

Manipulating Abstract Domains  
and Abstract Operations  
for Static Analysis of (Declarative) Programs

**Tino Cortesi**

Università di Venezia, Ca' Foscari

# Abstract Interpretation

---

The key-point in Abstract Interpretation is the definition of a domain of abstract properties.

A domain is a mathematical object, where the properties of interest about program execution are related with each other with respect to their relative degree of precision.

One of the main feature of Abstract Interpretation is that most properties of the resulting approximate semantics (precision, completeness, modularity, scalability) depend upon the choice of

~> a concrete semantics

~> an abstract domain of properties.

~> abstract operations that safely approximate the corresponding concrete operations

## Abstract Domains

---

Abstract domains may represent, for instance

- ↪ properties that enhance the optimization of the analysed programs
- ↪ properties required by program transformation techniques
- ↪ properties for program certification (security and secrecy)

## Overview

---

We are interested in general techniques to manipulate abstract domains:

↪ Domain Construction

↪ Domain Quality (optimality and completeness)

↪ Domain Comparison

↪ Domain Combination (product operations)

↪ Domain Decomposition (quotient and complement operations)

## Case Studies

---

We consider, as case-study, some domains for analysis of Logic Programs, namely

↪ Pos and Def for groundness analysis

A set  $\Sigma$  of substitutions **grounds** a program variable  $x$  if each substitution in  $\Sigma$  instantiates  $x$  to a term containing no variables.

↪ Sharing for sharing and aliasing analysis

Program variables  $x$  and  $y$  **may share** in a set  $\Sigma$  of substitutions if there exists at least one substitution in  $\Sigma$  that instantiates  $x$  and  $y$  to terms containing a common variable.

## References

---

→ A. Cortesi, G. File' and W. Winsborough, **Optimal Groundness Analysis Using Propositional Logic**, The Journal of Logic Programming, vol. 27(2), pp.137-167. (1996)

A. Cortesi, G. File', R. Giacobazzi, C. Palamidessi, and F. Ranzato, **Complementation in Abstract Interpretation**, ACM Transactions on Programming Languages and Systems, vol.19(1), pp. 7-47. (1997).

A. Cortesi, G. File', and W. Winsborough, **The Quotient of an Abstract Interpretation**, Theoretical Computer Science, vol. 202(1-2), pp.163–192 (1998).

A. Cortesi, and G. File', **Sharing is Optimal**, The Journal of Logic Programming Vol 38(3), pp. 371–386 (1999).

A. Cortesi, B. Le Charlier, and P. Van Hentenryck, **Combinations of Abstract Domains for Logic Programming: Open Product and Generic Pattern Construction**, Science of Computer Programming, Vol 38(1–3), pp. 27-71 (2000).

## Part I

# Abstract Domains for Analysis of (Constraint) Logic Programs

## A motivating example

---

<u>query</u>		<u>program</u>
?–	$\langle 0 \rangle p(u, v, w),$	$p(z_1, z_1, z_2).$
	$\langle 1 \rangle q(u),$	$p(z_1, z_2, z_1).$
	$\langle 2 \rangle v = x,$	$q(c).$
	$\langle 3 \rangle w = x,$	$use(x) :- \dots$
	$\langle 4 \rangle use(x).$	

Assume that at point  $\langle 0 \rangle$  the activation set is the empty substitution.

The concrete unification of  $p(u, v, w) = p(z_1, z_1, z_2)$  produces

$\{\{u \mapsto v, z_1 \mapsto v, z_2 \mapsto w\}\}$ .

The result of using the second clause for  $p$  is similar to that of the first, and gets

$\{\{u \mapsto w, z_1 \mapsto w, z_2 \mapsto v\}\}$ .

Results from the two clauses are combined by using the union operation.

<u>query</u>		<u>program</u>
?-	$\langle 0 \rangle p(u, v, w),$	$p(z_1, z_1, z_2).$
	$\langle 1 \rangle q(u),$	$p(z_1, z_2, z_1).$
	$\langle 2 \rangle v = x,$	$q(c).$
	$\langle 3 \rangle w = x,$	$use(x) :- \dots$
	$\langle 4 \rangle use(x).$	

$$\langle 1 \rangle \left\{ \{u \mapsto v, z_1 \mapsto v, z_2 \mapsto w\}, \{u \mapsto w, z_1 \mapsto w, z_2 \mapsto v\} \right\}.$$

$$\langle 2 \rangle \left\{ \{u \mapsto c, v \mapsto c, z_1 \mapsto c, z_2 \mapsto w\}, \{u \mapsto c, w \mapsto c, z_1 \mapsto c, z_2 \mapsto v\} \right\}$$

$$\langle 3 \rangle \left\{ \begin{array}{l} \{u \mapsto c, v \mapsto c, x \mapsto c, z_1 \mapsto c, z_2 \mapsto w\}, \\ \{u \mapsto c, w \mapsto c, v \mapsto x, z_1 \mapsto c, z_2 \mapsto x\} \end{array} \right\}$$

$$\langle 4 \rangle \left\{ \{u \mapsto c, v \mapsto c, x \mapsto c, w \mapsto c, z_1 \mapsto c, z_2 \mapsto c\} \right\}$$

Therefore at point  $\langle 4 \rangle$  each variable is ground. In particular the argument of  $use$  is a ground term.

## A Basic domain for Groundness

---

One candidate abstract domain for representing groundness is  $GR = \wp(Var)$ , where  $Var$  is the set of program variables.

$GR$  consists of sets  $V$  of variables, where  $V$  represents the set of substitutions that ground each variable in  $V$ .

However, such a domain fails to capture the propagation of groundness among program variables bound to terms that contain the same free variables.

## A Basic domain for Groundness

---

In our example,

<u>query</u>		<u>program</u>
?–	$\langle 0 \rangle p(u, v, w),$	$p(z_1, z_1, z_2).$
	$\langle 1 \rangle q(u),$	$p(z_1, z_2, z_1).$
	$\langle 2 \rangle v = x,$	$q(c).$
	$\langle 3 \rangle w = x,$	$use(x) :- \dots$
	$\langle 4 \rangle use(x).$	

At point  $\langle 0 \rangle$  the activation set is  $\emptyset$ .

At point  $\langle 1 \rangle$  is still  $\emptyset$  (no groundness information)

At point  $\langle 2 \rangle$  is  $\{u\}$  ( $u$  is ground)

At point  $\langle 3 \rangle$  and at point  $\langle 4 \rangle$  is still just  $\{u\}$ .

## The domain Pos

---

Pos is composed of propositional formulas that are true when all their variables are set to true (positive).

Intuitively,

$\rightsquigarrow$  the formula  $x \wedge y$  says that  $x$  and  $y$  are both ground;

$\rightsquigarrow$  the formula  $x \vee y$  says that either  $x$  or  $y$  is ground, or both;

$\rightsquigarrow$  the formula  $x \wedge y \wedge z \leftrightarrow u \wedge w$  expresses the equivalence of  $\{x, y, z\}$  and  $\{u, w\}$ .

