Logic Programs as Compact Denotations*

Patricia M. Hill
School of Computing, University of Leeds, UK
hill@comp.leeds.ac.uk

Fausto Spoto
Dipartimento di Informatica, Verona, Italy
Strada Le Grazie, 15, Ca' Vignal, 37134 Verona, Italy
spoto@sci.univr.it
Ph.: +39 0458027076 Fax: +39 0458027068

Abstract

In this paper, we show how a logic program can provide a compact representation of the abstract denotations used in denotational abstract interpretation where the logic variables express regularity in the abstract behaviour of commands. We show how this representation can be used in the implementation of sign, class and escape analysis for object-oriented programs. We also show that the time and space costs are smaller than those of a ground representation. Moreover, we show that in the case of sign analysis they are comparable to those of an implementation based on binary decision diagrams.

1 Introduction

The use of denotational semantics as the basis for abstract interpretation [4, 5] is appealing since it leads to clean, 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 provided the domain of abstract properties is finite. However, it could be so large that this naive approach becomes useless in practice. Hence, compact and efficient representations based on *binary decision diagrams* (*bdd's*) [3] are traditionally used. Here, the input/output relation is represented by a propositional formula which relates the values of the output to the values of the input. This formula is then implemented by a minimal (*reduced*) bdd.

In logic, general rules are defined by non-ground formulas and, in particular, relational databases can be defined very compactly by (non-ground) logic programs. Thus, in this paper, we investigate (both theoretically and experimentally) the use of (non-ground) logic programs to represent the abstract denotation of a piece of code. In this representation, the logic variables are used to provide a compact and efficient implementation of dependencies between the abstract input and the abstract output of the code. For instance, for any variable that is not used by a given command, the abstract value just has to be 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 value of the right hand side to the abstract value of the left hand side.

We apply that technique to three different analyses.

- *Sign analysis.* This approximates integer variables by their sign using the domain defined in [22]. This is described in more detail in the next section.
- *Class analysis* for object-oriented programs. This over-approximates the set of classes which an expression can have at run-time at a given program point. Indeed, an expression can have every type (class) compatible with its declared type, although only some of them actually arise at run-time. The *class analysis* we use here uses the three domains formalised in [13] by abstract interpretation and is derived from the ideas in [1], [6] and [20], respectively *Rapid type analysis* [1] provides just one set of classes which approximates all the classes instantiated up to a given program point. The dataflow analysis of [6] is more precise since it approximates every program variable with a set of possible classes. However, it does not provide any approximation for the fields of the objects. Instead, the analysis in [20], originally based on constraints, also provides a set of classes for every field.
- *Escape analysis* [2]. This determines which dynamically created objects will not *escape* from their creating method. This information allows us to allocate these objects to the stack instead of the heap. Compared to heap allocation, stack allocation reduces the garbage collection overhead at run-time. Moreover, in the case of object-oriented languages such as Java, a knowledge of those objects that cannot escape the methods of their creating threads can be used to remove unnecessary synchronisations when the objects are accessed. We use here the escape analysis for object-oriented languages defined in [9] which collects the set of *creation points* of the objects reachable at a given program point.

We experimentally show that our technique leads to analyses that are many orders of magnitude faster and requires much less memory than analyses implemented using a ground representation. Moreover, our experiments show that our technique is comparable, *w.r.t.* efficiency, to a bdd-based approach which needs a highly-optimised library for its imple-

*This work has been funded by EPSRC grant GR/R53401.

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

(a)

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

(b)

Figure 1: Two simple programs.

mentation and requires a large amount of memory to attain that efficiency.

The paper is organised as follows. In the next section we motivate the rest of the paper using the problem of sign analysis. Section 3 introduces our notation and terminology. Section 4 presents the framework for the analysis. Section 5 defines the compact representation of the abstract denotations using (non-ground) logic programs. Section 6 shows how these compact representations may be composed. Section 7 describes the implementation and summarises our experimental results. Section 8 discusses the related work. Section 9 concludes the paper.

Acknowledgement

We would like to thank Aurélie Lagasse who implemented our domain for sign analysis using bdd's.

2 Two Motivating Examples

In this section, we explain the problem of sign analysis. This is used throughout the paper, to illustrate our approach. In Figure 1, we present two simple example programs that we use to introduce and motivate the ideas presented more formally in the rest of the paper.

2.1 Representing Abstract Denotations

Consider the problem of a sign analysis of a program. We assume there is a set of types that includes the type *int*. We want to approximate the sign (positive or negative) of just those variables whose type is *int*. We use here the domain $\{*, +, -, u\}$ for sign analysis which is similar to that defined in [22].[1] The set $\{*, +, -, u\}$ is partially ordered by $\preceq$, which is the minimal reflexive relation such that $+ \preceq u$ and $- \preceq u$.

Suppose $V = \{v_1, \ldots, v_\ell\}$ is a set of variables of interest and that τ assigns a type to each element of V Then we define

$$S_\tau = \{\mathsf{empty}\} \cup \left\{ \varsigma : \mathsf{V} \mapsto \left\{\begin{matrix} *, + \\ -, u \end{matrix}\right\} \;\middle|\; \begin{array}{l} \text{for all } v \in V: \\ \varsigma(v) = * \text{ iff} \\ \tau(v) \neq \texttt{int} \end{array} \right\}. \quad (1)$$

The idea underlying the domain S_τ 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 are approximated by a don't care mark $*$.

The set S_τ 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 $\varsigma_1(v_i) \preceq \varsigma_2(v_i)$ for all i, $1 \le i \le \ell$.

[1] In [22] all variables have type *int*.

We will write an element $\varsigma \in S_\tau \setminus \{\mathsf{empty}\}$ as a Prolog list $[\mathtt{x}_1, \ldots, \mathtt{x}_\ell]$ where v_i is the ith variable in lexicographical order in $\{v_1, \ldots, v_\ell\}$ and $\varsigma(v_i) = x_i$ for every $i = 1, \ldots, \ell$.

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 `out` are addressable at that program point[2] and they all have type *int*, that denotation is a function $f : S_\tau \mapsto S_\tau$ with $\tau = [\mathtt{a} \mapsto int, \mathtt{b} \mapsto int, \mathtt{c} \mapsto int, \mathtt{out} \mapsto int\}$ which can be represented as

$$\begin{array}{ll}
\mathtt{empty} \to \mathtt{empty} & \\
[+,+,+,+] \to [+,+,+,+] & [+,+,+,-] \to [+,+,+,-] \\
[+,+,-,+] \to [+,+,-,+] & [+,+,-,-] \to [+,+,-,-] \\
[+,-,+,+] \to [-,-,+,+] & [+,-,+,-] \to [-,-,+,-] \\
[+,-,-,+] \to [-,-,-,+] & [+,-,-,-] \to [-,-,-,-] \\
[-,+,+,+] \to [+,+,+,+] & [-,+,+,-] \to [+,+,+,-] \\
[-,+,-,+] \to [+,+,-,+] & [-,+,-,-] \to [+,+,-,-] \\
[-,-,+,+] \to [-,-,+,+] & [-,-,+,-] \to [-,-,+,-] \\
[-,-,-,+] \to [-,-,-,+] & [-,-,-,-] \to [-,-,-,-]
\end{array} \quad (2)$$

Note that f is not explicitly specified for the elements of S_τ binding some variables to u. This is because these values can be recovered from those provided above [14, 19, 22] (see Definition 2 in Section 4). For instance,

$$\begin{aligned}
f([+,\mathtt{u},+,+]) &= f([+,+,+,+]) \sqcup f([+,-,+,+]) \\
&= [+,+,+,+] \sqcup [-,-,+,+] \\
&= [\mathtt{u},\mathtt{u},+,+]\ .
\end{aligned}$$

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

$$\begin{array}{ll}
\mathtt{io}(\mathtt{empty},\mathtt{empty}). & \mathtt{io}([+,+,+,+],[+,+,+,+]). \\
\mathtt{io}([+,+,+,-],[+,+,+,-]). & \ldots
\end{array}$$

This simple representation is not practical as its size grows exponentially with the number of program variables. However, there is some regularity in (2); namely, f does not change the values of `b`, `c` and `out`; and the value of `a` always becomes the value of `b`. We can therefore represent f *using logical variables* corresponding to the original program variables as

$$\mathtt{empty} \to \mathtt{empty} \qquad [\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{Out}] \to [\mathtt{B},\mathtt{B},\mathtt{C},\mathtt{Out}]. \quad (3)$$

This is smaller than (2) and the size of the representation grows linearly with the number of program variables. This notation can be naturally seen as isomorphic to the logic program:

$$\mathtt{io}(\mathtt{empty},\mathtt{empty}). \qquad \mathtt{io}([\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{Out}],[\mathtt{B},\mathtt{B},\mathtt{C},\mathtt{Out}]). \quad (4)$$

[2] The special variable `out` holds the return value of the function.

```
swap(a:array of int,i:int,j:int):array of int
{
  let temp:int in
  {
    temp:=a[i];
    a[i]=a[j];
    a[j]:=temp;
    out:=a;
  }
}

nested(a:array of int,a:array of int,n:int):int
{
  out:=0;

  let i:int in
  {
    n:=n-1;

    while (i <= n) do
    {
      let j:int in
        while (j <= n) do
        {
          out:=out+a[i]*b[j];
          a:=swap(a,i,n);
          b:=swap(b,j,n);
          out:=out+nested(a,b,n);
          a:=swap(a,i,n);
          b:=swap(b,j,n);
        }
    }
  }
}
```

Figure 2: A sample program to analyse.

