Logic Programs as Compact Denotations*

Patricia M. Hill[1] and Fausto Spoto[2]

[1] School of Computing, University of Leeds, LS2 9JT, Leeds, UK
hill@comp.leeds.ac.uk
[2] Dipartimento di Informatica, Università di Verona
Strada Le Grazie, 15, Ca' Vignal, 37134 Verona, Italy
fausto.spoto@univr.it

Abstract. This paper shows how logic programs can be used to implement the transition functions of denotational abstract interpretation. The logic variables express regularity in the abstract behaviour of commands. The technique has been applied to sign, class and escape analysis for object-oriented programs. We show that the time and space costs using logic programs for these analyses are smaller than those using a ground relational representation. Moreover, we show that, in the case of sign analysis, our technique requires less memory and has an efficiency comparable to that of an implementation based on binary decision diagrams.

Keywords: *Logic programs, denotational semantics, static analysis, abstract interpretation.*

1 Introduction

Abstract interpretation [1] is a framework for the formal analysis and transformation of programs. It can be seen as a generalisation and formalisation of other techniques for static analysis like the traditional dataflow analysis of programming languages [2]. The information it can derive statically (*i.e.,* at compile-time) about the run-time behaviour of the program has been used for debugging, documentation and optimisation of the compiled code.

Abstract interpretation is semantics-based in the sense that the derived information is a (correct) approximation of information in the concrete semantics of the language. In principle, provided it captures the property of interest, any type of program language semantics may be used for the concrete semantics. However, the choice of denotational semantics is appealing since it leads to a static analysis that is compositional. In this context, the abstract denotation (the *transition* or *transfer function*) of a piece of code is a function from the abstract properties of its input to those of its output. Provided the domain of abstract properties is finite, this relational approach can be naively implemented through a *ground* relational database. However, as such a domain can be very large, the traditional way to implement these relations is by means of a more compact representation based on *binary decision diagrams* (*bdd's*) [3]; a bdd denotes a propositional formula that defines the input/output relation determined by the abstract denotation.

In logic, general rules are defined by non-ground formulas and, in particular, ground relational databases can be defined very compactly by (non-ground) logic programs. Thus, in this paper, we investigate (both theoretically and experimentally) a *non-ground* representation, implemented as a logic program, for 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. Dependencies like

* This work has been funded by the MURST grant "Abstract Interpretation, type systems and control-flow analysis" and the EPSRC grant GR/R53401.

these can be naturally represented by non-ground clauses of a logic program. We plug this non-ground representation inside a fixpoint static analyser for object-oriented programs. The fixpoint computation is stopped by a two-stage equivalence test on denotations: first variance (*i.e.,* equality up to variable renaming) and then, if variance fails, explicit model comparison.

The first contribution of this paper is to describe and prove correct the methodology. This amounts to the formalisation of a compact representation for the transition functions through non-ground logic programs. We introduce the notion of *union-irreducibility* for the elements of an abstract domain and show how this notion can be used to reduce the size of the denotations.

The second contribution is the instantiation of that methodology for the following two static analyses:

– *Sign analysis.* This approximates integer variables by their sign using the domain defined in [4]. This is described in more detail in the next section.
– *Class analysis* for object-oriented programs. This overapproximates 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 analyses here are those formalised in [5] by abstract interpretation and are derived from the ideas in [6], [7] and [8], respectively. *Rapid type analysis* [6] provides just one set of classes which approximates all the classes instantiated up to a given program point. The *dataflow analysis* of [7] is more precise since it approximates every program variable with a set of possible classes. It is described in more detail in the next section. Although more precise than [6], [7] does not provide any approximation for the fields of the objects. This is instead the main feature of the *constraint analysis* in [8].

The third contribution is an experimental comparison of our technique (using non-ground logic programs) with the one based on the naive ground representation through ground logic programs. We compare here the sign and class analyses described above together with escape analysis for object-oriented languages. Note that:

– *Escape analysis* [9] determines which dynamically created objects will not *escape* from their creating method. This information is useful since it allows one to allocate these objects to the stack instead of the heap, reducing 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 [10], which collects the set of *creation points* of the objects reachable at a given program point.

The fourth contribution is to show that our implementation of sign analysis through non-ground logic programs is comparable to a bdd implementation that however requires a large amount of memory to attain that efficiency (the bdd implementation of sign analysis is due to Aurélie Lagasse). Note that we are considering different *implementations* of the same abstract domains. The precision of the analyses does not change, whether we use ground, non-ground or bdd-based implementations.

In the next section we describe both sign analysis for imperative and object-oriented programs and class analysis for object-oriented programs. We use these to illustrate the use of non-ground terms to provide a compact representation for the abstract denotations of imperative and object-oriented programs and motivate the rest of the paper. In Section 3, we introduce some technical notation and terminology needed for the rest of the paper. In order to formalise our compact representation and its analysis, we need to specify the ground relational approach and the underlying analysis framework. This is done in Section 4 where we introduce an important concept called *union-irreducibility*; elements in the ground representation are called *union-irreducible* if they convey information that cannot be constructed from that conveyed by strictly smaller elements. By contrast, a *union-reducible* element does not need to be explicitly represented in an implementation since the information it conveys is redundant. We also introduce the abstract operations for ground denotations and show how they preserve this union-irreducibility property. The compact denotations illustrated in Section 2 are described more formally in Section 5. We present the abstract domain of non-ground terms and also how we use non-ground logical clauses to represent more compactly the union-irreducible operations. Operations such as those needed for equivalence testing and composition are language independent and one implementation suffices for all instances of our framework. In Section 6, we specify and describe the use of predicates needed to implement these operations. In Section 7, we describe our implementation of the compact denotation and provide tables summarising our experimental results. Finally, in Section 8, we discuss related work and the advantages of using the methodology decribed here.

A preliminary and partial version of this paper appeared in [11].

2 Some Motivating Examples

Consider the problem of the *sign analysis* of a program. Assuming there is a set of types that includes the type *int*, sign analysis tries to determine the sign (positive or negative) of the variables of type *int*. We use here the domain $\{*, +, -, u\}$ which is similar to that defined in [4]. 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 is a *type environment i.e.,* a map which assigns a type to each element of $V = \mathsf{dom}(\tau)$. Then we define a domain

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

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 element empty represents the empty set of states.

The set S_τ is partially ordered *w.r.t.* $\sqsubseteq$ where $\mathsf{empty} \sqsubseteq s$ 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, $1 \le i \le \ell$. An element $\varsigma \in S_\tau \setminus \{\mathsf{empty}\}$ is represented by $[x_1, \ldots, x_\ell]$, where v_i is the ith variable in lexicographical order in $\{v_1, \ldots, v_\ell\}$ and $\varsigma(v_i) = x_i$ for every $i = 1, \ldots, \ell$.

2.1 Representing Abstract Denotations

Consider the following program, written in a pseudo-syntax for a simple imperative object-oriented language *i.e.,* Pascal functions with objects, fields and virtual calls.

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

The denotation of the assignment `a:=b` *w.r.t.* sign information is a (*transition*) function from the abstract sign properties of the variables before the assignment to those of the variables after it. Since only the variables `a`, `b`, `c` and `out` are addressable at that program point (we assume that the special variable `out` holds the return value of the function) and they all have type *int*, that denotation is a function $f : S_\tau \mapsto S_\tau$ (with $\tau = [\mathsf{a} \mapsto \mathit{int}, \mathsf{b} \mapsto \mathit{int}, \mathsf{c} \mapsto \mathit{int}, \mathsf{out} \mapsto \mathit{int}]$) which can be represented as

$$\begin{array}{ll} \mathsf{empty} \to \mathsf{empty} & \\ [+,+,+,+] \to [+,+,+,+] & [+,+,+,-] \to [+,+,+,-] \\ [+,+,-,+] \to [+,+,-,+] & [+,+,-,-] \to [+,+,-,-] \\ [+,-,+,+] \to [-,-,+,+] & [+,-,+,-] \to [-,-,+,-] \\ [+,-,-,+] \to [-,-,-,+] & \ldots \end{array} \tag{2}$$

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 [12, 13, 4] (see Definition 1 in Section 4). For instance,

$$f([+,u,+,+]) = f([+,+,+,+]) \sqcup f([+,-,+,+]) = [+,+,+,+] \sqcup [-,-,+,+] = [u,u,+,+].$$

Mapping (2) can be naturally seen as a ground logic program stating an input/output relationship *i.e.,*

$$\begin{array}{l} \texttt{io}(\texttt{empty},\texttt{empty}).\\ \texttt{io}([+,+,+,+],[+,+,+,+]).\\ \texttt{io}([+,+,+,-],[+,+,+,-]).\\ \ldots \end{array}$$

This simple representation is not practical as its size grows exponentially with the number of program variables. Note that the same would happen if we used a domain which collects the set of definitely positive and that of definitely negative variables, since we might still have to consider an exponential number of input configurations for a given command.

