u}]$<br>$[+,\mathtt{B}],[-,\mathtt{B}] \to [\mathtt{u},\mathtt{B}] \quad [-,\mathtt{B}],[+,\mathtt{B}] \to [\mathtt{u},\mathtt{B}]$<br>$[+,+],[-,-] \to [\mathtt{u},\mathtt{u}] \quad [+,-],[-,+] \to [\mathtt{u},\mathtt{u}]$<br>$[-,+],[+,-] \to [\mathtt{u},\mathtt{u}] \quad [-,-],[+,+] \to [\mathtt{u},\mathtt{u}]$ |

Figure 4: The compact representations over $A^*$ of the operations of Figure 3.

necessarily union-irreducible. If their are not, Definition 5 allows us to compute $T \circ^a (T_1, \ldots, T_n)$ from the values provided by $T$ for the union-irreducible elements.

But things are still more difficult because of the variables we allowed in clauses. For instance, we can fold the clause $[\mathtt{A},+] \to [-,\mathtt{u}]$ in the clauses $[\mathtt{B},+] \to [-]$ and $[\mathtt{B},-] \to [+]$ provided that we bind $\mathtt{B}$ to $-$. The result is that for the abstract input $[\mathtt{A},+]$ the abstract output is $[-]$ *or* $[+]$. Then Definition 5 allows us to conclude that the clause $[\mathtt{A},+] \to [\mathtt{u}]$ is the result of those two foldings.

## 4.1 The Simpler Case of $\circ^*$.

The above considerations suggest an algorithm for $\circ^*$. We tackle the simpler case of computing $T \circ^* (T_1)$ for the moment. Our algorithm tries to fold every clause $l \to r \in T$ in every clause $l' \to r' \in T_1$, by binding the variables in $r'$ and $l$ in such a way that the resulting instance of $l$ *entails* that of $r'$ (see `domain_entails` in Figure 5). Then it computes the join over $D$ of the heads of the clauses with the same tails. The first part of this algorithm is accomplished by the following Prolog program:

```
comb_compose(_Tau,Tau1,_Tauprime,T,T1,Result):-findall(In1->Out2,
 (member(In1->Out1,T1),member(In2->Out2,T),domain_entails(Tau1,Out1,In2)),
 Result).
```

**Example 13.** Consider the domain $S^*_\tau$ of Equation (5). Let $\tau = \{a, b \mapsto \mathtt{int}\}$. In our implementation (Section 5) the call `domain_entails(`$\tau$`,[+,u],[X,-])` succeeds