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

$$
\begin{array}{ll}
\texttt{empty}- > \texttt{empty} & \\
[+,+,+]- > [+] & [+,+,-]- > \texttt{empty} \\
[+,-,+]- > \texttt{empty} & [+,-,-]- > \texttt{empty} \\
[-,+,+]- > [-] & [-,+,-]- > \texttt{empty} \\
[-,-,+]- > \texttt{empty} & [-,-,-]- > \texttt{empty}
\end{array}
$$

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

$$
\begin{array}{ll}
\texttt{empty}- > \texttt{empty} & \\
[+,+,+]- > [+] & [+,+,-]- > [+] \\
[+,-,+]- > [\texttt{u}] & [+,-,-]- > [+] \\
[-,+,+]- > [\texttt{u}] & [-,+,-]- > [+] \\
[-,-,+]- > [+] & [-,-,-]- > [+]
\end{array}
$$

Figure 3: The ground analysis of the program in Figure 2.

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

$$
\begin{array}{ll}
\texttt{empty}- > \texttt{empty} & [\texttt{A},-,\texttt{B}]- > \texttt{empty} \\
[\texttt{A},+,-]- > \texttt{empty} & [\texttt{A},+,+]- > [\texttt{A}]
\end{array}
$$

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

$$
\begin{array}{ll}
\texttt{empty}- > \texttt{empty} & [\texttt{A},\texttt{B},-]- > [+] \\
[+,+,+]- > [+] & [-,-,+]- > [+] \\
[-,+,+]- > [\texttt{u}] & [+,-,+]- > [\texttt{u}]
\end{array}
$$

Figure 4: The non-ground analysis of the program in Fig. 2.

From now on, the (more evocative) arrow notation as in (3) denotes the corresponding logic program as in (4).

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

$$
\begin{array}{ll}
\texttt{empty} \rightarrow \texttt{empty} & \\
[\texttt{A},-,\texttt{C},\texttt{Out}] \rightarrow [-,-,\texttt{C},\texttt{Out}] & [\texttt{A},+,\texttt{C},\texttt{Out}] \rightarrow [\texttt{u},+,\texttt{C},\texttt{Out}]
\end{array}
\tag{5}
$$

Note that (5) is slightly more complex than (3), but is still much more compact than an exhaustive representation. Moreover, its size still grows linearly with the number of program variables.

2.2 Sign Analysis of Imperative Programs

As we said in Subsection 2.1, logic programs can be used to represent compactly the abstract denotation of a piece of code. This feature can be used by a static analyser based on denotational semantics. Consider for instance the program shown in Figure 2. Figure 3 shows its sign analysis computed by using the domain (1) implemented through ground dependencies (arrays of integers are treated as if they were a single integer). Figure 4 shows the same analysis computed through the same domain implemented, however, through a compact representation which uses logic variables. The result of both analyses are the same, but variables provide a more compact representation. Note, moreover, that the result in Figure 3 has been obtained in 118.732 seconds and required more than $13M$ Prolog atoms in the stack of on our machine, while the result in Figure 4 has been obtained in only 0.564 seconds and required just 56390 atoms (Figure 9). This definitely justifies our investigation.

3 Preliminaries

The *(co-)domain* of a function f is $\mathsf{dom}(f)$ ($\mathsf{cd}(f)$). A total (partial) function is denoted by $\mapsto$ ($\rightarrow$). We denote by $[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n]$ a function f whose domain is $\{v_1, \ldots, v_n\}$ and such that $f(v_i) = t_i$ for $i = 1, \ldots, n$. An *update* of f is denoted by $f[w_1 \mapsto d_1, \ldots, w_m \mapsto d_m]$, where the domain of f may be enlarged. By $f|_s$ ($f|_{-s}$) we denote the *restriction* of f to $s \subseteq \mathsf{dom}(f)$ (to $\mathsf{dom}(f) \setminus s$).

A pair $\langle C, \leq \rangle$ is a *poset* if $\leq$ is reflexive, transitive and antisymmetric on C. A poset is a *complete lattice* when *least upper bounds* (lub) and *greatest lower bounds* (glb) always exist.

In abstract interpretation [4], a *Galois connection* between two posets $\langle C, \leq \rangle$ and $\langle A, \preceq \rangle$ (the *concrete* and the *abstract* domain) is a pair of monotonic maps $\alpha : C \mapsto A$ and $\gamma : A \mapsto C$ such that $\gamma\alpha$ is extensive and $\alpha\gamma$ is reductive.

Given a set of variables *Vars* and a set of function symbols Σ with associated arity, the set of *terms* is the smallest set containing *Vars* and the application of every $f \in \Sigma$ to as many terms as the arity of f requires. A *substitution* is a map from a finite set of variables to terms. If t is a term and σ a substitution, we denote by $t\sigma$ the *application* of σ

to t. The partial ordering on terms is defined as $t_1 \leq t_2$ if there exists a substitution σ such that $t_1 = t_2\sigma$.

4 The Framework of Analysis

Our framework is not bound to any particular programming paradigm, although all our examples are for an imperative object-oriented language. In general, however, all we require is that the (concrete or abstract) denotation (*semantics*) of a program P is defined as the fixpoint of a transformer of denotations T_P. This T_P is itself defined as the *composition* of *basic* denotations. For instance, in the case of logic programs, the basic denotations are the denotations of the (concrete or abstract) built-in's and substitutions. Composition is the (concrete or abstract) composition of substitutions. In the case of imperative programs, the basic denotations are the denotations of every (concrete or abstract) single command or bytecode [17]. Composition is the (concrete or abstract) composition of bytecodes.

In order to sinplify the presentation, we assume there is a set of *program variables* that are lexicographically ordered and that there is a finite set of *types*. For simplicity we assume that *int* is the only predefined type, and all other types are classes defined by the programmer

Definition 1. A *type environment* τ is a map assigning a type to the variables in its domain. From now on, τ will silently stand for a type environment.

We now formalise our framework without committing to any particular set of basic denotations. Since our technique is used to implement static analyses, we consider abstract denotations only. Hence, we need to state which is the abstract domain, and which of its elements must be considered as inputs of a denotation. Namely, an element of an abstract domain is (γ-)*union-irreducible* if its concretisation is not the union of the concretisations of the elements which precede it in the partial ordering. This is a stronger notion than that of *join-irreducible* element [14]. While [14] aims at providing an upper bound to the number of iterations of a fixpoint computation, our union-irreducible elements are used as *meaningful* inputs for a denotation. Note that we do not require denotations to be (component-wise) additive [14].

Definition 2. Let $\langle C_\tau, \subseteq\rangle$ be a *concrete domain* of *states* and $\langle D_\tau, \sqsubseteq\rangle$ a finite complete lattice, such that a Galois connection exists between $\langle C_\tau, \subseteq\rangle$ and $\langle D_\tau, \sqsubseteq\rangle$, with *abstraction* and *concretisation* maps $\alpha_\tau : \wp(C_\tau) \mapsto D_\tau$ and $\gamma_\tau : D_\tau \mapsto \wp(C_\tau)$. The domain $\langle D_\tau, \sqsubseteq\rangle$ is called an *abstract domain*. We say that $d \in D_\tau$ is (γ-)*union-reducible* if

$$\gamma_\tau(d) = \cup\{\, \gamma_\tau(d') \mid d' \sqsubset d \,\} \ .$$

Otherwise, d is *union-irreducible*. The set of the union-(ir)reducible elements of D_τ is denoted by $\mathsf{ur}(D_\tau)$ ($\mathsf{ui}(D_\tau)$).

Example 3. Consider the domain S_τ of (1). Then S_τ is an abstract domain for sign analysis. we have

$$\mathsf{ui}(S_\tau) = \{\mathsf{empty}\} \cup \left\{ \varsigma \in S_\tau \setminus \{\mathsf{empty}\} \;\middle|\; \begin{array}{l} \varsigma(v) \neq u, \\ \text{for every} \\ v \in \mathsf{dom}(\tau) \end{array} \right\} .$$

Letting $\mathsf{dom}(\tau) = \{v_1, \ldots, v_\ell\}$, it can be shown that (for the concretisation map γ^S_τ, see [22])

$$\gamma^S_\tau(\varsigma) = \bigcup \left\{ \gamma^S_\tau([\mathsf{s}_1, \ldots, \mathsf{s}_\ell]) \;\middle|\; \begin{array}{l} s_i \in \{+, -\} \\ s_i \preceq \varsigma(v_i) \\ 1 \leq i \leq \ell \end{array} \right\} \cup \gamma^S_\tau(\mathsf{empty}) \ .$$

If $\varsigma \in \mathsf{ur}(S_\tau)$, then, there exists some $1 \leq j \leq \ell$ such that $\varsigma(v_j) = u$. Thus, in this case, we have

$$\gamma^S_\tau(\varsigma) = \cup\{\gamma^S_\tau(\varsigma') \mid \varsigma' \sqsubset \varsigma\} \ .$$

The next result shows that the concretisation map is uniquely identified by the union-irreducible elements of the domain.