However, there is some regularity in (2); namely, f does not change the values of `b`, `c` and `out`; and the input value of `b` always becomes the output value of `a`. We can therefore represent f, *by using logical variables* corresponding to the original program variables, as

$$\texttt{empty} \rightarrow \texttt{empty} \qquad [\texttt{A},\texttt{B},\texttt{C},\texttt{Out}] \rightarrow [\texttt{B},\texttt{B},\texttt{C},\texttt{Out}]. \tag{3}$$

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

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

From now on, the notation in (3) denotes the corresponding program in (4).

Consider now the program:

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

The denotation of the assignment `a:=b-1` can be represented by the program:

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

Although (5) is slightly more complex than (3), it is more compact than an exhaustive representation and its size grows linearly with the number of variables.

2.2 Sign Analysis of Imperative Programs

As explained 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. Figure 1 shows the program `nested` with its sign analysis. This analysis is first computed by using the domain S_τ in (1) implemented through ground dependencies (arrays of integers are treated as if they were a single integer *i.e.,* their sign is the least upper bound of the signs of their elements) and then through a compact representation which uses logic variables. The result is the same with both analyses, but variables provide a more compact representation. Note, moreover, that computing the ground dependencies required 118.732 seconds and 13453432 Prolog atoms

```
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,b: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);
        }
  }
}
```

$\texttt{nested} : [\texttt{a} \mapsto \mathit{int}, \texttt{b} \mapsto \mathit{int}, \texttt{n} \mapsto \mathit{int}] \to [\texttt{out} \mapsto \mathit{int}]$

$$\begin{array}{llll}
\mathsf{empty} \to \mathsf{empty} & & & \\
[+,+,+] \to [+] & [+,+,-] \to [+] & [+,-,+] \to [\texttt{u}] & [+,-,-] \to [+] \\
[-,+,+] \to [\texttt{u}] & [-,+,-] \to [+] & [-,-,+] \to [+] & [-,-,-] \to [+]
\end{array}$$

$$\begin{array}{llll}
\mathsf{empty} \to \mathsf{empty} & [\texttt{A},\texttt{B},-] \to [+] & & \\
[+,+,+] \to [+] & [-,-,+] \to [+] & [-,+,+] \to [\texttt{u}] & [+,-,+] \to [\texttt{u}]
\end{array}$$

Fig. 1. The program `nested` and its ground and non-ground sign analysis.

in the stack of our machine, while computing their compact representation required only 0.564 seconds and just 56390 atoms (Figure 7). This definitely justifies our investigation.

2.3 Class Analysis of Object-Oriented Programs

Figure 2 shows three simple classes written in an object-oriented language and their class analysis through the domain in [7], as formalised in [5] and shown below. We assume to have a finite set of *types* ordered *w.r.t.* a subclass relation. Types can be *elementary types*, like *int*, or *classes i.e.,* programmer-defined types.

$$CL_\tau = \{\mathsf{empty}\} \cup \left\{ \phi : \mathsf{dom}(\tau) \mapsto \wp(\mathit{types}) \;\middle|\; \begin{array}{l} \text{for every } v \in \mathsf{dom}(\tau) \\ \ \phi(v) \text{ are subtypes of } \tau(v) \\ \text{and if } \texttt{this} \in \mathsf{dom}(\tau) \\ \ \text{then } \phi(\texttt{this}) \neq \varnothing \end{array} \right\} . \qquad (6)$$

That abstract domain contains a bottom element empty, which represents the empty set of states, and functions over the variables of interest, binding each variable to a set of subtypes of its declared type. The variable `this` cannot be bound to $\varnothing$ because a `this` object must always exist. These functions can be represented by lists of sets of classes, one for every variable of interest. For instance, if the type environment describing the variables of interest is $\tau = [\texttt{this} \mapsto \texttt{main}, \texttt{x} \mapsto \texttt{a}]$, then the abstract element $[[\texttt{main}], [\texttt{b}]]$ represents the set of states where `this` is bound to an object of class `main` and `x` is bound to an object of class `b` (which is legal since `b` is a subclass of `a`, see

```
class a {           class b extends a {     class main {
  f:int;              g:int;                  clone(x:a):a {
  next:a;                                       if x=nil then out:=nil;
                      clone():b {               else {
  clone():a {           out:=new(b);              out:=x.clone();
    out:=new(a);        out.f:=this.f;            out.next:=
    out.f:=this.f;      out.g:=this.g;              this.clone(x.next);
  }                    }                        }
}                    }                      } }
```

$\mathtt{main.clone}\!:[\mathtt{this} \mapsto \mathtt{main}, \mathtt{x} \mapsto \mathtt{a}] \to [\mathtt{out} \mapsto \mathtt{a}]$

$$\mathtt{empty} \to \mathtt{empty} \qquad [[\mathtt{main}], []] \to [[]] \qquad [[\mathtt{main}], [\mathtt{a}]] \to [[\mathtt{a}]] \qquad [[\mathtt{main}], [\mathtt{b}]] \to [[\mathtt{b}]]$$

$\mathtt{a.clone}\!:[\mathtt{this} \mapsto \mathtt{a}] \to [\mathtt{out} \mapsto \mathtt{a}]$

$$\mathtt{empty} \to \mathtt{empty} \qquad [[\mathtt{a}]] \to [[\mathtt{a}]] \qquad [[\mathtt{b}]] \to [[\mathtt{a}]]$$

$\mathtt{b.clone}\!:[\mathtt{this} \mapsto \mathtt{b}] \to [\mathtt{out} \mapsto \mathtt{b}]$

$$\mathtt{empty} \to \mathtt{empty} \qquad [[\mathtt{b}]] \to [[\mathtt{b}]]$$

Fig. 2. Classes `a`, `b` and `main` and the ground class analysis of their `clone` methods.

Figure 2). The abstract element $[[\mathtt{main}], []]$ represents those states where `x` is bound to *nil*. Note that `[[],[b]]` is not an abstract element since we do not allow `this` to be bound to *nil* [5]. The domain CL_τ is partially ordered as $\mathsf{empty} \sqsubseteq s$ for every $s \in CL_\tau$ and $\phi_1 \sqsubseteq \phi_2$ if $\phi_1(v) \subseteq \phi_2(v)$ for every $v \in \mathsf{dom}(\tau)$.

The result of the analysis in Figure 2 shows that the method `clone` of the class `main` always returns an object of the same class as its `x` parameter. The same analysis, performed through a non-ground implementation of the same domain, yields the same results except for the denotation of the method `clone` of the class `a`, which is now

$$\mathtt{empty} \to \mathtt{empty} \qquad [[\mathtt{C}]] \to [[\mathtt{a}]]$$

Note that also the denotation of the method `clone` of the class `main` could be compacted by using a logic variable and become:

$$\mathtt{empty} \to \mathtt{empty} \qquad [[\mathtt{main}], \mathtt{X}] \to [\mathtt{X}]$$

Our analyser lost the regularity which that denotation contains. Nevertheless, the non-ground analysis, compared to the ground one, is performed in 0.07 seconds instead of 11.377 and requires to keep in memory 11527 Prolog atoms instead of 2293636 (Figure 8).

We conclude that a major increase in the efficiency of the analysis is achieved through a non-ground representation although the final results of the analyses could be further compacted. This is because the partial computations of the analyser are significantly enhanced by the non-ground representation. A further boost in performance could be achieved by recovering some regularity which is lost during the analysis. This aspect has, however, to be investigated further.

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. The cardinality of a set S is denoted by $\#S$.

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 [1], 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 ($\gamma\alpha(c) \geq c$) and $\alpha\gamma$ is reductive ($\alpha\gamma(a) \leq a$). In such a case, α is *additive i.e.,* it preserves lub's.

Given a finite set of variables *Vars* and a set of function symbols Σ with associated arity, the set of *terms* over *Vars* and Σ 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 (up to variable renaming) is defined as $t_1 \leq t_2$ if there exists a substitution σ such that $t_1 = t_2\sigma$. The partial ordering on substitutions is defined as $\theta_1 \leq \theta_2$ if there exists a substitution σ such that $\theta_1 = \theta_2\sigma$. We assume that *Vars* is lexicographically ordered.

4 The Framework of Analysis

In order to formalise our compact representation and its analysis, we need to specify the ground relational approach and the underlying analysis framework.

Although all our examples are in an imperative object-oriented language, this framework is not bound to any particular programming paradigm; 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 sequential *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 and composition is composition of substitutions. In the case of imperative programs, the basic denotations are the denotations of every (concrete or abstract) single command or bytecode and composition is the denotation of statement composition (semicolon).