It approximates, for instance, the substitution

$\{u \mapsto f(x, y), w \mapsto f(y, z)\}$ .

## The example revisited with Pos

<u>query</u>		<u>program</u>
?—	$\langle 0 \rangle p(u, v, w),$	$p(z_1, z_1, z_2).$
	$\langle 1 \rangle q(u),$	$p(z_1, z_2, z_1).$
	$\langle 2 \rangle v = x,$	$q(c).$
	$\langle 3 \rangle w = x,$	$use(x) :- \dots$
	$\langle 4 \rangle use(x).$	

At point  $\langle 0 \rangle$  the abstract activation is T (where T stands for true).

Abstract unification of  $p(u, v, w) = p(z_1, z_1, z_2)$  produces  
 $d_1 = (u \leftrightarrow z_1) \wedge (v \leftrightarrow z_1) \wedge (w \leftrightarrow z_2).$

Abstract unification of  $p(u, v, w) = p(z_1, z_2, z_2)$  produces  
 $d_2 = (u \leftrightarrow z_1) \wedge (v \leftrightarrow z_2) \wedge (w \leftrightarrow z_2).$

## The example revisited with Pos

<u>query</u>		<u>program</u>
?–	$\langle 0 \rangle p(u, v, w),$	$p(z_1, z_1, z_2).$
	$\langle 1 \rangle q(u),$	$p(z_1, z_2, z_1).$
	$\langle 2 \rangle v = x,$	$q(c).$
	$\langle 3 \rangle w = x,$	$use(x) :- \dots$
	$\langle 4 \rangle use(x).$	

As part of computing the abstract activation at point  $\langle 1 \rangle$ ,  $d_1$  and  $d_2$  are projected onto the variable set  $\{u, v, w, x\}$  by means of existential quantification:

$$\exists\{z_1, z_2\}.(u \leftrightarrow z_1) \wedge (v \leftrightarrow z_1) \wedge (w \leftrightarrow z_2) \equiv u \leftrightarrow v$$

$$\exists\{z_1, z_2\}.(u \leftrightarrow z_1) \wedge (v \leftrightarrow z_2) \wedge (w \leftrightarrow z_2) \equiv v \leftrightarrow w$$

Results from the two clauses are combined by using the  $\vee$  logical operator. Thus abstract activation at point  $\langle 1 \rangle$  is  $(u \leftrightarrow v) \vee (u \leftrightarrow w)$ .

## The example revisited with Pos

<u>query</u>		<u>program</u>
?–	$\langle 0 \rangle p(u, v, w),$	$p(z_1, z_1, z_2).$
	$\langle 1 \rangle q(u),$	$p(z_1, z_2, z_1).$
	$\langle 2 \rangle v = x,$	$q(c).$
	$\langle 3 \rangle w = x,$	$use(x) :- \dots$
	$\langle 4 \rangle use(x).$	

When a new unification is simulated, like  $q(u) = q(c)$ , the abstraction of the *mgu* is conjoined with the previous abstract context. In this case, the concrete *mgu* is  $\{u \mapsto c\}$ , which is just abstracted by  $u$ .

At program point  $\langle 2 \rangle$  the abstract activation is

$$((u \leftrightarrow v) \vee (u \leftrightarrow w)) \wedge u \equiv u \wedge (v \vee w)$$

## The example revisited with Pos

<u>query</u>		<u>program</u>
?-	$\langle 0 \rangle p(u, v, w),$	$p(z_1, z_1, z_2).$
	$\langle 1 \rangle q(u),$	$p(z_1, z_2, z_1).$
	$\langle 2 \rangle v = x,$	$q(c).$
	$\langle 3 \rangle w = x,$	$use(x) :- \dots$
	$\langle 4 \rangle use(x).$	

We obtain at point  $\langle 3 \rangle$

$$\begin{aligned}
 & (u \wedge (v \vee w)) \wedge (v \leftrightarrow x) \\
 \equiv & u \wedge ((v \wedge (v \leftrightarrow x)) \vee (w \wedge (v \leftrightarrow x))) \\
 \equiv & u \wedge ((v \wedge x) \vee (w \wedge (v \leftrightarrow x)))
 \end{aligned}$$

and at program point  $\langle 4 \rangle$

$$u \wedge v \wedge w \wedge x$$

which tells us that each variable is ground.

## Representing substitutions by formulas

---

The main intuition: each substitution defines a truth assignment, that assigns the value *true* to each variable that the substitution grounds:

$$\text{assign} : \text{Subst} \longrightarrow \mathbf{V} \longrightarrow \{\text{true}, \text{false}\}$$

$$\text{assign } \sigma \ x = \text{true} \text{ iff } \sigma \text{ grounds } x$$

### Example

Let  $h, g$  and  $a, c$  denote function and constant symbols.

Consider the formula  $f = x_1 \wedge (x_2 \leftrightarrow x_3)$

and the substitution  $\sigma = \{x_1 \mapsto g(a), x_2 \mapsto h(x_5), x_3 \mapsto (g(x_6))\}$ . By the definition of *assign* we have:

$$\text{assign } \sigma = \langle x_1/\mathbf{T}, x_2/\mathbf{F}, x_3/\mathbf{F}, x_4/\mathbf{F}, x_5/\mathbf{F}, x_6/\mathbf{F} \rangle \text{ satisfies } f.$$

## Concretization Function

---

A substitution's groundness and variable equivalence properties are preserved under instantiation: if  $\sigma$  grounds  $x$  then any  $\sigma' \trianglelefteq \sigma$  grounds  $x$ .

Thus, to represent the groundness and equivalence of a substitution  $\sigma$ , it is natural that the formulas that represent these properties approximate only instantiation closed sets of substitutions.

These observations motivate the following definition of the concretization function

$\gamma$ :

$$\gamma : \text{Pos} \longrightarrow \wp(\text{Subst}),$$

$$\gamma(f) = \{\sigma \in \text{Subst} : \forall \sigma' \trianglelefteq \sigma . \text{assign } \sigma' \models f\}.$$

## The ordering on Pos

---

The domain Pos is a partial order.

- `true` is the top element (means: no info)
- `false` is the bottom element (means: failure)
- for the other elements (on the same set of variables)  $f_1 \sqsubseteq f_2$  if and only if  $f_1 \models f_2$ .

### Example

Let  $h, g$  and  $a, c$  denote function and constant symbols.