Proposition 4. *Let* $d \in \mathsf{ur}(D_\tau)$. *We have*

$$\gamma_\tau(d) = \cup\{\, \gamma_\tau(d') \mid d' \in \mathsf{ui}(D_\tau) \text{ and } d' \sqsubset d \,\} \ .$$

Proof. By an iterated application of Definition 2, because D_τ is finite, and hence it does not contain infinite descending chains. □

We say that a function is ui-*induced* if it uniquely identified by its restriction to the union-irreducible elements.

Definition 5. Let $n \geq 1$ and $a \in D_{\tau_1} \times \cdots \times D_{\tau_n} \mapsto D_\tau$. We say that a is ui-*induced* if, whenever $d_i \in D_{\tau_i}$ for all $1 \leq i \leq n$ and $d_j \in \mathsf{ur}(D_{\tau_j})$ for a given $1 \leq j \leq n$, we have

$$a(d_1, \ldots, d_n) = \bigsqcup \left\{ a(d_1, \ldots, d', \ldots, d_n) \;\middle|\; \begin{array}{l} d' \in \mathsf{ui}(D_{\tau_j}) \\ d' \sqsubset d_j \end{array} \right\} .$$

Example 6. Let $\tau = [v \mapsto int, w \mapsto int]$. The map $a : D_\tau \mapsto D_\tau$ defined as

$$\begin{array}{lll} a(\mathsf{empty}) = [+, +] & & \\ a([+, +]) = [+, u] & a([+, -]) = [+, +] & a([+, u]) = [-, u] \\ a([-, +]) = [-, +] & a([-, -]) = \mathsf{empty} & a([-, u]) = [-, +] \\ a([u, +]) = [+, u] & a([u, -]) = [+, -] & a([u, u]) = [u, u] \end{array} \quad (6)$$

is not ui-induced, since

$$\{s \in S_\tau \mid s \sqsubset [u, +]\} = \{\mathsf{empty}, [+, +], [-, +]\}$$

and $a(\mathsf{empty}) \sqcup a([+, +]) \sqcup a([-, +]) = [u, u]$, but $a([u, +]) \neq [u, u]$. To make a ui-induced, we must replace (6) by

$$a([u, +]) = [u, u] \quad a([u, -]) = [+, +] \quad a([u, u]) = [u, u] \ . \quad (7)$$

Concrete denotations are monotonic maps over C. They are approximated by *abstract* denotations.

Definition 7. Let $n \geq 1$. The set $\mathbb{D}^{\tau_1, \ldots, \tau_n}_\tau$ of the *abstract* $(\tau_1, \ldots, \tau_n, \tau)$-*denotations* consists of the monotonic and ui-induced maps in $D_{\tau_1} \times \cdots \times D_{\tau_n} \mapsto D_\tau$. It is a complete lattice *w.r.t.* the pointwise extension of $\sqsubseteq$. The word *abstract* will usually be omitted.

$$
\begin{aligned}
&[op] : \mathbb{D}_{\tau}^{\tau_1,\cdots,\tau_n}, \text{ with } op : (C_{\tau_1} \times \cdots \times C_{\tau_n}) \mapsto C_\tau\\
&\circ : \left(\mathbb{D}_{\tau'}^{\tau_1,\ldots,\tau_n} \times \mathbb{D}_{\tau_1}^{\tau} \times \cdots \times \mathbb{D}_{\tau_n}^{\tau}\right) \mapsto \mathbb{D}_{\tau'}^{\tau} \quad \text{(infix)}\\
&[op](d_1,\ldots,d_n) = \alpha_\tau\Big(op\big(\gamma_{\tau_1}(d_1),\ldots,\gamma_{\tau_n}(d_n)\big)\Big)\\
&\big(T \circ (T_1,\ldots,T_n)\big)(d) =\\
&\begin{cases} T\big(T_1(d),\ldots,T_n(d)\big) & d \in \mathsf{ui}(D_\tau)\\ \bigsqcup\left\{ T\big(T_1(d'),\ldots,T_n(d')\big) \;\middle|\; \begin{array}{l} d' \in \mathsf{ui}(D_\tau)\\ d' \sqsubset d \end{array}\right\} & \text{otherwise.}\end{cases}
\end{aligned}
$$

Figure 5: Signature and implementation of the operations on denotations.

Example 8. Neither map a of Example 6 nor the map obtained from a by replacing (7) by (6) are a (τ,τ)-denotation. This is because $\mathsf{empty} \sqsubseteq [-,+]$ but $a(\mathsf{empty}) \not\sqsubseteq \mathsf{a}([-,+])$. Hence a is not monotonic. Instead, the map $b : D_\tau \mapsto D_\tau$ defined as

$$
\begin{array}{lll}
b(\mathsf{empty}) = \mathsf{empty} & & \\
b([+,+]) = [+,u] & b([+,-]) = [+,+] & b([+,u]) = [-,u]\\
b([-,+]) = [-,+] & b([-,-]) = \mathsf{empty} & b([-,u]) = [-,+]\\
b([u,+]) = [u,u] & b([u,-]) = [+,+] & b([u,u]) = [-,u] \quad (8)
\end{array}
$$

is both monotonic and ui-induced. It follows that it is a (τ,τ)-denotation. Note that since line (8) is induced by the previous ones, it is *superfluous* (we do not need to represent it in a computer).

The concrete transformer T_P on denotations is usually defined in terms of *basic* denotations, which are the denotation of a *basic* command (substitution, bytecode, ...), and whose exact definition is not relevant to this paper, and of the functional composition of concrete denotations, which models the conposition of substitutions or bytecodes. The abstract T_P operator uses the abstract counterparts of these operations.

Definition 9. Figure 5 defines the operations on abstract denotations.

The operation $[op]$ of Figure 5 uses the best approximation $\alpha op \gamma$ of op. For every language we want to analyse, and, for every concrete operation of the language, we have to provide such an approximation. I

Example 10. Consider the abstract domain S_τ of (1) where $\tau = [v \mapsto int, w \mapsto int]$. The denotation of the operation ($\mathtt{w} := \mathtt{v}$) on the concrete domain is approximated by the following denotation on S_τ

$$
\begin{array}{lll}
a(\mathsf{empty}) = \mathsf{empty} & & \\
a([+,+]) = [+,+] & a([+,-]) = [+,+] & a([+,u]) = [+,+]\\
a([-,+]) = [-,-] & a([-,-]) = [-,-] & a([-,u]) = [-,-]\\
a([u,+]) = [u,u] & a([u,-]) = [u,u] & a([u,u]) = [u,u] \ .
\end{array}
$$

The concrete operation $\mathtt{if\ v} \geq \mathtt{0\ then}$ is approximated by the following denotation on S_τ

$$
\begin{array}{lll}
b(\mathsf{empty}) = \mathsf{empty} & & \\
b([+,+]) = [+,+] & b([+,-]) = [+,-] & b([+,u]) = [+,u]\\
b([-,+]) = \mathsf{empty} & b([-,-]) = \mathsf{empty} & b([-,u]) = \mathsf{empty}\\
b([u,+]) = [+,+] & b([u,-]) = [+,-] & b([u,u]) = [+,u] \ .
\end{array}
$$

Observe that a and b are monotonic and ui-induced maps so that it follows that $a, b \in \mathbb{D}_\tau^\tau$.

It is outside of the scope of this paper to show that the *best* approximation of the composition of concrete denotations (which is functional) is the (non-functional) operation $\circ$ in Figure 5. We provide, instead, an example which shows that functional composition sometimes looses precision.

Example 11. Consider τ and a, b as in Example 10. Their functional composition ba is

$$
\begin{array}{lll}
ba(\mathsf{empty}) = \mathsf{empty} & & \\
ba([+,+]) = [+,+] & ba([+,-]) = [+,+] & ba([+,u]) = [+,+]\\
ba([-,+]) = \mathsf{empty} & ba([-,-]) = \mathsf{empty} & ba([-,u]) = \mathsf{empty}\\
ba([u,+]) = [+,u] & ba([u,-]) = [+,u] & ba([u,u]) = [+,u] \ .
\end{array} \tag{9}
$$

Note that $ba \notin \mathbb{D}_\tau^\tau$ since it is not ui-induced, indicating that some information has been lost. To see this, consider the input $[u,+]$. Nothing is known about the sign of the variable v, hence it may be positive or negative. If v is positive, $ba([+,+]) = [+,+]$. If v is negative, $ba([-,+]) = \mathsf{empty}$. Combining these we obtain $ba([+,+]) \sqcup ba([-,+]) = [+,+] \sqcup \mathsf{empty} = [+,+]$. However, using functional composition we have obtained $ba([u,+]) = [+,u]$ which although correct is less precise than $[+,+]$.

If we compute the composition as in Figure 5, line (9) is substituted by a more precise approximation

$$
ba([u,+]) = [+,+] \quad ba([u,-]) = [+,+] \quad ba([u,u]) = [+,+] \ .
$$

This example shows that functional composition is not able to prove that w is positive in the `then` branch of the command `w := v; if v ≥ 0 then` (Example 10), while the composition given in Figure 5 is.