Recall from Section 2 that a *type environment* τ is a map which assigns a type to the variables in its domain. From now on, τ will silently stand for a type environment. We suppose that there is a set C_τ of the concrete *states* of the computation for the variables described by τ. The exact nature of these states is irrelevant for this paper. They might be substitutions in the case of logic programs, or pairs of environment and memory in the case of an imperative language [5]. We assume that there is a generic abstract domain $\langle D_\tau, \sqsubseteq \rangle$ with no infinite descending chains, related to $\langle \wp(C_\tau), \subseteq \rangle$ through a Galois connection. We are interested in those elements in D_τ which convey some information that cannot be constructed by merging the information conveyed by the strictly smaller elements.

Definition 1. *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 1. Consider the domain S_τ in (1). Then

$$\mathsf{ui}(S_\tau) = \{\mathsf{empty}\} \cup \{\varsigma \in S_\tau \setminus \{\mathsf{empty}\} \mid \varsigma(v) \neq u \text{ for every } v \in \mathsf{dom}(\tau)\} \ .$$

Let $\mathsf{dom}(\tau) = \{v_1, \ldots, v_\ell\}$. It can be shown that the variables abstracted into u stand for the disjunction of $+$ and $-$ (for the concretisation map γ^S_τ, see [4]). Namely, for every $\varsigma \in S_\tau \setminus \{\mathsf{empty}\}$

we have

$$\gamma_\tau^S(\varsigma) = \bigcup \left\{ \gamma_\tau^S([s_1,\ldots,s_\ell]) \;\middle|\; \begin{array}{l} s_i \in \{+,-,*\} \\ s_i \preceq \varsigma(v_i) \text{ for } 1 \le i \le \ell \end{array} \right\} \cup \gamma_\tau^S(\mathsf{empty}) \ .$$

If $\varsigma \in \mathsf{ur}(S_\tau)$, then there exists some $1 \le j \le \ell$ such that $\varsigma(v_j) = u$. Thus, in this case, we have $\gamma_\tau^S(\varsigma) = \cup\{\gamma_\tau^S(\varsigma') \mid \varsigma' \sqsubset \varsigma\}$.

Example 2. Consider the domain CL_τ in (6). The union-irreducible elements of CL_τ are those that approximate each variable with an empty or singleton set:

$$\mathsf{ui}(CL_\tau) = \{\mathsf{empty}\} \cup \{\phi \in CL_\tau \setminus \{\mathsf{empty}\} \mid \#\phi(v) \le 1 \text{ for every } v \in \mathsf{dom}(\tau)\} \ .$$

Let $\mathsf{dom}(\tau) = \{v_1,\ldots,v_\ell\}$. It can be shown that the variables abstracted into a set s with more than one element actually stand for the disjunction of the elements in s (for the concretisation map γ_τ^{CL}, see [5]). Namely, for every $\phi \in CL_\tau \setminus \{\mathsf{empty}\}$ we have

$$\gamma_\tau^{CL}(\phi) = \bigcup \left\{ \gamma_\tau^{CL}([s_1,\ldots,s_\ell]) \;\middle|\; \begin{array}{l} \#s_i \le 1 \\ s_i \subseteq \phi(v_i) \text{ for } 1 \le i \le \ell \end{array} \right\} \cup \gamma_\tau^{CL}(\mathsf{empty}) \ .$$

If $\phi \in \mathsf{ur}(CL_\tau)$, then there exists some $1 \le j \le \ell$ such that $\#\phi(v_j) > 1$. Thus, in this case, we have $\gamma_\tau^{CL}(\phi) = \cup\{\gamma_\tau^{CL}(\phi') \mid \phi' \sqsubset \phi\}$.

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

Proposition 1. *Let $d \in \mathsf{ur}(D_\tau)$. Then $\gamma_\tau(d) = \cup\{\gamma_\tau(d') \mid d' \in \mathsf{ui}(D_\tau),\ d' \sqsubset d\}$.*

Proof. By an iterated application of Definition 1, because D_τ does not contain infinite descending chains. □

We are interested in functions that can be identified by their restriction to the union-irreducible elements. Such functions do not introduce imprecision in the approximation of the union-reducible elements and, in practice, only the union-irreducible elements are *meaningful* to represent the functions.

Definition 2. *Let $n \ge 1$ and $a \in D_{\tau_1} \times \cdots \times D_{\tau_n} \mapsto D_\tau$. We say that a is* ui-induced *if for every $(d_1,\ldots,d_n) \in D_{\tau_1} \times \cdots \times D_{\tau_n}$ such that there exists j, $1 \le j \le n$, with $d_j \in \mathsf{ur}(D_{\tau_j})$, we have*

$$a(d_1,\ldots,d_n) = \sqcup\{a(d_1,\ldots,d',\ldots,d_n) \mid d' \in \mathsf{ui}(D_{\tau_j}) \text{ and } d' \sqsubset d_j\} \ .$$

Example 3. Let $\tau = [\mathtt{v} \mapsto int, \mathtt{w} \mapsto int]$ and consider the domain S_τ in (1). The map $a : D_\tau \mapsto D_\tau$ defined as

$$\begin{array}{llll} a(\mathsf{empty}) = [+,+] & a([+,+]) = [+,u] & a([+,-]) = [+,+] & a([+,u]) = [+,u] \\ & a([-,+]) = [-,+] & a([-,-]) = \mathsf{empty} & a([-,u]) = [u,+] \\ & a([u,+]) = [+,u] & a([u,-]) = [+,-] & a([u,u]) = [u,u] \end{array} \quad (7)$$

is not ui-induced, since $a([u,+]) \ne [u,u]$ but $\{s \in S_\tau \mid s \sqsubset [u,+]\} = \{\mathsf{empty}, [+,+], [-,+]\}$ and $a(\mathsf{empty}) \sqcup a([+,+]) \sqcup a([-,+]) = [u,u]$. A ui-induced map is obtained by replacing line (7) with

$$a([u,+]) = [u,u] \qquad a([u,-]) = [+,+] \qquad a([u,u]) = [u,u] \ . \quad (8)$$

The monotonic maps over $\wp(C)$ are approximated by *abstract* denotations.

Definition 3. *Let $n \ge 1$. The set $\mathbb{D}_\tau^{\tau_1,\ldots,\tau_n}$ 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 $\sqsubseteq$.*

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

Fig. 3. Signature and implementation of the operations on denotations.

Example 4. Neither map a of Example 3 nor that obtained from a by replacing (7) by (8) are (τ,τ)-denotations. This is because $\mathsf{empty} \sqsubseteq [-,+]$ but $a(\mathsf{empty}) \not\sqsubseteq a([-,+])$. Hence a is not monotonic. Instead, the map $b : D_\tau \mapsto D_\tau$ defined as

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

is both monotonic and ui-induced and, hence, a (τ,τ)-denotation. As line (9) is induced by the previous ones, it is *superfluous* (i.e., ignored in an implementation).

Figure 3 defines the abstract denotation of every basic operation op by using its best approximation $\alpha op \gamma$ and also defines the composition of abstract denotations. Proposition 2 proves that the operations in Figure 3 are well-defined *i.e.,* they compute monotonic and ui-induced maps (Definition 3).

Proposition 2. *The operations in Figure 3 are well-defined.*

Proof. We have to prove that the conditions of Definition 3 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 $1 \le i \le n$ and let $d_j \in \mathsf{ur}(D_{\tau_j})$ for a given $1 \le j \le n$. Let $J = \{d' \in \mathsf{ui}(D_{\tau_j}) \mid d' \sqsubset d_j\}$. By Proposition 1, since op is a pointwise extension and α_τ is additive, we have

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

□

Example 5. Consider the abstract domain S_τ in (1) where $\tau = [\mathtt{v} \mapsto int, \mathtt{w} \mapsto int]$. The denotation of the operation $\mathtt{w} := \mathtt{v}$ on the concrete domain is approximated by the following denotation on S_τ

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

The concrete operation `if v` $\geq$ `0 then`, which checks whether `v` is positive, and otherwise diverges, is approximated by the following denotation on S_τ

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

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

Example 6. Consider the domain CL_τ in (6). Let $\tau = [\mathtt{out} \mapsto \mathsf{a}, \mathtt{this} \mapsto \mathsf{a}]$. The concrete operation `out:=new(a)` is approximated by the following abstract denotation:

$$
\begin{array}{lll}
c(\mathsf{empty}) = \mathsf{empty} & & \\
c([[], [\mathsf{a}]]) = [[\mathsf{a}], [\mathsf{a}]] & c([[], [\mathsf{b}]]) = [[\mathsf{a}], [\mathsf{b}]] & c([[\mathsf{a}], [\mathsf{a}]]) = [[\mathsf{a}], [\mathsf{a}]] \\
c([[\mathsf{a}], [\mathsf{b}]]) = [[\mathsf{a}], [\mathsf{b}]] & c([[\mathsf{b}], [\mathsf{a}]]) = [[\mathsf{a}], [\mathsf{a}]] & c([[\mathsf{b}], [\mathsf{b}]] = [[\mathsf{a}], [\mathsf{b}]].
\end{array}
$$

The concrete operation `out.f:=this` is approximated by

$$
\begin{array}{lll}
e(\mathsf{empty}) = \mathsf{empty} & & \\
e([[], [\mathsf{a}]]) = \mathsf{empty} & e([[], [\mathsf{b}]]) = \mathsf{empty} & e([[\mathsf{a}], [\mathsf{a}]]) = [[\mathsf{a}], [\mathsf{a}]] \\
e([[\mathsf{a}], [\mathsf{b}]]) = [[\mathsf{a}], [\mathsf{b}]] & e([[\mathsf{b}], [\mathsf{a}]]) = [[\mathsf{b}], [\mathsf{a}]] & e([[\mathsf{b}], [\mathsf{b}]] = [[\mathsf{b}], [\mathsf{b}]].
\end{array}
$$

Note that e returns `empty` when `out` is bound to *nil*, because in such a case writing to the field `out.f` yields an error. Moreover, note that the domain cannot express the fact that the class of `out.f` and that of `this` coincide after the execution of the operation [7].