| Operation | Semantics |
|---|---|
| `domain_entails(`$\tau$`,A,B)` with $\mathtt{A} \in D^*_\tau$ and $\mathtt{B} \in \mathsf{ui}(D^*_\tau)$ | computes $\theta_1, \ldots, \theta_n$ legal such that $\{\mathtt{B}\sigma \mid \sigma \in \Lambda^g,\ \mathtt{B}\sigma \sqsubseteq_{D_\tau} \mathtt{A}\sigma\} = \cup_{i=1,\ldots,n}\xi(\mathtt{B}\theta_i)$ |
| `domain_intersect_ui(`$\tau$`,A,B)` with $\mathtt{A}, \mathtt{B} \in \mathsf{ui}(D^*_\tau)$ | computes $\theta_1, \ldots, \theta_n$ legal such that for every $\sigma \in \Lambda^g$ we have $\mathtt{A}\theta_i\sigma = \mathtt{B}\theta_i\sigma$ and $\xi(\mathtt{A}) \cap \xi(\mathtt{B}) = \cup_{i=1,\ldots,n}\xi(\mathtt{A}\theta_i)$ |
| `domain_subtract_ui(`$\tau$`,A,B)` with $\mathtt{A}, \mathtt{B} \in \mathsf{ui}(D^*_\tau)$ and $\xi(\mathtt{A}) \cap \xi(\mathtt{B}) \neq \emptyset$ | computes $\theta_1, \ldots, \theta_n$ legal such that $\cup_{i=1,\ldots,n}\xi(\mathtt{A}\theta_i) = \xi(\mathtt{A}) \setminus \xi(\mathtt{B})$ |
| `domain_lub(`$\tau$`,A,B,L)` with $\mathtt{A}, \mathtt{B} \in D^*_\tau$ | computes $\theta_1, \ldots, \theta_n$ legal such that $\cup_{i=1,\ldots,n}\xi(\mathtt{A}\theta_i) = \xi(\mathtt{A})$, $\cup_{i=1,\ldots,n}\xi(\mathtt{B}\theta_i) = \xi(\mathtt{B})$ and for every $i = 1, \ldots, n$ and $\sigma \in \Lambda^g$ we have $\mathtt{L}\theta_i\sigma = \mathtt{A}\theta_i\sigma \sqcup_{D_\tau} \mathtt{B}\theta_i\sigma$ |
| `domain_bottom(`$\tau_{in}$`,`$\tau_{out}$`,B)` | computes $\mathtt{B} \in A^*_{\tau_{in},\tau_{out}}$ s.t. $\xi(\mathtt{B}) = \bot_{A_{\tau_{in},\tau_{out}}}$ |
| `domain_the_same(`$\tau_{in}$`,`$\tau_{out}$`,A,B)` | checks whether $\xi(\mathtt{A}) = \xi(\mathtt{B})$, with $\mathtt{A}, \mathtt{B} \in A^*_{\tau_{in},\tau_{out}}$ |
| `domain_nop(`$\tau$`,N)` | computes $\mathtt{N} \in A^*_{\tau,\tau}$ s.t. $\xi(\mathtt{N}) = [\mathsf{nop}_\tau]^a$ |
| `domain_get_int(`$\tau$`,I,GI)` | computes $\mathtt{GI} \in A^*_{\tau,\tau[\mathtt{int}/res]}$ s.t. $\xi(\mathtt{GI}) = [\mathsf{get\_int}^I_\tau]^a$ |
| $\vdots$ | $\vdots$ |
| `domain_union(`$\tau$`,U)` | computes $\mathtt{U} \in A^*_{\tau,\tau,\tau}$ s.t. $\xi(\mathtt{U}) = [\cup_\tau]^a$ |

Figure 5: Domain-specific operations.

and binds `X` to `+`. The call `domain_entails(`$\tau$`,[+,u],[+,X])` succeeds twice and binds `X` to `+` the first time and `X` to `-` the second time. Note that we do not allow `X` to be bound to `u` (Example 11). The call `domain_entails(`$\tau$`,[+,u],[+,+])` just succeeds, while the call `domain_entails(`$\tau$`,[+,+],[+,-])` fails. □

**Example 14.** Consider the computation of

$$\left\{ \begin{array}{ll} \mathtt{empty} \rightarrow \mathtt{empty} & \\ [\mathtt{A},+] \rightarrow [-] & [\mathtt{A},-] \rightarrow [+] \end{array} \right\} \circ^* \left( \left\{ \begin{array}{ll} \mathtt{empty} \rightarrow \mathtt{empty} & \\ [\mathtt{A},+] \rightarrow [-,\mathtt{u}] & [\mathtt{A},-] \rightarrow [+,+] \end{array} \right\} \right) .$$

The above algorithm for $\circ^*$ proceeds as follows. It considers every clause of the program on the right. Thus it starts with $\mathtt{empty} \rightarrow \mathtt{empty}$, whose head is entailed by the tail of the clause $\mathtt{empty} \rightarrow \mathtt{empty}$ of the program on the left. Folding those clauses results in the clause $\mathtt{empty} \rightarrow \mathtt{empty}$ itself. It then considers the clause $[\mathtt{A},+] \rightarrow [-,\mathtt{u}]$ on the right. Its head $[-,\mathtt{u}]$ is entailed by both the tails $[\mathtt{A},+]$ and $[\mathtt{A},-]$ of the program on the left, provided we bind the variable $\mathtt{A}$ of the clauses on the left to $-$. This results in *two* clauses $[\mathtt{A},+] \rightarrow [-]$ and $[\mathtt{A},+] \rightarrow [+]$. Finally, it considers the clause $[\mathtt{A},-] \rightarrow [+,+]$ on the right. Its head $[+,+]$ is entailed by the tail $[\mathtt{A},+]$ on the left provided we bind this last $\mathtt{A}$ to $+$. This results in the clause $[\mathtt{A},-] \rightarrow [-]$.