Proposition 12. *The operations in Fig. 5 are well-defined.*

Proof. We have to prove that the conditions of Definition 7 hold. Monotonicity follows because both $[op]$ and $\circ$ are implemented as composition of monotonic functions. Note that op in the definition of $[op]$ is the powerset extension of $op : (C_{\tau_1} \times \cdots \times C_{\tau_n}) \mapsto C_\tau$ and is, hence, monotonic. The result of the operation $\circ$ is ui-induced by definition. To prove that $[op]$ is ui-induced, let $d_i \in D_{\tau_i}$ for $1 \leq i \leq n$ and let $d_j \in \mathsf{ur}(D_{\tau_j})$ for a given $1 \leq j \leq n$. Let $J = \{d' \in \mathsf{ui}(D_{\tau_j}) \mid d' \sqsubset d_j\}$. By Proposition 4, since op is a pointwise extension and α_τ is additive, we have

$$
\begin{aligned}
&[op](d_1,\ldots,d_j,\ldots,d_n)\\
&= \alpha_\tau(op(\gamma_{\tau_1}(d_1),\ldots,\gamma_{\tau_j}(d_j),\ldots,\gamma_{\tau_n}(d_n)))\\
&= \alpha_\tau(op(\gamma_{\tau_1}(d_1),\ldots,\cup\{\gamma_{\tau_j}(d') \mid d' \in J\},\ldots\gamma_{\tau_n}(d_n)))\\
&= \alpha_\tau(\cup\{op(\gamma_{\tau_1}(d_1),\ldots,\gamma_{\tau_j}(d'),\ldots\gamma_{\tau_n}(d_n)) \mid d' \in J\})\\
&= \sqcup\{\alpha_\tau(op(\gamma_{\tau_1}(d_1),\ldots,\gamma_{\tau_j}(d'),\ldots,\gamma_{\tau_n}(d_n))) \mid d' \in J\}\\
&= \sqcup\{[op](d_1,\ldots,d',\ldots,d_n)) \mid d' \in J\} \ .
\end{aligned}
$$

□

5 Logic Programs as Compact Denotations

In Subsection 2.1 we have shown that logic programs can be used to represent abstract denotations (Definition 7) in a compact way. We detail here this representation.

Definition 13. We assume there is a set of terms D defined by a set of function symbols and variables W that can be used to represent the abstract domain elements. In particular, we assume that every $d \in D_\tau$ can be represented by a ground term in D. We let d stands for its representation in D and vice versa.

The compact representation of the abstract denotations is represented by, possibly non-ground, terms in D. Let $D^*_\tau \subseteq D$ such that $\{\, d^* \in D^*_\tau \mid vars(d^*) = \varnothing \,\} = D_\tau$.

We generalise the concept of union-irreducible and union-reducible given in Definition 2 to compact representations.

Definition 14. A (possibly non-ground) term $d^* \in D^*_\tau$ is *union-irreducible* if it has an instance $d \in \mathsf{ui}(D_\tau)$. Otherwise, it is *union-reducible*. The set of the union-irreducible elements in D^*_τ is denoted by $\mathsf{ui}(D^*_\tau)$.

Example 15. Consider the domain S_τ in Equation (1). We have already introduced the ground terms $\mathtt{empty}$ and $[\mathtt{x}_1, \ldots, \mathtt{x}_\ell]$ $(x_i \in \{*, +, -, \mathtt{u}\})$ to represent its elements. We assume $\mathsf{dom}(\tau) = \{v_1, \ldots, v_\ell\}$ and

$$S^*_\tau = \{\mathtt{empty}\} \cup \left\{ [\mathtt{x}_1, \ldots, \mathtt{x}_\ell] \;\middle|\; \begin{array}{l} \text{for } i = 1, \ldots, \ell, \\ \mathtt{x_i} \in \{*, +, -, \mathtt{u}\} \cup \mathtt{W} \\ \mathtt{x_i} = * \text{ iff } \tau(\mathtt{v_i}) \neq \mathit{int} \end{array} \right\} .$$

If $\tau = [\mathtt{a} \mapsto \mathit{int}, \mathtt{b} \mapsto \mathit{int}, \mathtt{c} \mapsto \mathit{int}, \mathtt{d} \mapsto \mathit{int}]$, the set S^*_τ contains the non-ground union-irreducible terms $[+, -, \mathtt{X}, \mathtt{Y}]$ and $[+, -, \mathtt{X}, \mathtt{X}]$, as well as the non-ground union-reducible terms $[+, \mathtt{u}, \mathtt{X}, \mathtt{Y}]$ and $[+, \mathtt{u}, \mathtt{X}, \mathtt{X}]$.

A *non-ground representation* is given by a set D^*_τ together with a set of substitutions Λ.

Definition 16. Let Λ be a set of substitutions closed by composition and let Λ^g denote its grounding subset. Let $\xi^\Lambda(X) = \{X\sigma \mid \sigma \in \Lambda^g\}$ for every syntactic object X (terms, clauses, programs). Let *Typing* denote a set of type environments. Then we say that the pair $\langle \{D^*_\tau \mid \tau \in \mathit{Typing}\}, \Lambda \rangle$ is a *non-ground representation* and Λ is *legal* if the pair satisfies the following conditions:

1. if $d^* \in D^*_\tau$ and $\sigma \in \Lambda$ then $d^*\sigma \in D^*_\tau$ (D^*_τ is closed *w.r.t.* application of substitutions in Λ),
2. if $d^* \in \mathsf{ui}(D^*_\tau)$ then $\xi^\Lambda(d^*) = \{d \in \mathsf{ui}(D_\tau) \mid d \leq d^*\}$ (union-irreducibility cannot be lost).

Example 17. A set of legal substitutions for a term in S^*_τ (Example 15) is

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

i.e., we do not allow variables to be bound to $\mathtt{u}$. This is because a set of legal substitutions must not lose union-irreducibility (point 2 of Definition 16).

In the following, we assume that a non-ground representation $\langle \{D^*_\tau \mid \tau \in \mathit{Typing}\}, \Lambda \rangle$ has been given and write ξ for ξ^Λ.

Definition 18. Let $n \geq 1$. A $(\tau_1, \ldots, \tau_n, \tau)$-*compact clause* is an arrow $l_1, \ldots, l_n \to r$, with $l_i \in \mathsf{ui}(D^*_{\tau_i})$ for $i = 1, \ldots, n$, $r \in D^*_\tau$ and $vars(r) \subseteq \cup\{vars(l_i) \mid i = 1, \ldots, n\}$. We say that $l_1, \ldots, l_n$ are its *inputs* and that r is its *output*. The *meaning* of a set t of $(\tau_1, \ldots, \tau_n, \tau)$-compact clauses is the unique ui-induced map $\overline{t} : D_{\tau_1} \times \cdots \times D_{\tau_n} \mapsto D_\tau$ such that, for every $L \in \mathsf{ui}(D_{\tau_1}) \times \cdots \times \mathsf{ui}(D_{\tau_n})$,

$$\overline{t}(L) = \sqcup\{r\theta \mid L' \to r \in t,\ \theta \in \Lambda^g \text{ and } L'\theta = L\} .$$

Example 19. If $\tau = [\mathtt{a} \mapsto \mathit{int}, \mathtt{b} \mapsto \mathit{int}, \mathtt{c} \mapsto \mathit{int}, \mathtt{out} \mapsto \mathit{int}]$ then $c_1 = (\mathtt{empty} \to [+, +, \mathtt{u}, -])$, $c_2 = ([+, -, \mathtt{X}, \mathtt{Y}] \to [\mathtt{u}, \mathtt{Y}, +, +])$ and $c_3 = ([+, \mathtt{X}, +, +] \to [+, -, \mathtt{X}, \mathtt{u}])$ are (τ, τ)-compact clauses. The meaning of $\{c_1, c_2, c_3\}$ is the unique ui-induced map $D_\tau \mapsto D_\tau$ such that

$$\begin{array}{ll}
\mathtt{empty} \to [+, +, \mathtt{u}, -] & \\
[+, +, +, +] \to [+, -, +, \mathtt{u}] & [+, +, +, -] \to \mathtt{empty} \\
[+, +, -, +] \to \mathtt{empty} & [+, +, -, -] \to \mathtt{empty} \\
[+, -, +, +] \to [\mathtt{u}, \mathtt{u}, \mathtt{u}, \mathtt{u}] & [+, -, +, -] \to [\mathtt{u}, -, +, +] \\
[+, -, -, +] \to [-, +, +, +] & [+, -, -, -] \to [\mathtt{u}, -, +, +] \\
[-, +, +, +] \to \mathtt{empty} & [-, +, +, -] \to \mathtt{empty} \\
[-, +, -, +] \to \mathtt{empty} & [-, +, -, -] \to \mathtt{empty} \\
[-, -, +, +] \to \mathtt{empty} & [-, -, +, -] \to \mathtt{empty} \\
[-, -, -, +] \to \mathtt{empty} & [-, -, -, -] \to \mathtt{empty}
\end{array}$$

Note that both c_2 and c_3 contribute to determine the output for the input $[+, -, +, +]$. Moreover, note that $\overline{\{c_1, c_2, c_3\}}$ is not monotonic. Hence, it is *not* a (τ, τ)-denotation (Definition 7).