It is out of the scope of this paper to show that the optimal abstract composition operation is that shown in Figure 3. We just note that, although the composition of concrete denotations is functional, its abstract counterpart $\circ$ in Figure 3 is *not* functional. Example 7 shows that functional composition loses precision even in quite common situations. This is why we use a more complex but more precise operation in Figure 3.

Example 7. Consider a, b as in Example 5. Their functional composition ba is

$$
\begin{array}{lll}
ba(\mathsf{empty}) = \mathsf{empty} & & \\
ba([+,+]) = [+,+] & ba([+,-]) = [+,+] & ba([+,u]) = [+,+] \\
ba([-,+]) = \mathsf{empty} & ba([-,-]) = \mathsf{empty} & ba([-,u]) = \mathsf{empty} \\
ba([u,+]) = [+,u] & ba([u,-]) = [+,u] & ba([u,u]) = [+,u] \ .
\end{array} \tag{10}
$$

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, we can use the information that $ba([+,+]) = [+,+]$. If `v` is negative, we can use the information that $ba([-,+]) = \mathsf{empty}$. 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 3, line (10) is substituted by the more precise approximation

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

This example shows that functional composition cannot prove that `w` is positive in the `then` branch of the command `w` $:=$ `v;  if v` $\geq$ `0 then` (Example 5), while the composition given in Figure 3 can.

5 Logic Programs as Compact Denotations

In Section 2, we have illustrated how logic programs may be used to represent in a compact way the abstract denotations specified in Definition 3. We now define this representation more formally.

Definition 4. *We assume that every element in D_τ is denoted by a ground term. Let D^*_τ denote a set of terms with variables in Vars and such that $\{d^* \in D^*_\tau \mid \mathit{vars}(d^*) = \varnothing\} = D_\tau$.*

We need to generalise to non-ground terms the concept of union-irreducibility and union-reducibility given in Definition 1.

Definition 5. *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 8. Consider the domain S_τ in (1). We have already introduced the ground terms `empty` and $[x_1,\ldots,x_\ell]$ ($x_i \in \{*,+,-,\mathtt{u}\}$) to represent its elements. Let $\mathsf{dom}(\tau) = \{v_1,\ldots,v_\ell\}$. We define

$$S^*_\tau = \{\mathtt{empty}\} \cup \left\{ [x_1,\ldots,x_\ell] \;\middle|\; \begin{array}{l} \text{for } i = 1,\ldots,\ell,\ x_i \in \{*,+,-,\mathtt{u}\} \cup \mathit{Vars} \\ \text{and } x_i = * \text{ iff } \tau(v_i) \neq \mathit{int} \end{array} \right\}. \tag{11}$$

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 $[+,\mathtt{u},\mathtt{X},\mathtt{X}]$.

Example 9. Consider the domain CL_τ in (6). We have already introduced the ground terms `empty` and $[s_1,\ldots,s_\ell]$ ($\#s_i \leq 1$) to represent its elements. Assume that we have a partition $\langle \mathit{Vars}_1, \mathit{Vars}_2 \rangle$ of *Vars* and that every $v \in \mathit{Vars}$ has a *type* $t(v)$ attached. Let $\mathsf{dom}(\tau) = \{v_1,\ldots,v_\ell\}$. We define

$$C^*_\tau = \{\mathtt{empty}\} \cup \left\{ [x_1,\ldots,x_\ell] \;\middle|\; \begin{array}{l} \text{for } i = 1,\ldots,\ell, \\ x_i \in \{[]\} \cup \{\text{lists of subtypes of } \tau(v_i)\} \\ \cup \mathit{Vars}_1 \cup \{[v] \mid v \in \mathit{Vars}_2\} \end{array} \right\}. \tag{12}$$

The variables in Vars_1 are used at the first level of the terms in C^*_τ, while the variables in Vars_2 are used at the second level. Hence they can be distinguished by looking at where they occur. Their type is consistent with the place where they occur. Let for instance $\tau = [\mathtt{out} \mapsto \mathtt{a}, \mathtt{this} \mapsto \mathtt{b}]$, where `a` and `b` are the classes in Figure 2. The set CL^*_τ contains the non-ground union-irreducible terms `[[b],[T]]` and `[O,[T]]`. We have $\mathtt{O} \in \mathit{Vars}_1$ and $\mathtt{T} \in \mathit{Vars}_2$. We have $t(\mathtt{O}) = \mathtt{a}$ and $t(\mathtt{T}) = \mathtt{b}$ because $\tau(\mathtt{out}) = \mathtt{a}$ and $\tau(\mathtt{this}) = \mathtt{b}$. It also contains the non-ground union-reducible terms `[[a,b],[T]]` and `[O,[a,b]]`. Note that we use Prolog lists to represent sets. This does not introduce ambiguity in the implementation since only union-irreducible elements, which are composed of empty or singleton sets (Example 2), are considered as input to the denotations.

A *non-ground representation* consists of a set D^*_τ and a set of substitutions Λ so that D^*_τ is closed *w.r.t.* application of substitutions in Λ and also so that application of any substitution in Λ to a union-irreducible element of D^*_τ preserves its union-irreducibility.

Definition 6. *Let Λ be a set of substitutions with variables in Vars and closed by composition and let Λ^g denote its* grounding subset *on Vars i.e., the set of ground instances of Λ with domain in Vars. Let $\xi^\Lambda(X) = \{X\sigma \mid \sigma \in \Lambda^g\}$ for every syntactic object X (terms, clauses, programs). Let Typenv denote the set of type environments. Then we say that the pair $\langle \{D^*_\tau \mid \tau \in \mathit{Typenv}\}, \Lambda \rangle$ is a* non-ground representation *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 10. Consider the domain S^*_τ in (11). Then $\langle\{S^*_\tau \mid \tau \in \mathit{Typenv}\}, \Lambda\rangle$ is a non-ground representation if

$$\Lambda = \{\sigma \mid \forall v \in \mathit{Vars} : \sigma(v) \in \mathit{Vars} \cup \{+, -, *\}\}$$

i.e., we do not allow variables to be bound to $\mathtt{u}$. This is because Λ must not lose union-irreducibility (point (2) of Definition 6).

Example 11. Consider the domain CL^*_τ in (12). Then $\langle\{\mathit{CL}^*_\tau \mid \tau \in \mathit{Typenv}\}, \Lambda\rangle$ is a non-ground representation if

$$\Lambda = \left\{ \sigma \left| \begin{array}{l} \forall v \in \mathit{Vars}_1 : \sigma(v) \in \mathit{Vars}_1 \cup \{[s] \mid s \in \{\text{subtypes of } t(v)\} \cup \mathit{Vars}_2\} \\ \forall v \in \mathit{Vars}_2 : \sigma(v) \in \mathit{Vars}_2 \cup \{\text{subtypes of } t(v)\} \end{array} \right. \right\}$$

i.e., variables in Vars_1 can be bound to empty or singleton sets of consistent types or to a set made of a variable in Vars_2, and variables in Vars_2 can be bound to a consistent type. Again, this is because the set of substitutions must not lose union-irreducibility (point (2) of Definition 6).

From now on, we write ξ for ξ^Λ and assume that $\langle\{D^*_\tau \mid \tau \in \mathit{Typenv}\}, \Lambda\rangle$ is given.

By using the non-ground representation given in Definition 6, we can write clauses that represent the ui-induced maps more compactly.