In conclusion, the first part of the computation above yields the program $\{\mathtt{empty} \rightarrow \mathtt{empty}\ [\mathtt{A},+] \rightarrow [-]\ [\mathtt{A},+] \rightarrow [+]\ [\mathtt{A},-] \rightarrow [-]\}$. If we compute the join of the clauses with the same tails, we obtain $\{\mathtt{empty} \rightarrow \mathtt{empty} \quad [\mathtt{A},+] \rightarrow [\mathtt{u}] \quad [\mathtt{A},-] \rightarrow [-]\}$. □

However, the above algorithm can yield clauses whose tails just overlap.

**Example 15.** The above algorithm over the domain $S_\tau$ computes

$$\left\{\begin{array}{l}\texttt{empty} \to \texttt{empty}\\ [\texttt{A}, +] \to [+]\\ [+, -] \to [-]\\ [-, -] \to [+]\end{array}\right\} \circ^* \left(\left\{\begin{array}{l}\texttt{empty} \to \texttt{empty}\\ [\texttt{X}] \to [\texttt{X}, \texttt{u}]\end{array}\right\}\right) = \left\{\begin{array}{l}\texttt{empty} \to \texttt{empty}\\ [\texttt{X}] \to [+]\\ [+] \to [-]\\ [-] \to [+]\end{array}\right\} .$$

The tail of $[\texttt{X}] \to [+]$ overlaps with that of both $[+] \to [-]$ and $[-] \to [+]$. □

To solve this problem, we use a procedure `make_disjunctive` which uses three domain specific operations: `domain_intersect_ui` computes the intersection of the tails, and `domain_subtract_ui` computes their difference. When the tails are finally mutually disjoint, it uses the `domain_lub` operation to compute the join of the heads of the clauses with the same tails (Figure 5). The `make_disjunctive` procedure can be consulted in the module `combinators` of our implementation (Section 5).

**Example 16.** In the case of Example 15, the call `domain_intersect_ui(`$\tau$`,[X],[+])` binds `X` to `+` while the call `domain_subtract_ui(`$\tau$`,[X],[+])` binds `X` to `-`. The two clauses $[\texttt{X}] \to [+]$ and $[+] \to [-]$ can be split obtaining the program

$$\{\texttt{empty} \to \texttt{empty} \qquad [+] \to [+] \qquad [+] \to [-] \qquad [-] \to [+]\} .$$

After the `domain_lub` operation we obtain the normalised program

$$\{\texttt{empty} \to \texttt{empty} \qquad [+] \to [\texttt{u}] \qquad [-] \to [+]\} .$$

□

## 4.2 The General Case of $\circ^*$.

Consider the operation $\circ^*$ in its general form. If the tails of the clauses in $T_1, \ldots, T_n$ cover $\{d_1, \ldots, d_n \mid d_i \in \mathsf{ui}(D_{\tau_i}),\ i = 1, \ldots, n\}$ with the same elements of $D^*_\tau$, for every such element $l$ we append the outputs $r_1, \ldots, r_n$ for all $l \to r_i \in T_i$ with $i = 1, \ldots, n$. We obtain just one program and we can proceed like in Subsection 4.1.

**Example 17.** Consider the computation over $S^*_\tau$ of

$$\left\{\begin{array}{l}\texttt{empty}, \texttt{empty} \to \texttt{empty}\\ \texttt{empty}, [\texttt{A}] \to \texttt{empty}\\ [\texttt{A}], \texttt{empty} \to \texttt{empty}\\ [\texttt{A}], [+] \to [+]\\ [\texttt{A}], [-] \to [-]\end{array}\right\} \circ^* \left(\left\{\begin{array}{l}\texttt{empty} \to \texttt{empty}\\ [\texttt{X}] \to [\texttt{X}]\end{array}\right\}, \left\{\begin{array}{l}\texttt{empty} \to \texttt{empty}\\ [\texttt{Y}] \to [+]\end{array}\right\}\right) .$$