As Example 19 shows, the meaning of a set of compact clauses is not necessarily a denotation (Definition 7). Thus Definition 20 defines a sufficient condition on sets of compact clauses which guarantees that their meaning *is* a denotation (Proposition 22). Moreover, it requires that sets of compact clauses have exhaustive and non-overlapping inputs. This defines a *normal form* which keeps the sets of compact clauses small.

Definition 20. Let $n, m \geq 1$. The set $\mathbb{CD}^{\tau_1, \ldots, \tau_n}_\tau$ of the $(\tau_1, \ldots, \tau_n, \tau)$-*compact denotations* is formed by those sets $\{L_1 \to r_1, \ldots, L_m \to r_m\}$ of $(\tau_1, \ldots, \tau_n, \tau)$-compact clauses such that

i) $\cup\{\xi(L_i) \mid 1 \leq i \leq m\} = \mathsf{ui}(D_{\tau_1}) \times \cdots \times \mathsf{ui}(D_{\tau_n})$,

ii) $\xi(L_j) \cap \xi(L_k) = \varnothing$ for $1 \leq j, k \leq m$, $j \neq k$,

iii) $\overline{\{L_1 \to r_1, \ldots, L_m \to r_m\}}$ is monotonic.

Example 21. The set of compact clauses $\{c_1, c_2, c_3\}$ of Example 19 does not satisfy any of the conditions of Definition 20. Instead, the set of compact clauses (5) at page 3 satisfies those conditions. Hence, it is a (τ, τ)-compact denotation.

Proposition 22. *Let $n \geq 1$ and $t \in \mathbb{CD}^{\tau_1, \ldots, \tau_n}_\tau$. We have $\overline{t} \in \mathbb{D}^{\tau_1, \ldots, \tau_n}_\tau$.*

Proof. By Definition 18, we know that $\overline{t}$ is ui-induced. By point iii of Definition 20 we know that it is monotonic. ☐

The next result provides an explicit definition of the meaning of a compact denotation, also for union-reducible inputs (compare with Definition 18).

Proposition 23. *Let $n \geq 1$, $t \in \mathbb{CD}^{\tau_1, \ldots, \tau_n}_\tau$ and $L \in D_{\tau_1} \times \cdots \times D_{\tau_n}$. We have*

$$\overline{t}(L) = \sqcup\{r\theta \mid L' \to r \in t,\ \theta \in \Lambda^g,\ L'\theta \sqsubseteq L\} .$$

Proof. If $L \in \mathsf{ui}(D_{\tau_1}) \times \cdots \times \mathsf{ui}(D_{\tau_n})$, by point i of Definition 20, there exists an $L' \to r \in t$ such that $L'\theta = L$ for a suitable $\theta \in \Lambda^g$. By Definition 18 and point iii of Definition 20, we conclude that

$$\begin{aligned}
\overline{t}(L) &= \sqcup\{\overline{t}(L'') \mid L'' \sqsubseteq L\} \\
&= \sqcup\{r\theta \mid L' \to r \in t,\ \theta \in \Lambda^g,\ L'' \sqsubseteq L \text{ and } L'\theta = L''\} \\
&= \sqcup\{r\theta \mid L' \to r \in t,\ \theta \in \Lambda^g \text{ and } L'\theta \sqsubseteq L\} .
\end{aligned}$$

In $L \notin \mathsf{ui}(D_{\tau_1}) \times \cdots \times \mathsf{ui}(D_{\tau_n})$, by Proposition 22, Definition 5 and the case shown above, we have

$$\begin{aligned}\overline{t}(L) &= \sqcup\{\overline{t}(L') \mid L' \in \mathsf{ui}(D_{\tau_1}) \times \cdots \times \mathsf{ui}(D_{\tau_n}) \text{ and } L' \sqsubset L\}\\ &= \sqcup\{r\theta \mid L'' \to r \in t,\ \theta \in \Lambda^g,\ L''\theta \sqsubseteq L' \text{ and } L' \sqsubset L\}\\ &= \sqcup\{r\theta \mid L'' \to r \in t,\ \theta \in \Lambda^g \text{ and } L''\theta \sqsubseteq L\}\\ (*) &= \sqcup\{r\theta \mid L'' \to r \in t,\ \theta \in \Lambda^g \text{ and } L''\theta \sqsubseteq L\}\ ,\end{aligned}$$

where point $*$ follows since $L \notin \mathsf{ui}(D_{\tau_1}) \times \cdots \times \mathsf{ui}(D_{\tau_n})$. □

Now that we have a notion of compact denotations $\mathbb{CD}$, we want to define operations which mimics over $\mathbb{CD}$ what the operations in Figure 5 do over $\mathbb{D}$. Section 6 shows the case of $\circ$. For what concerns $[op]$, instead, we assume that an explicit compact denotation for every op is given. Note that such compact denotations always exist, since a ground representation is monotonic and ui-induced (Proposition 12) and hence is a degenerate case of compact denotation. However, we should try to use logic variables as much as possible, in order to better exploit the capabilities of our non-ground representation.

Example 24. Consider again the operations given in Example 10. A non-degenerate compact denotation for $[op]$, where $op = (\mathtt{w} := \mathtt{v})$ is Example 10, is

$$\mathtt{empty} \to \mathtt{empty} \qquad [\mathtt{V},\mathtt{W}] \to [\mathtt{V},\mathtt{V}]\ .$$

A non-degenerate compact denotation for $[op]$, where $op =$ `if v` $\geq$ `0 then` is

$$\mathtt{empty} \to \mathtt{empty} \qquad [+,\mathtt{W}] \to [+,\mathtt{W}] \qquad [-,\mathtt{W}] \to \mathtt{empty}\ .$$

6 Composition of Compact Denotations

We define in this section an operation $\circ^*$ which composes compact denotations, and we show that it exactly mimics what $\circ$ does over denotations (Figure 5). We first consider the simpler case when in Figure 5 we have $n = 1$. We will then generalise the solution (Subsection 6.2)

6.1 The Simpler Case of $\circ^*$

We start by defining a *pre-composition* $\bullet$ which, although it correctly mimics $\circ$ (Proposition 35), is not closed on $\mathbb{CD}$ (Proposition 30).

Definition 25. Let $n \geq 1$, $t_1 \in \mathbb{CD}_{\tau}^{\tau'}$ and $t_2 \in \mathbb{CD}_{\tau'}^{\tau_1,\ldots,\tau_n}$. Assume that we have a procedure `domain_entails` complying to the specification given in Figure 6. We define the *compact pre-composition* $t_1 \bullet t_2 \in \mathbb{CD}_{\tau}^{\tau_1,\ldots,\tau_n}$ as

$$t_1 \bullet t_2 = \left\{ (L \to r)\theta \;\middle|\; \begin{array}{l} L \to r' \in t_2,\ l \to r \in t_1 \text{ ren. apart,} \\ \mathtt{domain\_entails}(\tau', l, r') \text{ computes } \theta \end{array} \right\}.$$

Example 26. Consider the domain S^*_τ of Example 15. Let $\tau = [\mathtt{a} \mapsto int, \mathtt{b} \mapsto int\}$. A possible implementation of `domain_entails` is such that

- `domain_entails(`τ`,[X,-],[+,u])` succeeds and binds `X` to `+`.
- `domain_entails(`τ`,[+,X],[+,u])` 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 17).
- `domain_entails(`τ`,[+,+],[+,u])` just succeeds.
- `domain_entails(`τ`,[+,-],[+,+])` fails.

Example 27. Consider the computation of

$$\begin{array}{l} \mathtt{empty} \to \mathtt{empty} \\ [\mathtt{A},+] \to [-] \\ [\mathtt{A},-] \to [+] \end{array} \quad \bullet \quad \begin{array}{l} \mathtt{empty} \to \mathtt{empty} \\ [\mathtt{A},+] \to [-,\mathtt{u}] \\ [\mathtt{A},-] \to [+,+] \end{array}\ .$$

We follow Definition 25. We first consider every clause on the right. Thus we start with $\mathtt{empty} \to \mathtt{empty}$, whose output is entailed only by the input of the clause $\mathtt{empty} \to \mathtt{empty}$ on the left. Folding those clauses we obtain the clause $\mathtt{empty} \to \mathtt{empty}$ itself. We then consider the clause $[\mathtt{A},+] \to [-,\mathtt{u}]$ on the right. Its output $[-,\mathtt{u}]$ is entailed by the inputs $[\mathtt{A},+]$ and $[\mathtt{A},-]$ of two clauses on the left, provided we bind the variable `A` to $-$. This results in *two* clauses $[\mathtt{A},+] \to [-]$ and $[\mathtt{A},+] \to [+]$. Finally, we consider the clause $[\mathtt{A},-] \to [+,+]$ on the right, whose output is entailed by the input $[\mathtt{A},+]$ of a clause on the left provided we bind `A` to $+$. This results in the clause $[\mathtt{A},-] \to [-]$.