Definition 7. *Let $n \geq 1$. A $(\tau_1, \ldots, \tau_n, \tau)$*-compact clause *is $l_1, \ldots, l_n \to r$, with $l_i \in \mathsf{ui}(D^*_{\tau_i})$ for $i = 1, \ldots, n$, $r \in D^*_\tau$ and $\mathit{vars}(r) \subseteq \cup\{\mathit{vars}(l_i) \mid i = 1, \ldots, n\}$. 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 12. If $\tau = [\mathtt{a} \mapsto \mathit{int}, \mathtt{b} \mapsto \mathit{int}, \mathtt{c} \mapsto \mathit{int}, \mathtt{out} \mapsto \mathit{int}]$ then

$$\begin{aligned} c_1 &= (\mathtt{empty} \to [+, +, \mathtt{u}, -]) \\ c_2 &= ([+, -, \mathtt{X}, \mathtt{Y}] \to [\mathtt{u}, \mathtt{Y}, +, +]) \\ c_3 &= ([+, \mathtt{X}, +, +] \to [+, -, \mathtt{X}, \mathtt{u}]) \end{aligned}$$

are (τ, τ)-compact clauses over the domain S^*_τ in (11). The meaning of $\{c_1, c_2, c_3\}$ is the unique ui-induced map $\overline{\{c_1, c_2, c_3\}} : S_\tau \mapsto S_\tau$ such that

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

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 3).

Example 13. If $\tau = [\mathtt{out} \mapsto \mathtt{a}, \mathtt{this} \mapsto \mathtt{a}]$ then

$$\begin{aligned} f_1 &= (\mathtt{empty} \to \mathtt{empty}) \\ f_2 &= ([\mathtt{0}, [\mathtt{T}]] \to [[\mathtt{a}], [\mathtt{T}]]) \end{aligned}$$

are (τ, τ)-compact clauses over the domain CL^*_τ in (12). The meaning of $\{f_1, f_2\}$ is the unique ui-induced map c in Example 6. It respects the conditions in Definition 3. Hence, it is a (τ, τ)-denotation.

As Example 12 shows, the meaning of a set of compact clauses is not necessarily a denotation because it can lack monotonicity (Definition 3). Thus Definition 8 requires monotonicity to guarantee that the meaning of a set of compact clauses is a denotation (Proposition 3). It also requires that sets of compact clauses have exhaustive and non-overlapping inputs, defining a *normal form* which keeps the sets of compact clauses small.

Definition 8. *Let $n, m \geq 1$. The set $\mathbb{CD}_{\tau}^{\tau_1,\ldots,\tau_n}$ 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) $\cup\{\xi(L_i) \mid 1 \leq i \leq m\} = \mathsf{ui}(D_{\tau_1}) \times \cdots \times \mathsf{ui}(D_{\tau_n})$ (exhaustivity),
ii) $\xi(L_j) \cap \xi(L_k) = \varnothing$ for $1 \leq j, k \leq m$, $j \neq k$ (non-overlapping),
iii) $\overline{\{L_1 \to r_1,\ldots,L_m \to r_m\}}$ is monotonic (monotonicity).

Example 14. The set of compact clauses $\{c_1, c_2, c_3\}$ of Example 12 does not satisfy any of the conditions of Definition 8. Instead, the sets of compact clauses (5) in Subsection 2.2 and $\{f_1, f_2\}$ of Example 13 satisfy those conditions. Hence, they are compact denotations.

The following result shows that the elements of $\mathbb{CD}$ are actually denotations.

Proposition 3. *Let $n \geq 1$ and $t \in \mathbb{CD}_{\tau}^{\tau_1,\ldots,\tau_n}$. We have $\overline{t} \in \mathbb{D}_{\tau}^{\tau_1,\ldots,\tau_n}$.*

Proof. By Definition 7, we know that $\overline{t}$ is ui-induced. By point (iii) of Definition 8 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 7).

Proposition 4. *Let $n \geq 1$, $t \in \mathbb{CD}_{\tau}^{\tau_1,\ldots,\tau_n}$ 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 8, there exists an $L' \to r \in t$ such that $L'\theta = L$ for a suitable $\theta \in \Lambda^g$. By Definition 7 and point (iii) of Definition 8, we conclude that

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

If $L \notin \mathsf{ui}(D_{\tau_1}) \times \cdots \times \mathsf{ui}(D_{\tau_n})$, by Proposition 3, Definition 2 and the case shown above, we have

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

where point $*$ follows since $L \notin \mathsf{ui}(D_{\tau_1}) \times \cdots \times \mathsf{ui}(D_{\tau_n})$ while $L''\theta \in \mathsf{ui}(D_{\tau_1}) \times \cdots \times \mathsf{ui}(D_{\tau_n})$ (Definition 7 and point 2 of Definition 6). □

Now that we have a notion of compact denotations $\mathbb{CD}$, we need operations which mimic over $\mathbb{CD}$ what the operations in Figure 3 do over $\mathbb{D}$. For the operations $[op]$, we assume that an explicit compact denotation for every op is given. Note that such compact denotations always exist, since we can use ground representations (Figure 3) as degenerate cases of compact denotations. However, we should try to use logic variables as much as possible, in order to better exploit the capabilities of our non-ground representation.

Predicate	Semantics
domain_entails(τ,A,B) with $\mathtt{A},\mathtt{B} \in D^*_\tau$	computes $\theta_1,\ldots,\theta_n \in \Lambda$ such that $\{\sigma \in \Lambda^g \mid \mathtt{A}\sigma \sqsubseteq \mathtt{B}\sigma\} =$ $= \{\sigma \in \Lambda^g \mid \sigma \leq \theta_i$ for some $1 \leq i \leq n\}$
domain_intersect_ui(τ,A,B) with $\mathtt{A},\mathtt{B} \in \mathsf{ui}(D^*_\tau)$	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 $\xi(\mathtt{A}) \cap \xi(\mathtt{B}) = \cup_{i=1,\ldots,n}\xi(\mathtt{A}\theta_i)$
domain_subtract_ui(τ,A,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 $\cup_{i=1,\ldots,n}\xi(\mathtt{A}\theta_i) = \xi(\mathtt{A}) \setminus \xi(\mathtt{B})$
domain_lub(τ,A,B,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 $\mathtt{L}\theta_i\sigma = \mathtt{A}\theta_i\sigma \sqcup_{D_\tau} \mathtt{B}\theta_i\sigma$
domain_bottom(τ_{in},τ_{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(τ_{in},τ_{out},A,B)	checks if $\xi(\mathtt{A}) = \xi(\mathtt{B})$, with $\mathtt{A},\mathtt{B} \in A^*_{\tau_{in},\tau_{out}}$

Fig. 4. Domain-specific predicates.

Example 15. Consider the operations given in Example 5. A non-degenerate compact denotation for $[op]$ is

$$\mathtt{empty} \to \mathtt{empty},\quad [\mathtt{V},\mathtt{W}] \to [\mathtt{V},\mathtt{V}] \qquad \text{when } op = (\mathtt{w} := \mathtt{v})$$

and

$$\mathtt{empty} \to \mathtt{empty},\quad [+,\mathtt{W}] \to [+,\mathtt{W}],\quad [-,\mathtt{W}] \to \mathtt{empty} \qquad \text{when } op = (\mathtt{if\ v} \geq \mathtt{0\ then}).$$

Example 16. Consider the operations given in Example 6. A non-degenerate compact denotation for $[op]$

$$\mathtt{empty} \to \mathtt{empty},\quad [\mathtt{0},[\mathtt{T}]] \to [[\mathtt{a}],[\mathtt{T}]] \qquad \text{when } op = (\mathtt{out} := \mathtt{new}(\mathtt{a}))$$

and

$$\mathtt{empty} \to \mathtt{empty},\quad [[],[\mathtt{T}]] \to \mathtt{empty},\quad [[\mathtt{0}],[\mathtt{T}]] \to [[\mathtt{0}],[\mathtt{T}]] \qquad \text{when } op = (\mathtt{out.f} := \mathtt{this}).$$

A correct composition operation over $\mathbb{CD}$ is described in the next section.

6 Language Independent Operations

Unlike operations *op* which depend on the programming language for which the framework is instantiated, some operators are language independent and are required for all instances of our framework.

Figure 4 collects the basic predicates that are needed to implement these operations. The predicates domain_entails, domain_intersect_ui, domain_subtract_ui and domain_lub, used to define the composition operation $\circ^*$, are described in Subsections 6.1 and 6.2. The predicate domain_bottom is used for initiating the fixpoint computation and domain_the_same to stop it. This last operation can be implemented by a two-stage test. It first checks for variance of logic programs. If they are not variant of each other, a more expensive model equivalence test is applied.

We now define the operation $\circ^*$ which composes compact denotations and show that it exactly mimics the operation $\circ$ (Figure 3).

6.1 The Case $n = 1$ for $\circ^*$

In this subsection, we specify $\circ^*$ for case when $n = 1$. The general case is discussed in Subsection 6.2. Note that all the examples here are for the domain S^*_τ in (11) for sign analysis. Similar examples can be constructed for the domain CL^*_τ in (12) for class analysis.

The first operation we need is the `domain_entails` operation. A call `domain_entails(`τ`,A,B)` computes all possible ways for making `A` entail `B`.