Consider the formulas:

$$f_1 = x_1 \wedge (x_2 \leftrightarrow x_3)$$

$$f_2 = (x_2 \leftrightarrow x_3)$$

$f_1 \sqsubseteq f_2$ , and this corresponds to the fact that the set of substitutions represented by  $f_1$  is a (proper) subset of the set of substitutions represented by  $f_2$

## Abstraction Function

---

The abstraction function, i.e. the function that maps a concrete element (a set of substitution) in its best approximation in  $\text{Pos}$  is defined as the adjoint of  $\gamma$ , i.e.,

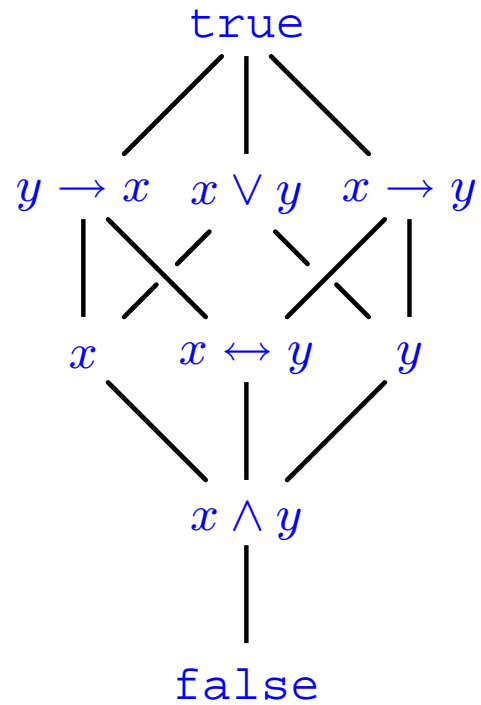
$$\alpha(\Sigma) = \bigwedge \{d \in \text{Pos}, \mid \gamma(d) \subseteq \Sigma\}.$$

As the concretization function is injective we get:

$(\text{Pos}, \gamma, \wp(\text{Subst}), \alpha)$  is a Galois Insertion.

## The lattice Pos

---



This is the lattice of Pos when only variables  $x$  and  $y$  are considered.

- `true` represents all the substitutions with domain  $\{x, y\}$
- `false` represents an empty set of substitutions.

## Abstract Operations

---

↪ The abstract **union** is defined by the logical join operator  $\vee$ .

↪ The abstract **projection**  $\pi$  amounts to existentially quantifying a formula

The existential quantification of a propositional formula obeys

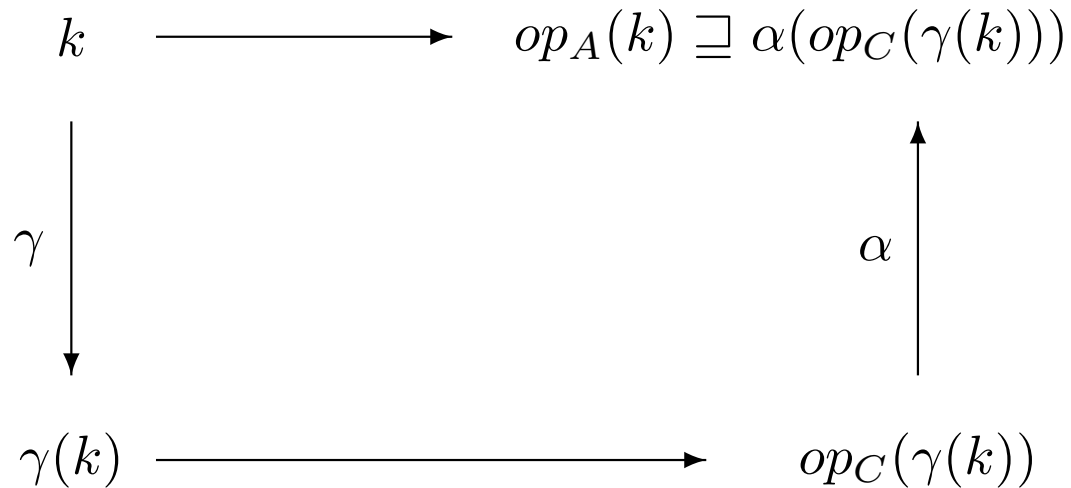
$$\exists x.f \equiv f\langle x/\mathbf{T} \rangle \vee f\langle x/\mathbf{F} \rangle$$

↪ The abstract **unification** is defined by logical conjunction.

$$\mathbf{u}(f, x = t) = f \wedge (x \leftrightarrow \bigwedge (var(t)))$$

## Correctness

---



An abstract operation  $op_A$  is **correct** wrt a concrete operation  $op_C$  if for every abstract object  $k$

$$\alpha(op_C(\gamma(k))) \sqsubseteq op_A(k)$$

# Optimality

---

$$\begin{array}{ccc}
 k & \longrightarrow & op_A(k) = \alpha(op_C(\gamma(k))) \\
 \downarrow \gamma & & \uparrow \alpha \\
 \gamma(k) & \longrightarrow & op_C(\gamma(k))
 \end{array}$$

An abstract operation  $op_A$  is **optimal** wrt a concrete operation  $op_C$  if for every abstract object  $k$

$$\alpha(op_C(\gamma(k))) = op_A(k)$$

## Strong Optimality ( $\alpha$ )

---

$$\begin{array}{ccc}
 \alpha(c) & \longrightarrow & op_A(\alpha(c)) = \alpha(op_C(c)) \\
 \uparrow \alpha & & \uparrow \alpha \\
 c & \longrightarrow & op_C(c)
 \end{array}$$

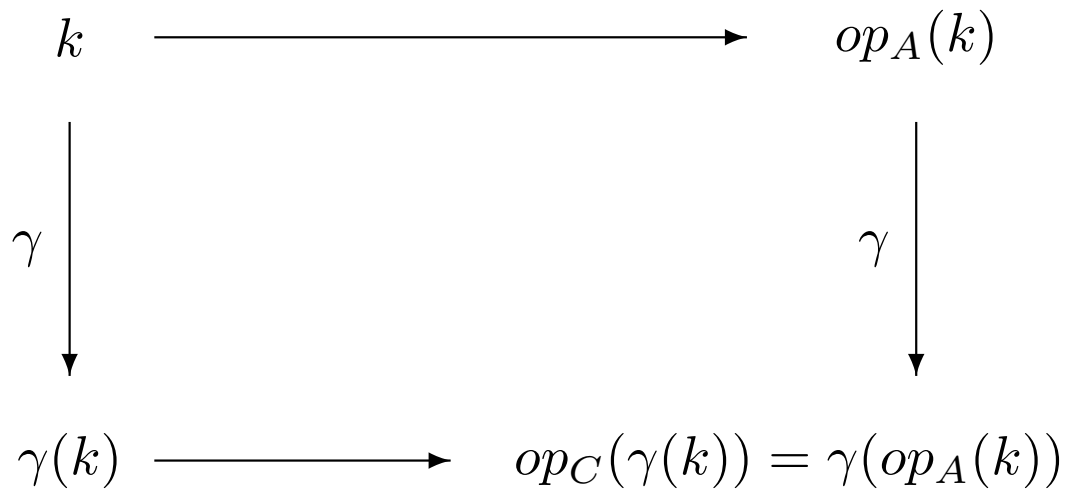
$op_A$  is  $\alpha$ -optimal (or complete) if for every concrete object  $c$

$$\alpha(op_C(c)) = op_A(\alpha(c))$$

The notion of  $\alpha$ -optimality says that when simulating a specific concrete computation through the corresponding abstract one, the only possible loss of information is due to the domain abstraction.