We merge the two arguments into one and, like in Subsection 4.1, we compute

$$\left\{\begin{array}{l}\texttt{empty}, \texttt{empty} \to \texttt{empty}\\ \texttt{empty}, [\texttt{A}] \to \texttt{empty}\\ [\texttt{A}], \texttt{empty} \to \texttt{empty}\\ [\texttt{A}], [+] \to [+]\\ [\texttt{A}], [-] \to [-]\end{array}\right\} \circ^* \left(\left\{\begin{array}{l}\texttt{empty} \to \texttt{empty}, \texttt{empty}\\ [\texttt{Z}] \to [\texttt{Z}], [+]\end{array}\right\}\right) = \left\{\begin{array}{l}\texttt{empty} \to \texttt{empty}\\ [\texttt{Z}] \to [+]\end{array}\right\} .$$

□

Otherwise, we must reduce the arguments to the case above. This can be done by choosing all possible tuples of clauses $l_i \to r_i \in T_i$ with $i = 1, \ldots, n$ and computing the intersection of their tails $l_1, \ldots, l_n$ through `domain_intersect_ui` (Figure 5). If it succeeds with computed answer $\theta$, then the clause $l_1\theta \to (r_1, \ldots, r_n)\theta$ is generated. This computation is done by the `pave` procedure of our implementation, which can be consulted in the module `combinators` (Section 5). The final algorithm for $\circ^*$ is

```
comb_compose(Tau,[Tau1,...,Taun],Tauprime,T,[T1,...,Tn],Result):-
  pave(Tau,[T1,...,Tn],Pavement),
  findall(In1->Out2,
    (member(In1->Out1,Pavement),member(In2->Out2,T),
     entails_list([Tau1,...,Taun],Out1,In2)),Temp),
  make_disjunctive(Tau,Tauprime,Temp,Result).

entails_list([],[],[]).
entails_list([TauH|TauT],[H1|T1],[H2|T2]):-
  domain_entails(TauH,H1,H2),entails_list(TauT,T1,T2).
```

**Example 18.** Consider the computation over $S^*_\tau$ of

$$\left\{\begin{array}{l} \texttt{empty}, \texttt{empty} \to \texttt{empty} \\ \texttt{empty}, [\texttt{A}] \to \texttt{empty} \\ [\texttt{A}], \texttt{empty} \to \texttt{empty} \\ [\texttt{A}], [+] \to [+] \\ [\texttt{A}], [-] \to [-] \end{array}\right\} \circ^* \left( \left\{ \begin{array}{l} \texttt{empty} \to \texttt{empty} \\ [\texttt{X}] \to [\texttt{X}] \end{array} \right\}, \left\{ \begin{array}{l} \texttt{empty} \to \texttt{empty} \\ [+] \to [+] \\ [-] \to [\texttt{u}] \end{array} \right\} \right) .$$

We apply `pave` to the two arguments and, like in Subsection 4.1, we compute

$$\left\{\begin{array}{l} \texttt{empty}, \texttt{empty} \to \texttt{empty} \\ \texttt{empty}, [\texttt{A}] \to \texttt{empty} \\ [\texttt{A}], \texttt{empty} \to \texttt{empty} \\ [\texttt{A}], [+] \to [+] \\ [\texttt{A}], [-] \to [-] \end{array}\right\} \circ^* \left( \left\{ \begin{array}{l} \texttt{empty} \to \texttt{empty}, \texttt{empty} \\ [+] \to [+], [+] \\ [-] \to [-], [\texttt{u}] \end{array} \right\} \right) = \left\{ \begin{array}{l} \texttt{empty} \to \texttt{empty} \\ [+] \to [+] \\ [-] \to [\texttt{u}] \end{array} \right\} .$$

□

Figure 5 collects all the operations which must be implemented by an abstract domain. We have not introduced `domain_bottom` yet, which is used to start the fixpoint computation, and `domain_the_same`, which is used to stop it.

# 5 Implementation

## 5.1 Overview of the LOOP Analyser

The LOOP (Localised for Object-Oriented Programs) analyser is a generic analyser for simple object-oriented programs, i.e., Pascal procedures with objects, fields and virtual calls. LOOP is an implementation in Prolog of the watchpoint semantics of [19] extended to deal with object-oriented features through the operations of [11].