Example 17. Let $\tau = [\mathtt{a} \mapsto \mathit{int}, \mathtt{b} \mapsto \mathit{int}]$. An implementation of `domain_entails` is such that `domain_entails(`τ`,[X,-],[+,u])` succeeds, binding `X` to `+`; `domain_entails(`τ`,[+,X],[+,u])` succeeds, binding `X` first to `+` and then to `-`; `domain_entails(`τ`,[+,+],[+,u])` succeeds and `domain_entails(`τ`,[+,-],[+,+])` fails. Note that `X` is not allowed to be bound to `u` (Example 10).

We define now a *pre-composition* $\bullet$ which, although it correctly mimics $\circ$, is not closed on $\mathbb{CD}$ (Example 18).

Definition 9. *Let $t_1 \in \mathbb{CD}^{\tau_1}_{\tau'}$ and $t_2 \in \mathbb{CD}^{\tau}_{\tau_1}$ be renamed apart. We define the* compact pre-composition *$t_1 \bullet t_2 \in \mathbb{CD}^{\tau}_{\tau'}$ as*

$$t_1 \bullet t_2 = \left\{ (l_2 \to r_1)\theta \;\middle|\; \begin{array}{l} l_2 \to r_2 \in t_2,\ l_1 \to r_1 \in t_1 \\ \mathtt{domain\_entails}(\tau_1, l_1, r_2) \textit{ computes } \theta \end{array} \right\}.$$

Example 18. Using Definition 9, we will compute

$$\begin{array}{ll} \mathtt{empty} \to \mathtt{empty} & \\ [\mathtt{B},+] \to [-] & [\mathtt{B},-] \to [+] \end{array} \;\bullet\; \begin{array}{ll} \mathtt{empty} \to \mathtt{empty} & \\ [\mathtt{A},+] \to [-,\mathtt{u}] & [\mathtt{A},-] \to [+,+] \end{array}$$
$$=$$
$$\mathtt{empty} \to \mathtt{empty} \quad [\mathtt{A},+] \to [-] \quad [\mathtt{A},+] \to [+] \quad [\mathtt{A},-] \to [-]\ . \tag{13}$$

To see this, 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. Consider next the clause $[\mathtt{A},+] \to [-,\mathtt{u}]$. Its output $[-,\mathtt{u}]$ is entailed by the inputs $[\mathtt{B},+]$ and $[\mathtt{B},-]$ of two clauses on the left, provided we bind the variable $\mathtt{B}$ to $-$. This results in *two* clauses $[\mathtt{A},+] \to [-]$ and $[\mathtt{A},+] \to [+]$. Finally, consider the clause $[\mathtt{A},-] \to [+,+]$, whose output is entailed by the input $[\mathtt{B},+]$ of a clause on the left provided we bind $\mathtt{B}$ to $+$. This results in the clause $[\mathtt{A},-] \to [-]$. Note that (13) is not a compact denotation, since condition (ii) of Definition 8 is not satisfied (there are overlapping inputs).

We prove that $\bullet$ exactly mimics $\circ$ over $\mathbb{CD}$. We first prove this result for the special case when compact denotations are actually ground (Lemma 1), and then for the general case (Proposition 4).

Lemma 1. *Given the hypotheses of Definition 9, if t_1 and t_2 are ground then $\overline{t_1 \bullet t_2} = \overline{t_1} \circ \overline{t_2}$.*

Proof. Let $l \in \mathsf{ui}(D_\tau)$. 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 3.

Since t_1 and t_2 are ground we have

$$\begin{aligned} (\overline{t_1 \bullet t_2})(l) &= \sqcup\{r \mid l \to r \in t_1 \bullet t_2\} \\ &= \bigsqcup \left\{ r \;\middle|\; l \to r \in \left\{ l_2 \to r_1 \;\middle|\; \begin{array}{l} l_2 \to r_2 \in t_2,\ l_1 \to r_1 \in t_1 \\ \mathtt{domain\_entails}(\tau_1, l_1, r_2) \\ \text{succeeds} \end{array} \right\} \right\} \\ &= \bigsqcup \left\{ r_1 \;\middle|\; \begin{array}{l} l \to r_2 \in t_2,\ l_1 \to r_1 \in t_1 \\ \mathtt{domain\_entails}(\tau_1, l_1, r_2) \text{ succeeds} \end{array} \right\}\ . \end{aligned} \tag{14}$$

Since r_2 and l_1 are ground, $\texttt{domain\_entails}(\tau_1, l_1, r_2)$ succeeds if and only if $l_1 \sqsubseteq r_2$. Hence (14) is equal to

$$\begin{aligned}
& \sqcup \{r_1 \mid l \to r_2 \in t_2,\ l_1 \to r_1 \in t_1,\ l_1 \sqsubseteq r_2\} \\
\text{(point (ii) of Definition 8)} &= \sqcup\{r_1 \mid l_1 \to r_1 \in t_1 \text{ and } l_1 \sqsubseteq r_2\} \quad \text{with } l \to r_2 \in t_2 \\
\text{(Proposition 4)} &= \overline{t_1}(r_2) \quad \text{with } l \to r_2 \in t_2 \\
\text{(since } l \in \mathsf{ui}(D_\tau)) &= \overline{t_1}(\overline{t_2}(l)) = (\overline{t_1} \circ \overline{t_2})(l)\ .
\end{aligned}$$

□

Proposition 5. *Given the hypotheses of Definition 9, we have* $\overline{t_1 \bullet t_2} = \overline{t_1} \circ \overline{t_2}$.

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

$$\begin{aligned}
& \overline{t_1 \bullet t_2} \\
\text{(Lemma 2)} &= \overline{\xi(t_1 \bullet t_2)} \\
\text{(Lemma 3)} &= \overline{\xi(t_1) \bullet \xi(t_2)} \\
\text{(Lemma 1)} &= \overline{\xi(t_1)} \circ \overline{\xi(t_2)} \\
\text{(Lemma 2)} &= \overline{t_1} \circ \overline{t_2}\ .
\end{aligned}$$

□

We prove in Proposition 6 that condition (ii) is the only property of Definition 8 which is not preserved by $\bullet$. To this purpose, we need first the following two results. Namely, Lemma 2 says that a non-ground representation and its ground instance have the same meaning. Lemma 3 says that ξ commutes *w.r.t.* $\bullet$.

Lemma 2. *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

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

□

Lemma 3. *Given the hypotheses of Definition 9, we have* $\xi(t_1 \bullet t_2) = \xi(t_1) \bullet \xi(t_2)$.

Proof. The set of clauses $\xi(t_1 \bullet t_2)$ can be rewritten as

$$\begin{aligned}
& \left\{ (l_2 \to r_1)\sigma \,\middle|\, \begin{array}{l} l_2 \to r_2 \in t_2,\ l_1 \to r_1 \in t_1 \\ \texttt{domain\_entails}(\tau_1, l_1, r_2) \text{ computes } \theta \\ \sigma \in \Lambda^g,\ \sigma \leq \theta \end{array} \right\} \\
&= \left\{ (l_2 \to r_1)\sigma \,\middle|\, \begin{array}{l} l_2 \to r_2 \in t_2,\ l_1 \to r_1 \in t_1 \\ l_1\sigma \sqsubseteq r_2\sigma,\ \sigma \in \Lambda^g \end{array} \right\} \\
(*) &= \left\{ l_2\sigma_2 \to r_1\sigma_1 \,\middle|\, \begin{array}{l} l_2 \to r_2 \in t_2,\ l_1 \to r_1 \in t_1 \\ l_1\sigma_1 \sqsubseteq r_2\sigma_2,\ \sigma_1, \sigma_2 \in \Lambda^g \\ \mathsf{dom}(\sigma_1) \cap \mathsf{dom}(\sigma_2) = \varnothing \end{array} \right\} \\
&= \left\{ l_2 \to r_1 \,\middle|\, \begin{array}{l} l_2 \to r_2 \in \xi(t_2),\ l_1 \to r_1 \in \xi(t_1) \\ \texttt{domain\_entails}(\tau_1, l_1, r_2) \text{ succeeds} \end{array} \right\} \\
&= \xi(t_1) \bullet \xi(t_2)\ ,
\end{aligned}$$

where point $*$ holds because t_1 and t_2 are renamed apart. □

Proposition 6. *The operation $\bullet$ of Definition 9 preserves conditions (i) and (iii) of Definition 8, but in general loses condition (ii).*

Proof. We prove that condition (i) is maintained. In the hypotheses of Definition 9 we have

$$\begin{aligned}
&\cup\{\xi(l) \mid l \to r_1 \in t_1 \bullet t_2\}\\
&= \cup\left\{\xi(l_2\theta)\,\middle|\, \begin{array}{l} l_2 \to r_2 \in t_2,\ l_1 \to r_1 \in t_1 \\ \texttt{domain\_entails}(\tau_1, l_1, r_2) \text{ computes } \theta \end{array}\right\}\\
&= \left\{ l_2\sigma \,\middle|\, \begin{array}{l} l_2 \to r_2 \in t_2,\ l_1 \to r_1 \in t_1 \\ \texttt{domain\_entails}(\tau_1, l_1, r_2) \text{ computes } \theta \\ \sigma \in \Lambda^g,\ \sigma \le \theta \end{array}\right\}\\
&= \left\{ l_2\sigma \,\middle|\, \begin{array}{l} l_2 \to r_2 \in t_2,\ l_1 \to r_1 \in t_1 \\ l_1\sigma \sqsubseteq r_2\sigma,\ \sigma \in \Lambda^g \end{array}\right\} .
\end{aligned} \tag{15}$$

Since t_1 satisfies condition (i) of Definition 8, (15) is equal to

$$\{l_2\sigma \mid l_2 \to r_2 \in t_2,\ \sigma \in \Lambda^g\}$$

which is equal to $\mathsf{ui}(D_\tau)$ since also t_2 satisfies condition (i) of Definition 8.