## Strong Optimality ( $\gamma$ )

---



$op_A$  is  $\gamma$ -optimal if for every abstract object  $k$ :  $op_C(\gamma(k)) = \gamma(op_A(k))$ .

If you look at the abstract interpretation as source programs with their meaning and at the concrete interpretation as object programs with their meaning, then the notion of  $\gamma$ -optimality resembles the "diagram commuting" condition for compiler correctness.

## Correctness and Optimalities in Pos

---

- ↪  $\alpha$  and  $\gamma$ -optimality imply standard optimality.
- ↪ abstract union, projection, and unification in Pos are correct (as expected!)
- ↪ even for an abstract domain as powerful as Pos,  $\gamma$ -optimality is too strong to be met.
- ↪ abstract union and projection operation in Pos are  $\alpha$ -optimal
- ↪ abstract unification is just optimal.

The intuitive reason for the negative results on abstract unification is that in the concrete unification, the substitutions produced depend on the particular unification performed, and Pos cannot take care, for instance, of some unification failures.

## Sharing Analysis

---

Program variables  $x$  and  $y$  **may share** in a set  $\Sigma$  of substitutions if there exists at least one substitution in  $\Sigma$  that instantiates  $x$  and  $y$  to terms containing a common variable.

Sharing information is useful for supporting, for instance, AND-parallelism, and program transformations.

One candidate abstract domain for representing groundness is PS (pair sharing), defined as  $PS = \wp(Var \times Var)$ , where  $Var$  is the set of program variables.

An element of PS is a set of pairs  $(x_i, x_j)$ . It says that  $x_i$  and  $x_j$  may share a common variable.

## Example of sharing analysis

---

<u>query</u>		<u>program</u>
?—	$\langle 0 \rangle p(u, v, w),$	$p(z_1, z_1, z_2).$
	$\langle 1 \rangle q(u),$	$p(z_1, z_2, z_1).$
	$\langle 2 \rangle v = x,$	$q(c).$
	$\langle 3 \rangle w = x,$	$use(x) :- \dots$
	$\langle 4 \rangle use(x).$	

At point  $\langle 0 \rangle$  the activation set is  $\emptyset$ .

Abstract unification of  $p(u, v, w) = p(z_1, z_1, z_2)$  produces  $\{(u, z_1), (v, z_1), (u, v), (w, z_2)\}$ , that projected onto the variable set  $\{u, v, w, x\}$  produces  $\{(u, v)\}$ .

The same way, abstract unification of  $p(u, v, w) = p(z_1, z_2, z_1)$  produces  $\{(u, w)\}$ .

Results from the two clauses are combined through a transitive closure of the union. Thus abstract activation at point  $\langle 1 \rangle$  is  $\{(u, v), (u, w), (v, w)\}$ .

<u>query</u>		<u>program</u>
?–	$\langle 0 \rangle p(u, v, w),$	$p(z_1, z_1, z_2).$
	$\langle 1 \rangle q(u),$	$p(z_1, z_2, z_1).$
	$\langle 2 \rangle v = x,$	$q(c).$
	$\langle 3 \rangle w = x,$	$use(x) :- \dots$
	$\langle 4 \rangle use(x).$	

At program point  $\langle 1 \rangle$  the activation set is  $\{(u, v), (u, w), (v, w)\}$ .

At program point  $\langle 2 \rangle$ , as  $u$  is instantiated to a ground term, all pairs containing  $u$  are removed. Thus, we get  $\{(v, w)\}$ .

At program point  $\langle 3 \rangle$  and  $\langle 4 \rangle$  the activation set is  $\{(v, w), (v, x), (w, x)\}$ . This says that when  $use$  is called, program terms variables  $v, x$  and  $w$  are instantiated to terms that *may* share a variable (not necessarily the same).

## Set Sharing

---

Instead of pairs of variables, collect sets of program variables that are instantiated to terms that may share *the same* variable

$$\text{Sharing} = \{A \subseteq \wp(\text{Var}), A \neq \emptyset \Rightarrow \emptyset \in A\}.$$

Given an element  $S \in \text{Sharing}$ , any variable in  $\text{Var}$  that does not appear in any set of  $S$  is ground in any substitution represented by  $S$ .

### Example

Let  $\text{Var} = \{x, y, z, w\}$ , and consider  $\{\emptyset, \{w\}, \{y, z, w\}\} \in \text{Sharing}$ .

This set represents the substitutions under which  $x$  is ground and  $y, z,$  and  $w$  may share. In particular,  $\sigma_1 = \{x \mapsto a, y \mapsto b, z \mapsto c\}$  and

$\sigma_2 = \{x \mapsto b, y \mapsto v, z \mapsto v, w \mapsto v\}$  (where  $a, b, c$  are constant symbols and  $v$  is a free variable) satisfy these properties.

## Concretization and Abstraction Functions

---

For  $y \in Var$  and  $\sigma \in Subst$ , let  $share(\sigma, y)$  be the set of variables of interest whose images under  $\sigma$  contain the variable  $y$

$$share(\sigma, y) = \{x \in Var \mid y \in var(\sigma(x))\}.$$

For instance, if  $\sigma = \{x_1 \mapsto y_1, x_2 \mapsto g(y_1, y_2)\}$ , then  $share(\sigma, y_1) = \{x_1, x_2\}$ , and  $share(\sigma, y_2) = \{x_2\}$ .

For  $\Sigma \in \wp(Subst)$  and  $S \in Sharing$

$$\alpha(\Sigma) = \{share(\sigma, y) \mid \sigma \in \Sigma, y \in Var\},$$

$$\gamma(S) = \{\sigma \in Subst \mid \alpha(\{\sigma\}) \subseteq S\}.$$

## Injectivity of $\gamma$

---

For any  $S \in \text{Sharing}$  there exists a substitution  $\sigma$  such that  $\alpha(\{\sigma\}) = S$ .

### Example

Let  $U = \{x_1, x_2, x_3, x_4\}$  be the set of variables of interest.