Figure 6 draws a picture of the structure of LOOP. The highest-level module `analyser` implements a fixpoint engine for a denotational semantics in terms of

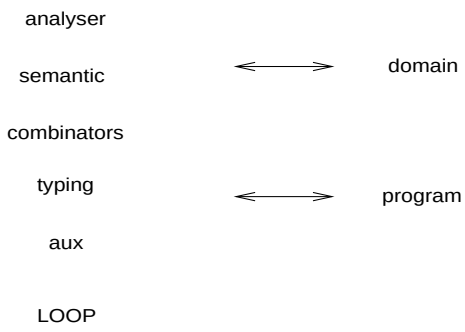


Figure 6: The structure of the LOOP analyser.

```
Procedure swap :  [ (a, int), (i, int), (j, int)] -> [ (swap, int)]

empty− > empty
[+,+,+]− > [+]   [+,+,−]− > empty   [+,−,+]− > empty   [+,−,−]− > empty
[−,+,+]− > [−]   [−,+,−]− > empty   [−,−,+]− > empty   [−,−,−]− > empty

Procedure nested :  [ (a, int), (b, int), (n, int)] -> [ (nested, int)]

   empty− > empty
   [+,+,+]− > [+]   [+,+,−]− > [+]   [+,−,+]− > [u]   [+,−,−]− > [+]
   [−,+,+]− > [u]   [−,+,−]− > [+]   [−,−,+]− > [+]   [−,−,−]− > [+]
```

Figure 7: The sign analysis of `nested` without variables.

the operations in `semantic` (for instructions, conditionals, loops). Those operations are themselves compiled in terms of calls to the module `combinators` which implements the combinators[3] of Figure 2. The module `typing` implements the typings (Definition 2). The module `aux` implements auxiliary and logging functions.

As Figure 6 shows, LOOP does not contain any abstract domain which is, instead, an *external* module implementing the operations of Figure 5. The program to be analysed is another external module which contains the abstract syntax of the code.

LOOP is able to perform both an abstract interpretation and a combination of abstract compilation [7] and partial evaluation of the program. The result is the same in both cases, but abstract compilation is often cheaper in time and space. We will always apply abstract compilation in our experiments.

More information about the LOOP analyser can be found at the following web address: `http://www.sci.univr.it/~spoto/loop/index.html`.

## 5.2 Experimental Evaluation

We have implemented six abstract domains for LOOP. The domain `signs` implements the domain of Definition 2 through ground logic programs. Since we do not have variables, the `domain_subtract_ui` operation always fails. The version of `signs` which uses variables is implemented by `signs_vars`. The domain `df` implements in a ground way the *df* domain for *class analysis* defined in [11] as a formalisation through abstract interpretation of the ideas of [5]. Class analysis of object-oriented programs overapproximates the set of classes which an expression

[3]The full-featured watchpoint semantics needs more combinators than those in Figure 2.

```
Procedure swap :  [ (a, int), (i, int), (j, int)] -> [ (swap, int)]

empty-> empty   [A,-,B]-> empty   [A,+,-]-> empty   [A,+,+]-> [A]


Procedure nested :  [ (a, int), (b, int), (n, int)] -> [ (nested, int)]

empty-> empty   [A,B,-]-> [+]   [+,+,+]-> [+]
[-,-,+]-> [+]   [-,+,+]-> [u]   [+,-,+]-> [u]
```

Figure 8: The sign analysis of `nested` with variables.

can have at run-time in a given program point. Indeed, an expression can have every type (class) compatible with its declared type, but only some of them actually arise at run-time. The *df* domain approximates program variables with a set of possible classes, but it provides a trivial approximation for the fields of the objects. Its non-ground version which uses Prolog variables is implemented by `df_vars`. The domain `ps` implements in a ground way the *ps* domain for a more precise class analysis. It has been defined in [11] through abstract interpretation from the technique of [18]. It improves the `df` domain above by providing a set of classes for fields too, and not just for program variables. Its non-ground version which uses Prolog variables is `ps_vars`. The domains `df` and `ps` have been implemented by using lists to represent the set of classes allowed for variables and fields (see [11]). The union-irreducible elements are those which allow at most one class for every variable.