We prove that condition (iii) is maintained. Let $l', l'' \in \mathsf{ui}(D_\tau)$ such that $l' \sqsubseteq l''$. By Lemmas 2 and 3 we have

$$\begin{aligned}
(\overline{t_1 \bullet t_2})(l') &= \overline{\xi(t_1 \bullet t_2)}(l')\\
&= \overline{\xi(t_1) \bullet \xi(t_2)}(l')\\
&= \sqcup\left\{ r_1 \,\middle|\, \begin{array}{l} l' \to r_2 \in \xi(t_2),\ l_1 \to r_1 \in \xi(t_1) \\ \texttt{domain\_entails}(\tau_1, l_1, r_2) \text{ succeeds} \end{array}\right\}\\
&= \sqcup\{r_1 \mid l' \to r_2 \in \xi(t_2),\ l_1 \to r_1 \in \xi(t_1),\ l_1 \sqsubseteq r_2\}\\
(*) &\sqsubseteq \sqcup\{r_1 \mid l'' \to r_2 \in \xi(t_2),\ l_1 \to r_1 \in \xi(t_1),\ l_1 \sqsubseteq r_2\}\\
&= (\overline{t_1 \bullet t_2})(l'') ,
\end{aligned}$$

where point $*$ follows by points (i) and (iii) of Definition 8 applied to t_2.

Example 18 shows that condition (ii) can be lost. □

We can make condition (ii) hold through a *normalisation* procedure which, without changing their meaning, transforms a set of clauses to a set whose inputs are disjoint. It 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_lub` and `domain_subtract_ui` operations specified in Figure 4. The interested reader can find its definition inside our implementation (Section 7).

Example 19. If `make_disjunctive` is applied to the set (13) in Example 18, we only have to compute the $\sqcup$ of the two outputs $[+]$ and $[-]$ for the same input $[\mathtt{A}, +]$. We obtain the compact denotation $\mathtt{empty} \to \mathtt{empty}, [\mathtt{A}, +] \to [\mathtt{u}], [\mathtt{A}, -] \to [-]$.

We can now define the counterpart of $\circ$ over $\mathbb{CD}$. Proposition 5 proves that it is correct and optimal and preserves all three conditions of Definition 8.

Definition 10. *Assuming the hypotheses of Definition 9, we define the abstract composition as* $t_1 \circ^* t_2 = \texttt{make\_disjunctive}(t_1 \bullet t_2)$.

Example 20. Proceeding as in Example 18, we have

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

The input of $[\mathtt{X}] \to [+]$ overlaps with that of both $[+] \to [-]$ and $[-] \to [+]$.

Then `domain_intersect_ui(`τ`,[X],[+])` binds `X` to `+` while `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

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

After the `domain_lub` operation we obtain the compact denotation

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

Since `make_disjunctive` guarantees that also condition (ii) of Definition 8 is satisfied we can conclude that the same holds for $\circ^*$ which, moreover, satisfies all the conditions of Definition 8.

6.2 The General Case of $\circ^*$

In the previous subsection we assumed that $n = 1$. We now show how to remove this restriction and consider the operation $\circ^*$ in its general form. If the clauses in T_i (Figure 3) cover $\mathsf{ui}(D_\tau)$ with the same inputs for every $1 \leq i \leq n$, for every such input l we append the corresponding outputs. We obtain one compact denotation and we can proceed as in Subsection 6.1.

Example 21. Consider the computation of

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

We can merge the two arguments into one and compute

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

which, proceeding as in Subsection 6.1, is

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

If the previous condition does not hold, we must reduce the arguments of $\circ^*$ 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 $\mathsf{ui}(D_\tau)$ with the same inputs. This is achieved by using the procedures `domain_intersect_ui` and `domain_subtract_ui` in Figure 4.

Example 22. Consider the computation of

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

We apply `pave` to the two arguments and obtain

$$\begin{array}{lcl} \texttt{empty},\texttt{empty} \to \texttt{empty} & & \\ \texttt{empty},[\texttt{A}] \to \texttt{empty} & & \texttt{empty} \to \texttt{empty},\texttt{empty} \\ [\texttt{A}],\texttt{empty} \to \texttt{empty} & \circ^* & [+] \to [+],[+] \\ [\texttt{A}],[+] \to [+] & & [-] \to [-],[\texttt{u}] \\ [\texttt{A}],[-] \to [-] & & \end{array}$$

which, proceedings as in Subsection 6.1, is

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

7 Implementation

LOOP (Localised for Object-Oriented Programs) [14] is a generic analyser for our simple object-oriented programs. LOOP is an implementation in Prolog of the watchpoint semantics of [4] extended to deal with object-oriented features through the operations of [5].

The structure of LOOP is given in Figure 5. 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 3. The module `typenv` implements the type environments (Section 2). The module `aux` implements auxiliary and logging functions. External modules must contain the abstract syntax 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 [15] 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* [4]. In general, the cost of the analysis depends on the number of watchpoints $\#w$. If $\#w = 0$, just an input/output analysis of the program is performed.

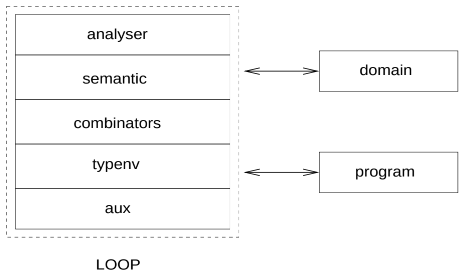

Fig. 5. The structure of LOOP.

analysis	reference	ground	non-ground	bdd's
sign analysis	Equation (1)	`signs`	`signs_vars`	`signs_bdd`
rapid type (*i.e.*, class) analysis	[6, 5]	`rt`	`rt_vars`	
dataflow class analysis	[7, 5]	`df`	`df_vars`	
constraint class analysis	[8, 5]	`ps`	`ps_vars`	
escape analysis	[10]	`e`	`e_vars`	

Fig. 6. The abstract domains implemented for our LOOP analyser.

We have implemented in Prolog five abstract analyses (domains) for LOOP, with both a ground and a non-ground representation. We have also implemented sign analysis by using bdd's [3], normally viewed as the most efficient way for representing abstract denotations. Figure 6 shows those domains. The domain `signs_bdd` is implemented in C++, by using the BuDDy library for

Bmrk	#w	`signs`		`signs_vars`		`signs_bdd` small		and large	
		time	space	time	space	time	space	time	space
`fib`	0	1.683	348291	0.049	7658	0.047	1000	0.026	1000000
`fib`	3	2.186	478132	0.119	15896	0.111	1000	0.064	1000000
`fib`	6	2.234	493412	0.147	20771	0.131	1000	0.091	1000000
`nested`	0	118.732	13453432	0.564	56390	0.758	1000	0.195	1000000
`nested`	7	119.004	13671816	0.859	145752	1.659	1000	0.484	1000000
`nested`	13	120.012	13852568	1.149	230778	3.281	1000	0.916	1000000
`arith`	0	108.578	13617292	0.163	26242	0.364	1000	0.114	1000000
`arith`	7	108.993	13654012	0.181	32238	0.393	1000	0.136	1000000
`arith`	14	109.453	13730012	0.198	38485	0.481	1000	0.168	1000000

Fig. 7. Time and space required for sign analysis.

bdd's [16]. 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.