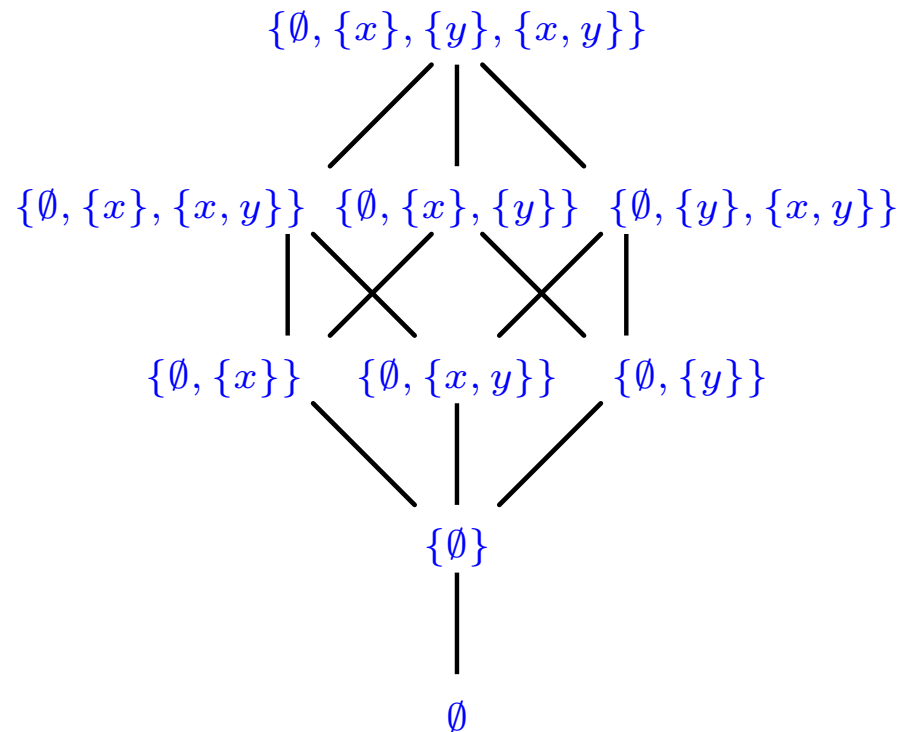
Consider the element  $S = \{\emptyset, \{x_1, x_2, x_3\}, \{x_2, x_3\}, \{x_1, x_3\}, \{x_3\}\}$ .

The substitution  $\sigma$  is based on the following correspondence between elements in  $S$  and variables in  $Var \setminus U$ .

$$\underbrace{\{x_1, x_2, x_3\}}_{y_1}, \underbrace{\{x_2, x_3\}}_{y_2}, \underbrace{\{x_1, x_3\}}_{y_3}, \underbrace{\{x_3\}}_{y_4}.$$

$$\sigma = \left\{ \begin{array}{ll} x_1 \mapsto f(y_1, y_3), & x_2 \mapsto f(y_1, y_2), \\ x_3 \mapsto f(y_1, y_2, y_3, y_4), & x_4 \mapsto a \end{array} \right\}$$

## The Lattice of Sharing



This is the lattice of Sharing when only variables  $x$  and  $y$  are considered.

- $\{\emptyset, \{x\}, \{y\}, \{x, y\}\}$  represents all the substitutions.
- $\{\emptyset\}$  represent the substitutions with domain  $\{x, y\}$  that ground both  $x$  and  $y$ ; whereas  $\emptyset$  represents the empty set of substitutions.

## Abstract Operations

---

- ↪ The abstract **union** is defined by set union  $\cup$ .
- ↪ The abstract **projection**  $\pi$  is defined in terms of intersection  $\cap$ .
- ↪ The abstract **unification** is defined through a closure operation.
  - The *closure under union* of  $S \in \text{Sharing}$  denoted  $S^*$ , is the smallest superset of  $S$  satisfying  $A \in S^* \wedge A' \in S^* \rightarrow (A \cup A') \in S^*$ .
  - The part of  $S$  that is *relevant* to a term  $t$ , denoted  $rel(S, t)$ , is the set  $\{A \in S \mid \text{Var}(t) \cap A \neq \emptyset\}$ .
  - If  $S, Z \in \text{Sharing}$ , the *cross product*  $S \otimes Z$  is  $\{(A \cup B) \mid A \in S, B \in Z\}$ .
  - The abstract unification (basic case) is

$$(S, x = t) \mapsto (S - (rel(x, S) \cup rel(t, S))) \cup (rel(x, B) \otimes rel(t, B))^*$$

## Unification on Sharing: Example

<u>query</u>	<u>program</u>
?–	$\langle 0 \rangle p(u, v, w),$
	$p(z_1, z_1, z_2).$
	$\langle 1 \rangle q(u),$
	$p(z_1, z_2, z_1).$
	$\langle 2 \rangle v = x,$
	$q(c).$
	$\langle 3 \rangle w = x,$
	$use(x) :- \dots$
	$\langle 4 \rangle use(x).$

At point  $\langle 0 \rangle$  the activation set is  $\emptyset$ .

Abstract unification of  $p(u, v, w) = p(z_1, z_1, z_2)$  produces  $\{\emptyset, \{u, v, z_1\}, \{w, z_2\}\}$ , that projected onto the variable set  $\{u, v, w, x\}$  produces  $\{\emptyset, \{u, v\}, \{w\}\}$ .

The same way, abstract unification of  $p(u, v, w) = p(z_1, z_2, z_1)$  produces  $\{\emptyset, \{u, w\}, \{v\}\}$ .

Results from the two clauses are combined through set union. Thus abstract activation at point  $\langle 1 \rangle$  is  $\{\emptyset, \{u, v\}, \{u, w\}, \{v\}, \{w\}\}$ .

<u>query</u>		<u>program</u>
?-	$\langle 0 \rangle p(u, v, w),$	$p(z_1, z_1, z_2).$
	$\langle 1 \rangle q(u),$	$p(z_1, z_2, z_1).$
	$\langle 2 \rangle v = x,$	$q(c).$
	$\langle 3 \rangle w = x,$	$use(x) :- \dots$
	$\langle 4 \rangle use(x).$	

$\langle 1 \rangle$  the state is  $\{\emptyset, \{u, v\}, \{u, w\}, \{v\}, \{w\}\}$ .

$\langle 2 \rangle$  as  $u$  is instantiated to a ground term, all pairs containing  $u$  are removed.  
Thus, we get  $\{\emptyset, \{v\}, \{w\}\}$ .

$\langle 3 \rangle$  the activation set is  $\{\emptyset, \{v, x\}, \{w\}\}$ .

$\langle 4 \rangle$  the activation set is  $\{\emptyset, \{v, x, w\}\}$ .

This says that when  $use$  is called, program terms variables  $v$ ,  $x$  and  $w$  are instantiated to terms that *may* share exactly the same variables.

## Correctness and Optimality

---

The results about optimality are similar to the ones for Pos:

- ↪ abstract union, projection, and unification in `Sharing` are correct (as expected!)
- ↪ also in the case of `Sharing`,  $\gamma$ -optimality is too strong to be met.
- ↪ abstract union and projection operation in `Sharing` are  $\alpha$ -optimal (complete)
- ↪ abstract unification is just optimal.