We have used variance to implement `domain_the_same` (Figure 5). Variance is correct but incomplete. In our experiments, we never noticed any case of incompleteness. Note that programs larger than our benchmarks are usually made of small procedures. Thus we do not expect problems of incompleteness for larger programs, too. Anyway, after a given number of iterations, a complete test should be used.

For sign analysis, we have used three benchmarks. `fib` is the Fibonacci procedure, `nested` is a recursive procedure over an array where the recursive call is inside a `while` loop. It uses an auxiliary `swap` procedure which swaps two elements of an array[4]. `arith` implements some arbitrary precision arithmetic operations.

Our experiments have been performed by using SWI-Prolog version 3.4.5 on a Pentium III 736 Mhz machine with 256 Mbytes of RAM.

Figure 7 shows the analysis of the `nested` benchmark through the `signs` domain. Figure 8 shows the same analysis through the `signs_vars` domain. Both domains yield the same results, but variables provide a more compact representation.

For class analysis we use two benchmarks. `inv` inverts twice the class of a variable through two calls to a virtual function. That pair of calls is inside a `while` loop. `clone` implements a generic cloning of a list. This operation is performed by a virtual call to the `clone` method of the elements of the list.

Figure 9 compares the time (in seconds) and space (amount of data structures built during the analysis) costs of the sign analysis performed with `signs` and with `signs_vars`. It even shows how the analysis scales with the number of watchpoints. Figure 10 does the same for class analysis. It considers both the `df` and the `ps` domains. As you can see, in all cases the `X_vars` domain performs better than the `X` domain. It gains up to three orders of magnitude in time and space. Note how the

[4] `signs` and `signs_vars` deal with arrays as if they were a single integer.

| Benchmark | # wpnts | signs | | signs_vars | |
|---|---|---|---|---|---|
| | | time | space | time | space |
| fib | 0 | 1.25 | 315732 | 0.04 | 10262 |
| fib | 3 | 1.62 | 443200 | 0.06 | 22866 |
| fib | 6 | 1.64 | 465952 | 0.08 | 29940 |
| nested | 0 | 99.20 | 12493527 | 0.42 | 99909 |
| nested | 7 | 99.78 | 12860167 | 0.70 | 262543 |
| nested | 13 | 100.31 | 13172359 | 1.00 | 417630 |
| arith | 0 | 92.72 | 12425933 | 0.14 | 38386 |
| arith | 7 | 92.88 | 12485517 | 0.15 | 46364 |
| arith | 14 | 93.20 | 12620077 | 0.17 | 59888 |

Figure 9: Time and space required for sign analysis.

| Bmrk | # | df | | df_vars | | ps | | ps_vars | |
|---|---|---|---|---|---|---|---|---|---|
| | | time | space | time | space | time | space | time | space |
| inv | 0 | 0.40 | 130773 | 0.03 | 11089 | 0.43 | 146007 | 0.03 | 12343 |
| inv | 2 | 0.42 | 139001 | 0.03 | 13619 | 0.45 | 155199 | 0.03 | 15189 |
| inv | 4 | 0.43 | 145093 | 0.04 | 15273 | 0.47 | 162003 | 0.04 | 17047 |
| clone | 0 | 9.39 | 2388093 | 0.07 | 22109 | 258.62 | 29721631 | 0.12 | 49017 |
| clone | 6 | 9.42 | 2444777 | 0.10 | 40397 | 259.19 | 29951489 | 0.21 | 106513 |
| clone | 11 | 12.58 | 3378142 | 0.25 | 133754 | 350.11 | 40318492 | 0.66 | 398586 |

Figure 10: Time and space required for class analysis.

domains which use variables are more sensitive to the number of watchpoints.

The cost in space of the analyses, i.e., the amount of data structures built during them, is independent from the language used to implement LOOP. Thus we think that even their time cost is not related to that language.

# 6 Related Works

There are many frameworks for the abstract interpretation of imperative languages based on denotational semantics [4, 15, 16, 19, 20]. We followed here the approach of [19] which allows to specify the program points of interest for the analysis. Note, however, that the present work can be applied to the other frameworks, too.