In conclusion, Definition 25 computes

$$\begin{array}{ll} \mathtt{empty} \to \mathtt{empty} & [\mathtt{A},+] \to [-] \\ [\mathtt{A},+] \to [+] & [\mathtt{A},-] \to [-] \end{array}\ . \tag{10}$$

Note that (10) is not a compact denotation, since condition ii of Definition 20 is not satisfied (there are overlapping inputs). We can prove that this is the only property of Definition 20 which is not preserved by $\bullet$ (Proposition 30). To this purpose, we first need some preliminary results.

Lemma 28. *Let $n \geq 1$ and $t \in \mathbb{CD}_{\tau}^{\tau_1,\ldots,\tau_n}$. Then $\overline{t} = \overline{\xi(t)}$.*

Proof. Let $L \in \mathsf{ui}(D_{\tau_1}) \times \cdots \times \mathsf{ui}(D_{\tau_n})$. We have

$$\begin{aligned}\overline{t}(L) &= \sqcup\{r\sigma \mid L' \to r \in t,\ \sigma \in \Lambda^g \text{ and } L'\sigma = L\}\\ &= \sqcup\{r \mid L' \to r \in \xi(t) \text{ and } L' = L\}\\ &= \overline{\xi(t)}(L)\ .\end{aligned}$$

□

Lemma 29. *In the hypotheses of Definition 25, we have $\xi(t_1 \bullet t_2) = \xi(t_1) \bullet \xi(t_2)$.*

Proof. We have that $\xi(t_1 \bullet t_2)$ can be rewritten as

$$\begin{aligned}&\left\{ (L \to r)\theta \;\middle|\; \begin{array}{l} L \to r' \in t_2,\ l \to r \in t_1 \text{ ren. apart,} \\ \mathtt{domain\_entails}(\tau', l, r') \text{ computes } \theta \\ \sigma \in \Lambda^g,\ \sigma \leq \theta \end{array} \right\}\\ &= \left\{ (L \to r)\sigma \;\middle|\; \begin{array}{l} L \to r' \in t_2,\ l \to r \in t_1 \text{ ren. apart,} \\ l\sigma \sqsubseteq r'\sigma,\ \sigma \in \Lambda^g \end{array} \right\}\\ &= \left\{ (L \to r)\sigma_1\sigma_2 \;\middle|\; \begin{array}{l} L \to r' \in t_2,\ l \to r \in t_1 \text{ ren. apart,} \\ l\sigma_1 \sqsubseteq r'\sigma_2,\ \sigma_1,\sigma_2 \in \Lambda^g \\ \mathsf{dom}(\sigma_1) \cap \mathsf{dom}(\sigma_2) = \varnothing \end{array} \right\}\\ &= \left\{ L \to r \;\middle|\; \begin{array}{l} L \to r' \in \xi(t_2),\ l \to r \in \xi(t_1) \\ \mathtt{domain\_entails}(\tau', l, r') \text{ succeeds} \end{array} \right\}\\ &= \xi(t_1) \bullet \xi(t_2)\ .\end{aligned}$$

□

Proposition 30. *The operation $\bullet$ of Definition 25 preserves conditions i and iii of Definition 20, but in general looses condition ii.*

Proof. We prove that condition i is maintained. We have

$$\begin{aligned}
&\cup\{\xi(L) \mid L \to r \in t_1 \bullet t_2\}\\
&= \cup\left\{\xi(L\theta)\;\middle|\; \begin{array}{l} L \to r' \in t_2,\ l \to r \in t_1 \text{ ren. apart,}\\ \texttt{domain_entails}(\tau', l, r') \text{ computes } \theta\end{array}\right\}\\
&= \left\{L\sigma\;\middle|\; \begin{array}{l} L \to r' \in t_2,\ l \to r \in t_1 \text{ ren. apart,}\\ \texttt{domain_entails}(\tau', l, r') \text{ computes } \theta\\ \sigma \in \Lambda^g,\ \sigma \le \theta\end{array}\right\}\\
&= \left\{L\sigma\;\middle|\; \begin{array}{l} L \to r' \in t_2,\ l \to r \in t_1 \text{ ren. apart,}\\ l\sigma \sqsubseteq r'\sigma,\ \sigma \in \Lambda^g\end{array}\right\}\ . \qquad (11)
\end{aligned}$$

Since both t_1 and t_2 satisfy condition i of Definition 20, (11) is equal to

$$\{L\sigma \mid L \to r' \in t_2,\ \sigma \in \Lambda^g\} = \mathsf{ui}(D_{\tau_1}) \times \cdots \times \mathsf{ui}(D_{\tau_n})\ .$$

We prove that condition iii is maintained. Let $L_1, L_2 \in \mathsf{ui}(D_{\tau_1}) \times \cdots \times \mathsf{ui}(D_{\tau_n})$ such that $L_1 \sqsubseteq L_2$. By Lemmas 28 and 29 we have

$$\begin{aligned}
(\overline{t_1 \bullet t_2})(L_1) &= \overline{\xi(t_1 \bullet t_2)}\\
&= \overline{\xi(t_1) \bullet \xi(t_2)}\\
&= \sqcup\left\{r\;\middle|\; \begin{array}{l} L_1 \to r' \in \xi(t_2),\ l \to r \in \xi(t_1)\\ \texttt{domain_entails}(\tau', l, r') \text{ succeeds}\end{array}\right\}\\
&= \sqcup\{r \mid L_1 \to r' \in \xi(t_2),\ l \to r \in \xi(t_1),\ l \sqsubseteq r'\}\\
(*) &\sqsubseteq \sqcup\{r \mid L_2 \to r' \in \xi(t_2),\ l \to r \in \xi(t_1),\ l \sqsubseteq r'\}\\
\text{(as before)} &= (\overline{t_1 \bullet t_2})(L_2)\ ,
\end{aligned}$$

where point $*$ follows by points ii and iii of Definition 20 applied to t_2.

Example 27 shows that condition ii can be lost. □

We can overcome the problem of Example 27 and of Proposition 30 through a *normalisation* procedure which makes disjunctive the inputs of a set of clauses, still preserving their meaning. It just iteratively splits two overlapping but distinct inputs into their intersection and their two differences. When no such inputs exist anymore, $\sqcup$ is applied to the outputs corresponding to the same input. Hence this `make_disjunctive` procedure can be defined through the `domain_intersect_ui`, `domain_subtract_ui` and `domain_lub` operations specified in Figure 6. The interest reader can find its definition inside our implementation (Section 7).

We can now define the counterpart of $\circ$ over $\mathbb{CD}$.

Definition 31. Let $n \ge 1$, $t_1 \in \mathbb{CD}_{\tau}^{\tau'}$ and $t_2 \in \mathbb{CD}_{\tau'}^{\tau_1,\ldots,\tau_n}$. We define

$$t_1 \circ^* t_2 = \texttt{make\_disjunctive}(t_1 \bullet t_2)\ .$$

Example 32. If we apply `make_disjunctive` to the set (10) of compact clauses, we just have to compute the $\sqcup$ of the two outputs for the same input $[\mathtt{A}, +]$. We obtain the compact denotation

$$\texttt{empty} \to \texttt{empty} \quad [\mathtt{A}, +] \to [\mathtt{u}] \quad [\mathtt{A}, -] \to [-]\ .$$

Example 33. We have

$$\begin{array}{l}\texttt{empty} \to \texttt{empty}\\ [\mathtt{A}, +] \to [+]\\ [+, -] \to [-]\\ [-, -] \to [+]\end{array} \bullet \begin{array}{l}\texttt{empty} \to \texttt{empty}\\ [\mathtt{X}] \to [\mathtt{X}, \mathtt{u}]\end{array} = \begin{array}{l}\texttt{empty} \to \texttt{empty}\\ [\mathtt{X}] \to [+]\\ [+] \to [-]\\ [-] \to [+]\end{array}\ .$$

The input of $[\mathtt{X}] \to [+]$ overlaps with that of both $[+] \to [-]$ and $[-] \to [+]$. The call `domain_intersect_ui(`τ`,[X],[+])` binds `X` to `+` while the call `domain_subtract_ui(`τ`,[X],[+])` binds `X` to `-`. The two clauses $[\mathtt{X}] \to [+]$ and $[+] \to [-]$ can hence be split and we obtain the set of clauses

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

After the `domain_lub` operation we obtain the compact denotation

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

Hence it is enough to prove that $\bullet$ mimics $\circ$ over $\mathbb{CD}$ (Proposition 35). The result for $\circ^*$ will follow. To that purpose, we need the following result.

Lemma 34. *In the hypotheses of Definition 25, if t_1 and t_2 are ground then $\overline{t_1 \bullet t_2} = \overline{t_1} \circ \overline{t_2}$.*

Proof. Let $L \in \mathsf{ui}(D_{\tau_1}) \times \cdots \times \mathsf{ui}(D_{\tau_n})$. We prove that $\overline{t_1 \bullet t_2}$ and $\overline{t_1} \circ \overline{t_2}$ coincide over L. This will entail the thesis by Definition 7.