## Part II

# Comparing Abstract Interpretations

## The notion of Quotient

---

An abstract domain  $D$  expresses, in general, several properties of the concrete domain.

Given an abstract domain  $D$  and a property  $P$  expressed by  $D$ , we may identify the subset of  $D$  that is useful for computing  $P$ -information:

*the quotient of  $D$  with respect to  $P$ .*

**Main Idea:** Construct equivalence classes of elements of  $D$  such that elements in the same class have the same behaviour with respect to the property  $P$ , i.e. such that there is no sequence of (abstract) operations that distinguish them wrt property  $P$ .

## The notion of Quotient

---

- let  $\alpha_{DP}$  be the abstraction function from  $D$  to  $P$ , and let  $\mu$  be any operation in  $D$ . Define the equivalence relation  $r_p$  on  $D$  by

$$(d_1, d_2) \in r_p$$

$$\Leftrightarrow$$

$$\forall i \geq 0 : \alpha_{DP}(\mu^i(d_1)) = \alpha_{DP}(\mu^i(d_2))$$

- The relation  $r_p$  is additive when  $\forall S \subseteq r_p$ , if  $S_1 = \{a \mid (a, b) \in S\}$  and  $S_2 = \{b \mid (a, b) \in S\}$ , it is true that  $(\sqcup_D S_1, \sqcup_D S_2) \in r_p$ .
- If  $r_p$  is additive, the quotient of  $D$  with respect to  $P$  is the set  $\mathcal{Q}_P(D)$  defined by:

$$\mathcal{Q}_P(D) = \{\sqcup_D [d]_p \mid d \in D\}.$$

and this is a complete lattice that abstracts  $D$  and is abstracted by  $P$ .

## Comparing Abstract Interpretations

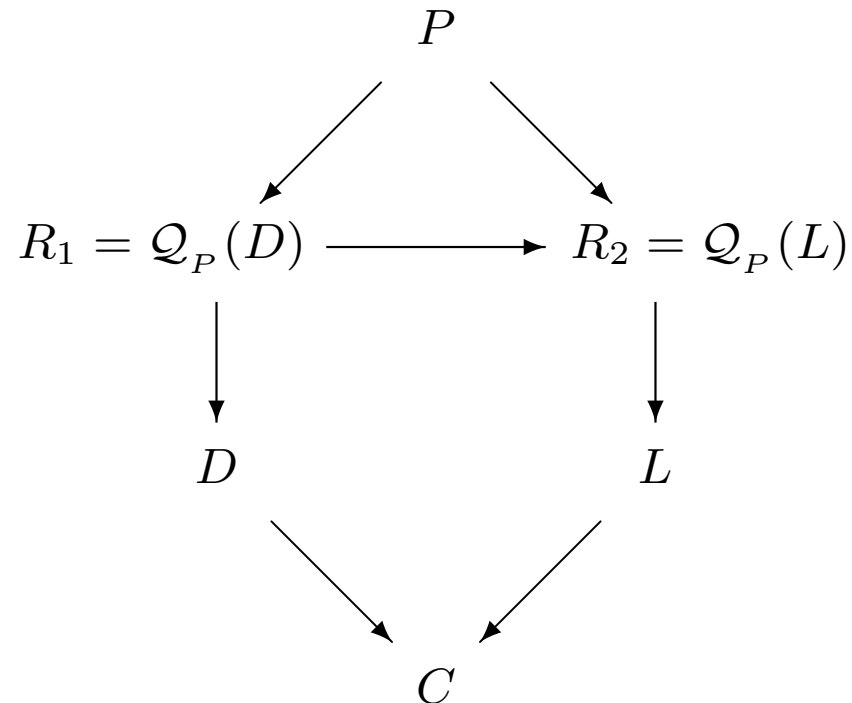
---

Through the notion of quotient, rather sophisticated *comparisons* between domains, can be carried out:

- ↪ Assume to have two abstract domains that both compute some property  $P$ , but that also express distinct properties and thus are incomparable as a whole.
- ↪ Such domains can be compared with respect to the precision with which they compute  $P$ -information, by comparing their quotients with respect to  $P$ .

## Comparing Abstract Interpretations

---



The arrows correspond to concretization functions.

Optimality conditions are required on operations on  $D$  and  $L$ .

**Theorem** *The computation of  $P$  using the domain  $L$  is at least as precise as the computation of  $P$  using the domain  $D$ .*

## The quotient of Sharing wrt Groundness

---

We have already observed that `Sharing` expresses also information relevant for groundness analysis. More precisely,

- ↗ An element  $S$  of `Sharing` may express *definite groundness* of a variable, if it belongs to the "variables of interest" and it does not appear in  $S$ .
- ↗ An element  $S$  of `Sharing` may express *groundness dependency*.

For instance, if the variables of interest are  $\{u, v, w, x\}$ , the element  $S = \{\emptyset, \{v, x, w\}\}$  says that:

- $u$  is ground
- $v, x$  and  $w$  are instantiated to terms that may share exactly the same variables. In other words, for every substitution  $\sigma$  represented by  $S$ , the terms  $\sigma(v)$  cannot contain a variable that is not contained also in terms  $\sigma(x)$  and  $\sigma(w)$ .

## The quotient of Sharing wrt Groundness

---

We say that a set of variables  $V_1$  covers another set of variables  $V_2$  with respect to a substitution  $\sigma$  if for each instance  $\sigma'$  of  $\sigma$ , if  $\sigma'$  grounds all variables in  $V_1$  it also grounds all the variables in  $V_2$ .

Let  $S \in \text{Sharing}$ , and let  $V_1, V_2$  be two sets of variables (in the set of variables of interest for  $S$ ).

$$\begin{array}{c}
 S \text{ implies that } V_1 \text{ covers } V_2 \\
 \text{if} \\
 \forall A \in S : V_2 \cap A \neq \emptyset \Rightarrow V_1 \cap A \neq \emptyset.
 \end{array}$$

### Example

$\rightsquigarrow \{\emptyset, \{u, v\}, \{u, w\}, \{v\}, \{w\}\}$  says that  $\{v, w\}$  covers  $\{u\}$ .