Bmrk	#w	`rt`		`rt_vars`		`df`		`df_vars`		`ps`		`ps_vars`	
		time	space	time	space	time	space	time	space	time	space	time	space
`inv`	0	0.048	16608	0.016	5556	0.431	94363	0.032	6823	0.460	105601	0.032	7619
`inv`	2	0.051	17946	0.020	6894	0.441	98637	0.034	8077	0.464	110381	0.034	9031
`inv`	4	0.058	18906	0.022	7854	0.444	101745	0.035	8879	0.472	113857	0.035	9933
`clone`	0	0.071	22951	0.042	11563	11.377	2293636	0.070	11527	283.687	29282466	0.162	25307
`clone`	6	0.095	27805	0.055	16417	11.450	2320420	0.078	18935	284.624	29393040	0.165	47855
`clone`	11	0.125	40457	0.079	26650	15.297	3149401	0.296	78574	376.172	39337617	0.495	240844
`figures`	0	2.791	696952	1.106	216494	0.934	166487	0.092	26099	3.574	844796	0.237	53094
`figures`	9	2.881	739802	1.128	259344	0.971	177605	0.106	32497	3.651	891134	0.250	70796
`figures`	17	2.956	776338	1.193	295880	1.071	188949	0.132	37938	3.717	936542	0.255	85612

Fig. 8. Time and space required for class analysis.

For sign analysis, we use three benchmarks: `fib`, the Fibonacci procedure; `nested`, the program in Figure 1; and `arith` which implements some arbitrary precision arithmetic operations. For class and escape analysis we use three benchmarks: `inv`, a procedure that changes from `a` to `b` and back the class of a variable. It does it twice through two calls, inside a `while` loop, to a virtual function; `clone` (Figure 2), 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 geometric figures with a generic method for rotating them.

Figure 7 compares time (in seconds) and space (amount of data structures built during the analysis) costs of the `signs` and `signs_vars` analyses. It can be seen that the analysis scales with the number of watchpoints. Figures 8 and 9 do the same for the three domains for class analysis and the domain for escape analysis summarised in Figure 6, respectively. These show that `X_vars` is always more efficient than `X` and gains up to three orders of magnitude in time and space (for instance, the case of the analyses `ps` and `ps_vars` for the benchmark `clone`). The cost in space of the analyses *i.e.,* the number of data structures built, is independent of the choice of the language used to implement LOOP, indicating that their time cost is also probably unrelated to this choice.

Figure 7 also compares our compact representation of the domain for sign analysis in (1) with the other implementation we have built through a highly-optimised library for bdd's [16]. We have run the test with two different initial numbers of bdd nodes and cache size. These are suggested by the authors of [16] for "small" and "large" examples, respectively. Figure 7 shows that a compact representation through logic programs has an efficiency comparable with that of an implementation based

Bmrk	#w	`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

Fig. 9. Escape analysis.

on bdd's and consumes much less memory space. The more efficient results in the column headed *large* were obtained by considering the sign analysis of programs no 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. W.r.t. space, Figure 7 reports the number of Prolog atoms, for `signs_vars`, and of bdd nodes, for `signs_bdd`. Those units can be considered the same, since they are related by a small constant.

8 Related Work and Conclusion

Non-ground logic programs have been used for the static analysis of both functional and logic programs. In the case of ML, they have been used to model type dependencies [17, 18], but only a single clause (a single type dependency) was used. In the case of logic programs they have been used for type analysis [19–21]. Non-ground terms have been used also for mode and structure analysis of logic programs [22]. The application of non-ground terms or clauses to the abstract interpretation of other programming paradigms has not before been studied. In [23], non-ground constraints have been used for set-based static analysis, which is not based on abstract interpretation. There, dependencies between variables cannot be expressed. Moreover, since constraints have a *flat* structure, the (abstract) values of variables in different instantiations of the same procedure or in different program points are indistinguishable.

This paper has introduced the notion of union-reducibility (Definition 1), which is new. In [12, 13], a notion of (component-)additivity is used to limit the number of iterations to obtain a fixpoint. Union-irreducibility, as defined here, might force the analyser to consider more inputs in a denotation than additivity, but allows one to represent more precisely non-additive denotations.

Compared to bdd's [3], there are several advantages in using our technique.

- Logic programs are closer to the specification since denotations are immediately recognisable as logic programs (*i.e.,* sets of input/output arrows), while a bdd uses a formula to code the denotation.
- The symbolic capability of logic programs helps the programming task. For instance, we do not need to *code* sets of classes or escape contexts through binary digits, but we can use logic programming lists.
- Our technique just requires the unification procedure and not the whole machinery of logic programming (resolution, backtracking). 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 7). This makes our technique particularly attractive for applications where memory is limited. This is a particularly important requirement for smart cards if they are to use analysers to protect themselves against security violations in the downloaded software.

References

1. 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.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles Techniques and Tools.* Addison Wesley Publishing Company, 1986.
3. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
4. F. Spoto. Watchpoint Semantics: A Tool for Compositional and Focussed Static Analyses. In P. Cousot, editor, *Proc. of SAS'01*, Lecture Notes in Computer Science, pages 127–145. Springer-Verlag, 2001.
5. T. Jensen and F. Spoto. Class Analysis of Object-Oriented Programs through Abstract Interpretation. In F. Honsell and M. Miculan, editors, *Proceedings of FOSSACS 2001*, volume 2030 of *Lecture Notes in Computer Science*, pages 261–275. Springer-Verlag, 2001. Extended version to appear in ACM TOPLAS.

6. 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.
7. 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.
8. J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In *Proc. OOPSLA'91*, volume 26(11) of *ACM SIGPLAN Notices*, pages 146–161, 1991.
9. B. Blanchet. Escape Analysis for Object Oriented Languages. Application to Java^{TM}. In *OOPSLA'99*, volume 34(10) of *SIGPLAN Notices*, pages 20–34, 1999.
10. P. M. Hill and F. Spoto. A Foundation of Escape Analysis. In H. Kirchner and C. Ringeissen, editors, *Proc. of AMAST'02*, volume 2422 of *Lecture Notes in Computer Science*, pages 380–395, St. Gilles les Bains, La Réunion island, France, 2002. Springer-Verlag.
11. P. M. Hill and F. Spoto. Logic Programs as Compact Denotations. In G. Gupta, editor, *Proc. of Practical Applications of Declarative Programming*, volume 2562 of *Lecture Notes in Computer Science*, pages 339–356, New Orleans, Louisiana, USA, 2003. 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, 1999. Springer-Verlag.
13. H. R. Nielson and F. Nielson. Bounded Fixed Point Iteration. In *Proc. of POPL'92*, pages 71–82. ACM Press, 1992.
14. F. Spoto. The LOOP Analyser. `www.sci.univr.it/∼spoto/loop/`.
15. 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.
16. J. Lind-Nielsen. BuDDy - A Binary Decision Diagram Package. `www.itu.dk/research/buddy/`.
17. R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.*, 146:29–60, 1969.
18. R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and Systems Sciences*, 17-3:348–375, 1978.
19. M. Codish and B. Demoen. Deriving Polymorphic Type Dependencies for Logic Programs Using Multiple Incarnations of Prop. In B. Le Charlier, editor, *Proc. SAS*, volume 864 of *Lecture Notes in Computer Science*, pages 281–296. Springer-Verlag, 1994.
20. 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.
21. G. Levi and F. Spoto. An Experiment in Domain Refinement: Type Domains and Type Representations for Logic Programs. In *Principles of Declarative Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 152–169. Springer-Verlag, 1998.
22. J. Gallagher, D. Boulanger, and H. Saglam. Practical Model-Based Static Analysis for Definite Logic Programs. In J. W. Lloyd, editor, *Proc. of the Int. Logic Programming Symp., ILPS'95*, pages 351–365, Portland, Oregon, 1995. MIT Press.
23. N. Heintze and J. Jaffar. Set Constraints and Set-Based Analysis. In A. Borning, editor, *Proc. of Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*, pages 281–298. Springer-Verlag, 1994.

Vitae

Patricia M. Hill has a BSc from the University of Bristol, UK in Mathematics and a PhD from the Open University, Milton Keynes, UK. She is a Senior Research Fellow in the School of Computing of the University of Leeds where she leads the research group working on Logic Programming and Program Analysis. Prior to this, she was a Research Fellow at the University of Bristol from 1986 to 1990.

Dr Hill's main research interests are related to the area of logic programs and static analysis. She has published more than twelve journal articles; is a co-author of the book *The Gödel Programming Language*, MIT Press, 1994; and an author of two book chapters "A Semantics for Typed Logic Programs" in *Types in Logic Programming*, MIT Press, 1992 and "Meta-Programming in Logic Programming" in *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume V: Logic Programming*, Oxford University Press, 1996.

Fausto Spoto has a *Laurea* (BSc) and a PhD in Computer Science from the Università di Pisa, Italy. He is a permanent Research Fellow in the Dipartimento di Informatica of the Università di Verona, Italy.

His main interests are related to the static analysis of programming languages, to the compositional definition of those analyses and their application to real cases of programming languages. In particular, to the object-oriented programming paradigm. He is also interested in the formal definition of analyses which have up to now been defined in an informal or imprecise way, in order to prove their correctness and compare them on a formal basis. He has published five journal articles and more than fifteen conference papers.