Since t_1 and t_2 are ground we have:[3]

$$\begin{aligned}
&(\overline{t_1 \bullet t_2})(L)\\
&= \sqcup\{r \mid L \to r \in t_1 \bullet t_2\}\\
&= \bigsqcup\left\{r\;\middle|\; L \to r \in \left\{L'' \to r''\;\middle|\; \begin{array}{l} L'' \to r' \in t_2\\ l \to r'' \in t_1\\ \texttt{dom_entails}(\tau', l, r')\\ \text{succeeds}\end{array}\right\}\right\}\\
&= \bigsqcup\left\{r''\;\middle|\; \begin{array}{l} L \to r' \in t_2,\ l \to r'' \in t_1\\ \texttt{dom_entails}(\tau', l, r') \text{ succeeds}\end{array}\right\}\ . \qquad (12)
\end{aligned}$$

Since r' and l are ground, `dom_entails(`τ', l, r'`)` succeeds if and only if $l \sqsubseteq r'$. Hence, by Definition 20, (12) is equal to

$$\begin{aligned}
&\sqcup\{r'' \mid L \to r' \in t_2,\ l \to r'' \in t_1,\ l \sqsubseteq r'\}\\
&= \sqcup\{r'' \mid l \to r'' \in t_1 \text{ and } l \sqsubseteq r'\} \quad \text{with } L \to r' \in t_2\\
&= \overline{t_1}(r') \quad \text{with } L \to r' \in t_2\\
&= \overline{t_1}(\overline{t_2}(L)) = (\overline{t_1} \circ \overline{t_2})(L)\ .
\end{aligned}$$

□

Proposition 35. *In the hypotheses of Definition 25,*

$$\overline{t_1 \bullet t_2} = \overline{t_1} \circ \overline{t_2}\ .$$

Proof. Since $\xi(t_1)$ and $\xi(t_2)$ are ground, we have

$$\begin{aligned}
&\overline{t_1 \bullet t_2}\\
\text{(Lemma 28)} &= \overline{\xi(t_1 \bullet t_2)}\\
\text{(Lemma 29)} &= \overline{\xi(t_1) \bullet \xi(t_2)}\\
\text{(Lemma 34)} &= \overline{\xi(t_1)} \circ \overline{\xi(t_2)}\\
\text{(Lemma 28)} &= \overline{t_1} \circ \overline{t_2}\ .
\end{aligned}$$

□

[3] For space reasons, here we denote `domain_entails` by `dom_entails`.

Operation	Semantics
$\mathtt{domain\_entails}(\tau,\mathtt{A},\mathtt{B})$ with $\mathtt{A},\mathtt{B}\in D^*_\tau$	computes $\theta_1,\ldots,\theta_n\in\Lambda$ such that $\{\sigma\in\Lambda^g \mid \mathtt{A}\sigma\sqsubseteq\mathtt{B}\sigma\}=\{\sigma\in\Lambda^g \mid \sigma\le\theta_i \text{ for some } 1\le i\le n\}$
$\mathtt{domain\_intersect\_ui}(\tau,\mathtt{A},\mathtt{B})$ with $\mathtt{A},\mathtt{B}\in\mathsf{ui}(\mathtt{D}^*_\tau)$	computes $\theta_1,\ldots,\theta_n\in\Lambda$ such that for every $\sigma\in\Lambda^g$ we have $\mathtt{A}\theta_i\sigma=\mathtt{B}\theta_i\sigma$ and $\xi(\mathtt{A})\cap\xi(\mathtt{B})=\cup_{i=1,\ldots,n}\xi(\mathtt{A}\theta_i)$
$\mathtt{domain\_subtract\_ui}(\tau,\mathtt{A},\mathtt{B})$ with $\mathtt{A},\mathtt{B}\in\mathsf{ui}(D^*_\tau)$ and $\xi(\mathtt{A})\cap\xi(\mathtt{B})\neq\varnothing$	computes $\theta_1,\ldots,\theta_n\in\Lambda$ such that $\cup_{i=1,\ldots,n}\xi(\mathtt{A}\theta_i)=\xi(\mathtt{A})\setminus\xi(\mathtt{B})$
$\mathtt{domain\_lub}(\tau,\mathtt{A},\mathtt{B},\mathtt{L})$ with $\mathtt{A},\mathtt{B}\in D^*_\tau$	computes $\theta_1,\ldots,\theta_n\in\Lambda$ 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 $\mathtt{L}\theta_i\sigma=\mathtt{A}\theta_i\sigma\sqcup_{D_\tau}\mathtt{B}\theta_i\sigma$
$\mathtt{domain\_bottom}(\tau_{in},\tau_{out},\mathtt{B})$	computes $\mathtt{B}\in A^*_{\tau_{in},\tau_{out}}$ s.t. $\xi(\mathtt{B})=\bot_{A_{\tau_{in},\tau_{out}}}$
$\mathtt{domain\_the\_same}(\tau_{in},\tau_{out},\mathtt{A},\mathtt{B})$	checks whether $\xi(\mathtt{A})=\xi(\mathtt{B})$, with $\mathtt{A},\mathtt{B}\in A^*_{\tau_{in},\tau_{out}}$

Figure 6: Domain-specific operations.

6.2 The General Case of $\circ^*$

Consider the operation $\circ^*$ in its general form. If the clauses in T_i (Figure 5) cover $\mathsf{ui}(D_{\tau_1})\times\cdots\times\mathsf{ui}(D_{\tau_n})$ with the same inputs for every $1\le i\le n$, for every such input l we append the corresponding outputs. We obtain one compact denotation and we can proceed like in Subsection 6.1.

Example 36. Consider the computation over S^*_τ of

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

We can merge the two arguments into one and compute

$$\begin{array}{l}\mathtt{empty},\mathtt{empty}\to\mathtt{empty}\\ \mathtt{empty},[\mathtt{A}]\to\mathtt{empty}\\ [\mathtt{A}],\mathtt{empty}\to\mathtt{empty}\\ [\mathtt{A}],[+]\to[+]\\ [\mathtt{A}],[-]\to[-]\end{array}\circ^*\begin{array}{l}\mathtt{empty}\to\mathtt{empty},\mathtt{empty}\\ [+]\to[+],[+]\\ [-]\to[-],[\mathtt{u}]\end{array}$$

which, proceeding as in Subsection 6.1, is

$$\mathtt{empty}\to\mathtt{empty}\qquad[\mathtt{Z}]\to[+]\ .$$

Otherwise, we must reduce the arguments to the case seen above. We use here a procedure `pave` which splits the clauses in $T_1,\ldots,T_n$ in such a way that they finally cover $\mathsf{ui}(D_{\tau_1})\times\cdots\times\mathsf{ui}(D_{\tau_n})$ with the same inputs.

Example 37. Consider the computation over S^*_τ of

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

We apply `pave` to the two arguments and obtain

$$\begin{array}{l}\mathtt{empty},\mathtt{empty}\to\mathtt{empty}\\ \mathtt{empty},[\mathtt{A}]\to\mathtt{empty}\\ [\mathtt{A}],\mathtt{empty}\to\mathtt{empty}\\ [\mathtt{A}],[+]\to[+]\\ [\mathtt{A}],[-]\to[-]\end{array}\circ^*\begin{array}{l}\mathtt{empty}\to\mathtt{empty},\mathtt{empty}\\ [+]\to[+],[+]\\ [-]\to[-],[\mathtt{u}]\end{array}$$

which, proceedings as in Subsection 6.1, is

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

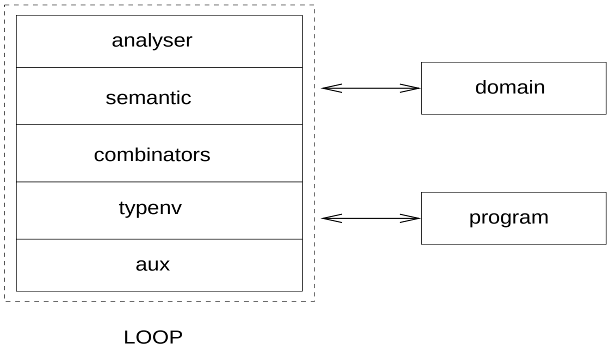

Figure 7: The structure of the LOOP analyser.

Figure 6 collects all the operations implemented by an abstract domain. The operation `domain_bottom` is used to start the fixpoint computation and `domain_the_same` to stop it. In addition to these operations, we recall that the programmer must specify a compact denotation for every basic operation *op* (see end of Section 5).

7 Implementation

7.1 Overview of the LOOP Analyser

The LOOP (Localised for Object-Oriented Programs) analyser [21] 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 [22] extended to deal with object-oriented features through the operations of [13].

The structure of Loop is given in Figure 7. The highest-level module `analyser` implements a fixpoint engine for a denotational semantics in terms of the operations in `semantic` (for instructions, conditionals, loops). Those operations are themselves compiled in terms of calls to `combinators`, a module which implements the operations of Figure 5. The module `typenv` implements the type environments (Definition 1). The module `aux` implements auxiliary and logging functions. External modules must contain the abstract syn-

analysis	reference	ground	non-ground	bdd's
sign analysis	Equation (1)	`signs`	`signs_vars`	`signs_bdd`
rapid type (*i.e.*, class) analysis	[1, 13]	`rt`	`rt_vars`	
dataflow class analysis	[6, 13]	`df`	`df_vars`	
constraint class analysis	[20, 13]	`ps`	`ps_vars`	
escape analysis	[9]	`e`	`e_vars`	

Figure 8: The abstract domains implemented for our LOOP analyser.

tax of the code of the program to be analysed as well as the abstract domains and their operations. Note that LOOP can perform both an abstract interpretation and a combination of abstract compilation [8] and partial evaluation of the program. The result is the same in both cases, but abstract compilation is in general cheaper in time and space and is the one used in our experiments.