$\rightsquigarrow \{\emptyset, \{u, v\}, \{w\}\}$  says that  $\{u\}$  covers  $\{v\}$  and vice versa (the two variables are equivalent with respect to groundness)

## Comparing Pos and Sharing wrt GR

---

<u>query</u>		<u>program</u>
?–	$\langle 0 \rangle p(u, v, w),$	$p(z_1, z_1, z_2).$
	$\langle 1 \rangle q(u),$	$p(z_1, z_2, z_1).$
	$\langle 2 \rangle v = x,$	$q(c).$
	$\langle 3 \rangle w = x,$	$use(x) :- \dots$
	$\langle 4 \rangle use(x).$	

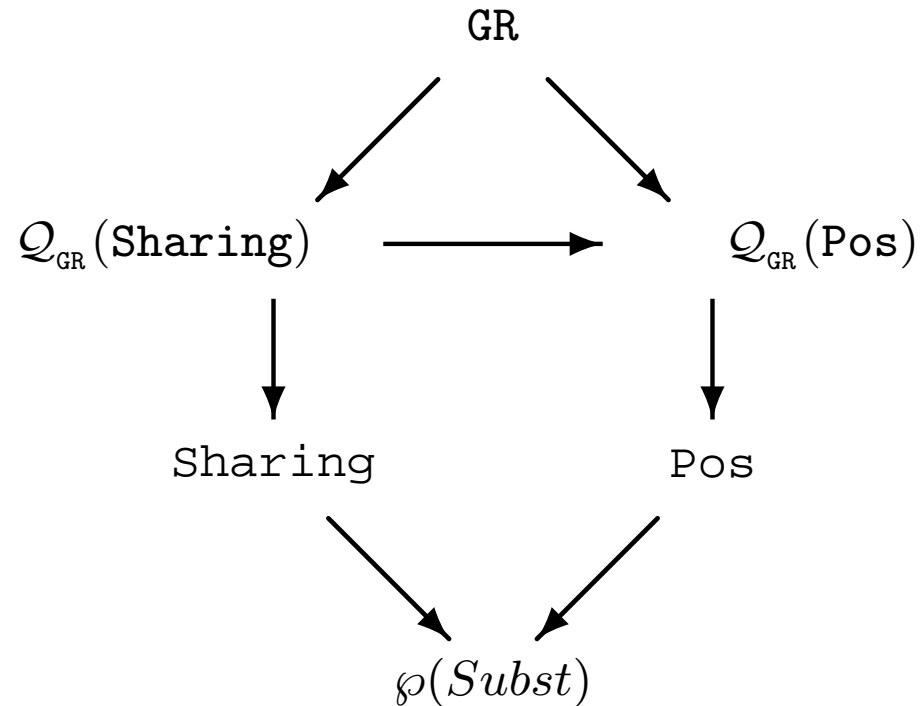
Let us compare the two abstract executions

$\langle 1 \rangle$	$(u \leftrightarrow v) \vee (u \leftrightarrow w)$	$\{\emptyset, \{u, v\}, \{u, w\}, \{v\}, \{w\}\}$
$\langle 2 \rangle$	$u \wedge (v \vee w)$	$\{\emptyset, \{v\}, \{w\}\}$
$\langle 3 \rangle$	$u \wedge ((v \wedge x) \vee (w \wedge (v \leftrightarrow x)))$	$\{\emptyset, \{v, x\}, \{w\}\}$
$\langle 4 \rangle$	$u \wedge v \wedge w \wedge x$	$\{\emptyset, \{v, x, w\}\}$

Thus, at each step Pos is more precise than Sharing wrt groundness.

## Comparing Pos and Sharing wrt GR

---



- ↪ The quotient of Pos with respect to GR is Pos itself, as the associated relation  $r_{GR}$  is just the identity.
- ↪ The quotient of Sharing with respect to GR is a domain equivalent to Def, a proper sublattice of Pos.

## A characterization of $r_{GR}$ for Sharing

---

↪ We may define a function that extracts the covering information from Sharing.

Let  $V$  denote the set of variables of interest.

$$\mathcal{C}(S) = \begin{cases} \text{false} & \text{if } S = \emptyset \\ \bigwedge \left\{ \bigwedge W \rightarrow x \mid \begin{array}{l} x \in V, W \subseteq V \\ \forall A \in S. x \in A \Rightarrow \\ W \cap A \neq \emptyset \end{array} \right\} & \text{otherwise} \end{cases}$$

↪ For instance,

$$\mathcal{C}(\{\emptyset, \{u, v\}, \{u, w\}, \{v\}, \{w\}\}) \equiv (v \wedge w) \rightarrow u$$

↪ Let  $S_1, S_2 \in \text{Sharing}$ ,  $(S_1, S_2) \in r_{GR}$  if and only if  $\mathcal{C}(S_1) \equiv \mathcal{C}(S_2)$ .

## The domain Def

---

↪ The function  $\mathcal{C}$  is nothing but an abstraction function from `Sharing` to the subset of `Pos` of formulas that satisfy the following model intersection property:

$$M_1 \models f, M_2 \models f \quad \Rightarrow \quad M_1 \cap M_2 \models f$$

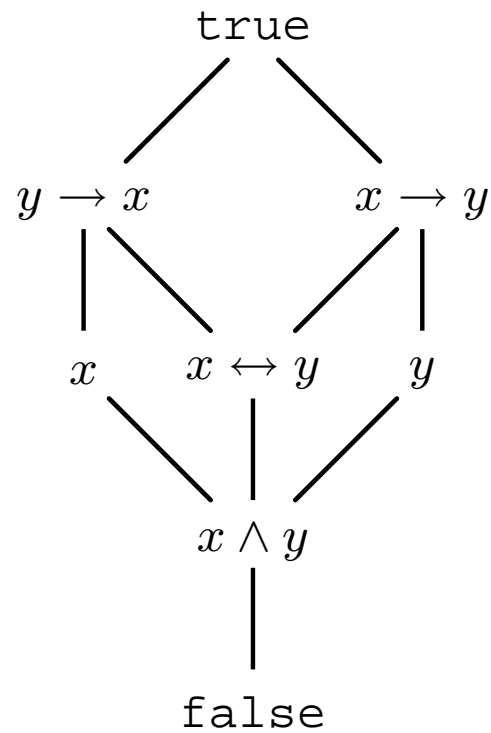
↪ The name “definite” for such formulas comes from the following well-known syntactical characterization: for each  $f \in \text{Def}$  there is an equivalent formula consisting of a conjunction of definite implications of the form  $\bigwedge W \rightarrow x$ .

↪ `Def` is properly included in `Pos`. For instance, the formula  $x \vee y \in \text{Pos} \setminus \text{Def}$ .

In fact, let  $M_1 = \{x\}$ , and  $M_2 = \{y\}$ , it is immediate to see that  $M_1 \models x \vee y, M_2 \models x \vee y$  whereas  $M_1 \cap M_2 = \emptyset \not\models x \vee y$ .