Logic programs with variables have been used to model type dependences in the input/output behaviour of a piece of ML code [8, 14]. However, abstract denotations there are just made of one clause (a single type dependence). A similar technique has been applied to the logic programs themselves [9, 13]. Their applicability to other kinds of analyses and programming paradigms has not been studied before.

The reduction of the dimension of the abstract denotation of a piece of code and, consequently, of the number of iterations needed to reach the fixpoint, has been considered in [12, 17]. W.r.t. [12], our notion of union-irreducible elements (Definition 3) might lead to consider more points in the input of a denotation, but it allows to represent more precisely non-(component-)additive denotations.

# 7 Conclusion

We have shown how abstract denotations can be implemented by using logic programs whose variables represent dependences between the input and the output of a denotation. We have provided an experimental evaluation which shows that this technique improves both the time and the space requirements of sign and class analysis w.r.t. traditional, *ground* implementations.

We plan to apply the same technique to other analyses, like *escape analysis* [2].

We want to use constraint logic programs [10] instead of logic programs to represent abstract denotations. They allow a flexible representation of the dependences contained in a denotation. In particular, abstract domains based on sets (like the two domains for class analysis considered in this paper) should benefit from the representation of abstract denotations in terms of programs over set-constraints [6].

# References


[1] K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier and The MIT Press, 1990.

[2] B. Blanchet. Escape Analysis for Object Oriented Languages. Application to Java$^{TM}$. In *Proc. of OOPSLA'99*, volume 34(10) of *SIGPLAN Notices*, pages 20–34, Denver, CO, USA, October 1999. ACM Press.

[3] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252, 1977.

[4] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *6th ACM Symp. on Principles of Programming Languages*, pages 269–282, 1979.

[5] 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.

[6] A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and Constraint Logic Programming. *ACM Transactions and Programming Languages and Systems*, 22(5):861–931, September 2000.

[7] M. Hermenegildo, W. Warren, and S.K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(2 & 3):349–366, 1992.

[8] R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.*, 146:29–60, 1969.

[9] J. M. Howe and A. King. Implementing Groundness Analysis with Definite Boolean Functions. In G. Smolka, editor, *ESOP 2000*, volume 1782 of *Lecture Notes in Computer Science*, pages 200–214, Berlin, Germany, 2000. Springer-Verlag.

[10] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19-20:503–581, 1994.

[11] T. Jensen and F. Spoto. Class Analysis of Object-Oriented Programs through Abstract Interpretation. In F. Honsell and M. Miculan, editors, *Proceedings of the FOSSACS 2001 Conference*, volume 2030 of *Lecture Notes in Computer Science*, pages 261–275, Genova, Italy, April 2001. Springer-Verlag.

[12] J. Köller and M. Mohnen. A New Class of Function for Abstract Interpretation. In A. Cortesi and G. Filé, editors, *Proc. of the Static Analysis Symposium, SAS'99*, volume 1694 of *Lecture Notes in Computer Science*, pages 248–263, Venice, Italy, September 1999. Springer-Verlag.

[13] G. Levi and F. Spoto. An Experiment in Domain Refinement: Type Domains and Type Representations for Logic Programs. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 152–169, Pisa, Italy, September 1998. Springer-Verlag.

[14] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and Systems Sciences*, 17-3:348–375, 1978.

[15] F. Nielson. A Denotational Framework for Data Flow Analysis. *Acta Informatica*, 18:265–287, 1982.

[16] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[17] H. R. Nielson and F. Nielson. Bounded Fixed Point Iteration. In *Proc. of the 19th Symposium on Principles of Programming Languages, POPL'92*, pages 71–82, Albuquerque, New Mexico, January 1992. ACM Press.

[18] J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In *Proc. of OOPSLA'91*, volume 26(11) of *ACM SIGPLAN Notices*, pages 146–161. ACM Press, November 1991.

[19] F. Spoto. Watchpoint Semantics: A Tool for Compositional and Focussed Static Analyses. In P. Cousot, editor, *Proceedings of Static Analysis Symposium, SAS'01*, Lecture Notes in Computer Science, Paris, July 2001. Springer-Verlag.

[20] G. Winskel. *The Formal Semantics of Programming Languages.* The MIT Press, 1993.