The analysis computed by LOOP is automatically *focused* on a set of program points deemed *of interest* for the analysis. Those program points are called *watchpoints* [22]. In general, the cost of the analysis depends on the number of watchpoints # *wpnts*; so that with #*wpnts* = 0, just an input/output analysis of the program is performed.

7.2 Five Abstract Analyses

We have implemented five abstract analyses (domains) using LOOP both with a standard ground representation and through a non-ground representation, as shown in this paper. Moreover, in the case of sign analysis we have also implemented the domain using bdd's [3], normally viewed as the most efficient way for representing abstract denotations. Figure 8 shows those domains. In each case, the operations of Figure 6 have been implemented in a separate module.

All our domains have been implemented in Prolog. Lists have been used to represent sets of classes or sets of creation points. Only `signs_bdd` has been implemented in C++, by using the BuDDy library for bdd's [16].

7.3 Experimental Evaluation

Our experiments have been performed by using SICStus Prolog version 3.8.5 on a Pentium III 736 Mhz machine with 256 Mbytes of RAM.

For sign analysis, we use three benchmarks: `fib`, the Fibonacci procedure; `nested`, a recursive procedure over an array where the recursive call is inside a `while` loop and uses an auxiliary procedure `swap` that swaps two array elements; and `arith` which implements some arbitrary precision arithmetic operations. For class and escape analysis we use three benchmarks: `inv`, a procedure that inverts the class of a variable twice through two calls inside a `while` loop to a virtual function; `clone`, which implements a generic cloning of a list by means of a virtual call to the `clone` method of the elements of the list; and `figures` which implements some classes of geometric figures with a generic method for rotating them.

Figure 9 compares 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 also shows how the analysis scales with the number of watchpoints. Figures 10, 11 and 12 do the same for the three

Bmrk	# wpnts	`signs`		`signs_vars`	
		time	space	time	space
`fib`	0	1.683	348291	0.049	7658
`fib`	3	2.186	478132	0.119	15896
`fib`	6	2.234	493412	0.147	20771
`nested`	0	118.732	13453432	0.564	56390
`nested`	7	119.004	13671816	0.859	145752
`nested`	13	120.012	13852568	1.149	230778
`arith`	0	108.578	13617292	0.163	26242
`arith`	7	108.993	13654012	0.181	32238
`arith`	14	109.453	13730012	0.198	38485

Figure 9: Time and space required for sign analysis.

Bmrk	# wpnts	`rt`		`rt_vars`	
		time	space	time	space
`inv`	0	0.048	16608	0.016	5556
`inv`	2	0.051	17946	0.020	6894
`inv`	4	0.058	18906	0.022	7854
`clone`	0	0.071	22951	0.042	11563
`clone`	6	0.095	27805	0.055	16417
`clone`	11	0.125	40457	0.079	26650
`figures`	0	2.791	696952	1.106	216494
`figures`	9	2.881	739802	1.128	259344
`figures`	17	2.956	776338	1.193	295880

Figure 10: Time and space required for class analysis: I.

Bmrk	# wpnts	`df`		`df_vars`	
		time	space	time	space
`inv`	0	0.431	94363	0.032	6823
`inv`	2	0.441	98637	0.034	8077
`inv`	4	0.444	101745	0.035	8879
`clone`	0	11.377	2293636	0.070	11527
`clone`	6	11.450	2320420	0.078	18935
`clone`	11	15.297	3149401	0.296	78574
`figures`	0	0.934	166487	0.092	26099
`figures`	9	0.971	177605	0.106	32497
`figures`	17	1.071	188949	0.132	37938

Figure 11: Time and space required for class analysis: II.

Bmrk	# wpnts	ps		ps_vars	
		time	space	time	space
`inv`	0	0.460	105601	0.032	7619
`inv`	2	0.464	110381	0.034	9031
`inv`	4	0.472	113857	0.035	9933
`clone`	0	409.298	29282466	0.110	25307
`clone`	6	420.844	29393040	0.137	47855
`clone`	11	550.112	39337617	0.461	240844
`figures`	0	3.621	844796	0.183	53094
`figures`	9	3.719	891134	0.187	70796
`figures`	17	3.725	936542	0.200	85612

Figure 12: Time and space required for class analysis: III.

Bmrk	# wpnts	e		e_vars	
		time	space	time	space
`inv`	0	0.421	12722	0.205	7930
`inv`	2	0.432	14060	0.217	9268
`inv`	4	0.442	15020	0.227	10228
`clone`	0	0.867	15398	0.556	10872
`clone`	6	1.122	24680	0.745	19102
`clone`	11	1.166	31800	0.789	26222
`figures`	0	6.588	142880	4.179	88280
`figures`	9	6.725	161564	4.336	110588
`figures`	17	6.894	181208	4.491	133520

Figure 13: Time and space required for escape analysis.

domains for class analysis. Finally, Figure 13 compares time and space for the domain for escape analysis. These show that the domain `X_vars` is always more efficient than the domain `X`. It gains up to three orders of magnitude in time and space. The cost in space of the analyses *i.e.,* the amount of data structures built during them, is independent from the choice of language used to implement LOOP indicating that their time cost is also probably unrelated to this choice.

It is interesting to compare our sample implementation in Prolog with another based on a highly-optimised library for bdd's [16]. Figure 14 compares our compact representation of the domain for sign analysis (Equation (1)) with another implementation we have built through bdd's. We have run the test with two different initial numbers of bdd nodes and cache size. These are suggested by the authors of the bdd library [16] for "small" and "large" examples, respectively. Figure 14 shows that a compact representation through logic programs has an efficiency comparable with that of an implementation based on bdd's. The more efficient results in the column headed *large* were obtained by considering the sign analysis of programs of more than thirty lines long as a "large" example, making the maximal amount of memory available to the bdd library, and by disabling the garbage collection of bdd nodes.

8 Related Works

Logic programs with variables have been used to model type dependencies in the input/output behaviour of a piece of ML code [10, 18]. However, there the abstract denotations are just a single clause (a single type dependency). A similar technique has been applied to the logic programs themselves [11, 15]. The applicability of logic programs to other analyses and programming paradigms has not before been studied.

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 [14, 19]. *W.r.t.* [14], our notion of union-irreducible elements (Definition 2) might force the analyser to consider more points in the input of a denotation, but will allow more precise non-(component-)additive denotations to be represented.

9 Conclusion

We have shown how abstract denotations can be implemented by using logic programs whose variables represent dependencies between the input and the output of a denotation. We have provided an experimental evaluation which shows that this technique reduces both the time and the space requirements of sign, class and escape analysis *w.r.t. ground* implementations, and has an efficiency comparable with that of implementations based on binary decision diagrams.

There are however several advantages in using our technique, compared with bdd's.

- Logic programs are closer to the specification since denotations are immediately recognisable as logic programs (*i.e.,* sets of inputs/output arrows), while this is not true for a bdd which implements a formula to code the denotation.
- The symbolic capability of logic programs helps the programming task. For instance, when abstract inputs and outputs are sets, such as in class and escape analysis, the bdd-based requires to *code* such sets through binary digits, whereas logic programs can use lists.
- Our technique just requires the unification procedure and not the whole power of logic programming. Section 7 shows that only a relatively small amount of memory is needed, with efficiency similar to that obtained with a bdd library which requires a large statically allocated space for the bdd nodes and the cache (Figure 14). This makes our technique ideal when memory is a concern, for instance for the static analysis of applications uploaded onto a smart card.

We would like to use constraint logic programs [12] rather than logic programs to represent abstract denotations. The constraints allow a flexible representation of the dependencies contained in a denotation. In particular, abstract domains based on sets (such as the domains for class and escape analysis considered in this paper) should benefit from the representation of abstract denotations in terms of programs over set-constraints [7].

References

[1] 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.

Benchmark	# wpnts	signs_vars	signs_bdd "small"	signs_bdd "large"
		time	time	time
fib	0	0.049	0.047	0.026
fib	3	0.119	0.111	0.064
fib	6	0.147	0.131	0.091
nested	0	0.564	0.758	0.195
nested	7	0.859	1.659	0.484
nested	13	1.149	3.281	0.916
arith	0	0.163	0.364	0.114
arith	7	0.181	0.393	0.136
arith	14	0.198	0.481	0.168

Figure 14: Our compact representation compared with another based on binary decision diagrams.

[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] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] M. P. Hill and F. Spoto. A Foundation of Escape Analysis. Submitted for publication, 2002.

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

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] J. Lind-Nielsen. BuDDy - A Binary Decision Diagram Package. `http://www.itu.dk/research/buddy/`.

[17] T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification*. Java^{TM} Series. Addison-Wesley, 1999.

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

[19] 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.

[20] 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.

[21] F. Spoto. The LOOP Analyser. `http://www.sci.univr.it/~spoto/loop/`.

[22] 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.