## The lattice Def

---

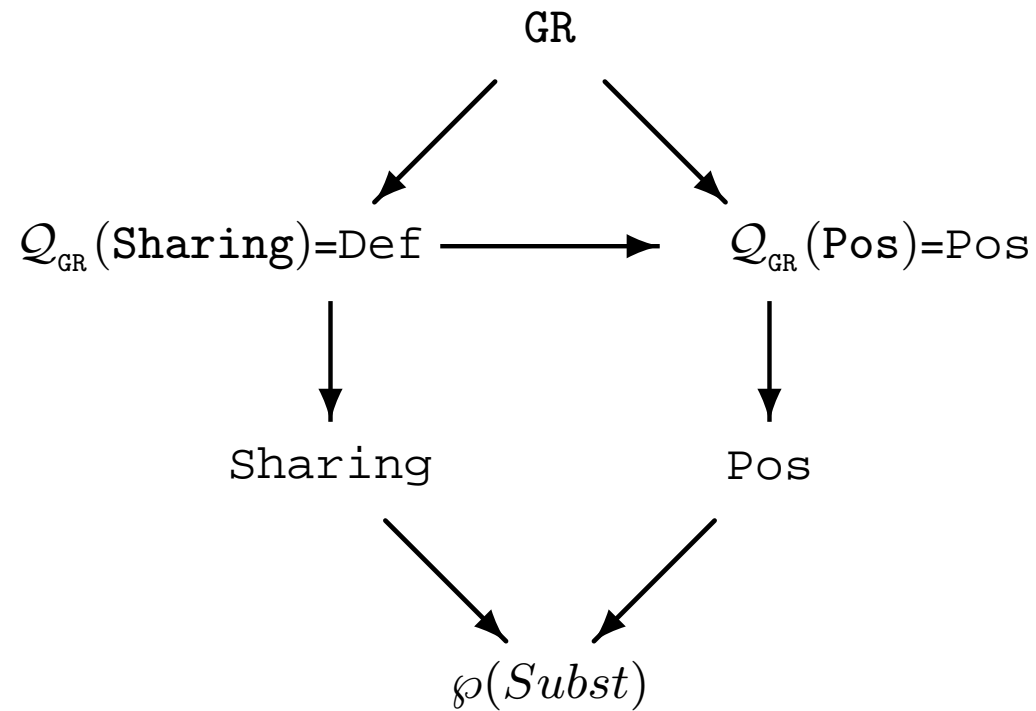


With respect to Pos, Def is not able to express logical disjunction.

It is easy to see that there is a Galois Insertion between Def and Pos.

## The resulting picture

---



By the theory above, groundness analysis using `Pos` is (strictly) more precise than than groundness analysis using `Sharing`.

## Part III

# Combining and Decomposing Abstract Domains

## Combining Domains

---

We may have a *several* analyses, and may want to combine them into *one* analysis. We'll see a number of techniques to combine analyses:

Direct Product

Reduced Product

Pseudo-Reduced Product

Granger's Product

Open Product

The benefit of having such a catalogue of techniques is twofold:

- ~> relative small set of "basic" analyses, whose correctness have been established, can be used to construct rather sophisticated analyses
- ~> from an implementation point of view, opens up the possibility of reusing existing implementations.

## Direct Product

---

The *direct product* is the simplest combination of abstract domains.

Given two abstract domains  $D_1$  and  $D_2$  with concretization functions  $\gamma_1 : D_1 \rightarrow C$  and  $\gamma_2 : D_2 \rightarrow C$ , the *direct product domain* is the domain

$$D = D_1 \times D_2$$

with concretization function

$$\gamma(\langle d_1, d_2 \rangle) = \gamma_1(d_1) \sqcap \gamma_2(d_2)$$

Given a concrete operation  $OP_c : C \rightarrow C$  and two corresponding abstract operations  $OP_1$  and  $OP_2$  on  $D_1$  and  $D_2$ ,

$$OP(\langle d_1, d_2 \rangle) = \langle OP_1(d_1), OP_2(d_2) \rangle.$$

The main disadvantage of the direct product is its lack of precision since there is no interaction between the components.

## Reduced Product

---

The *reduced product* was proposed by the Cousots to overcome some of the limitations of the direct product.

Its key idea is to cluster into equivalence classes the elements of the direct product having the same concretization and to work on the more precise representative of each class.

More formally, consider the function  $reduce : D_1 \times D_2 \rightarrow D_1 \times D_2$  defined as

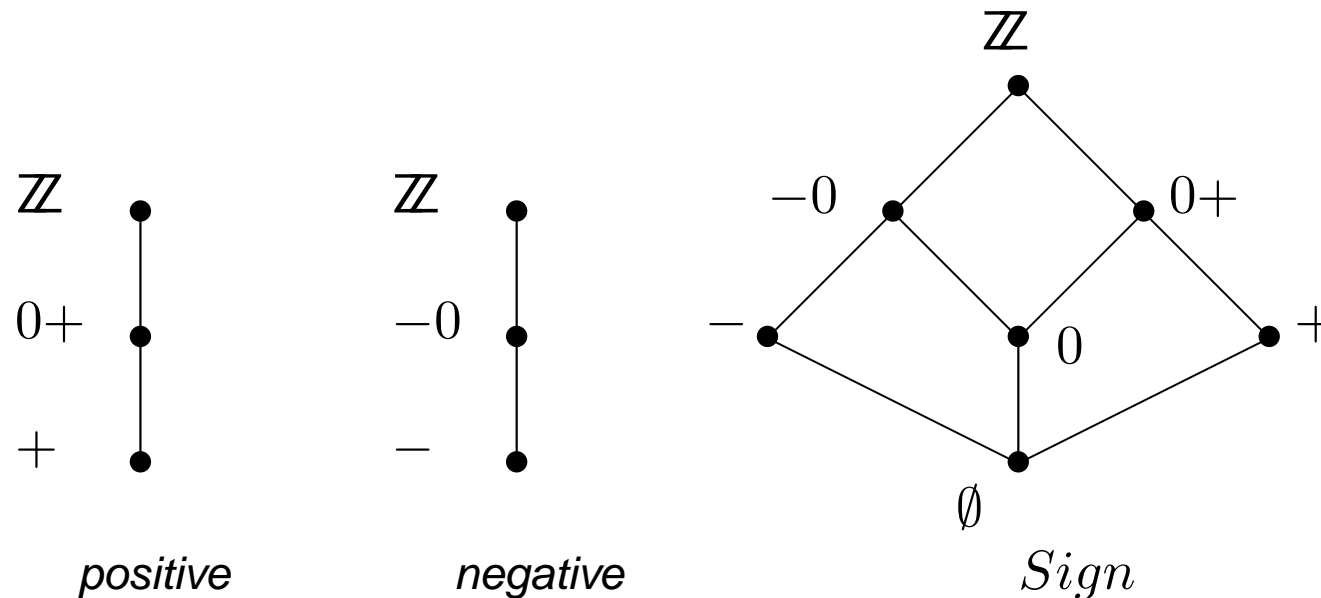
$$reduce(\langle d_1, d_2 \rangle) = \sqcap \{ \langle e_1, e_2 \rangle \mid \gamma(\langle e_1, e_2 \rangle) = \gamma(\langle d_1, d_2 \rangle) \}.$$

The reduced product domain is the domain

$$D = \{ reduce(\langle d_1, d_2 \rangle) \mid d_1 \in D_1 \wedge d_2 \in D_2 \}$$

The reduced product is the most precise refinement of the cartesian product.

## The reduced product: Example

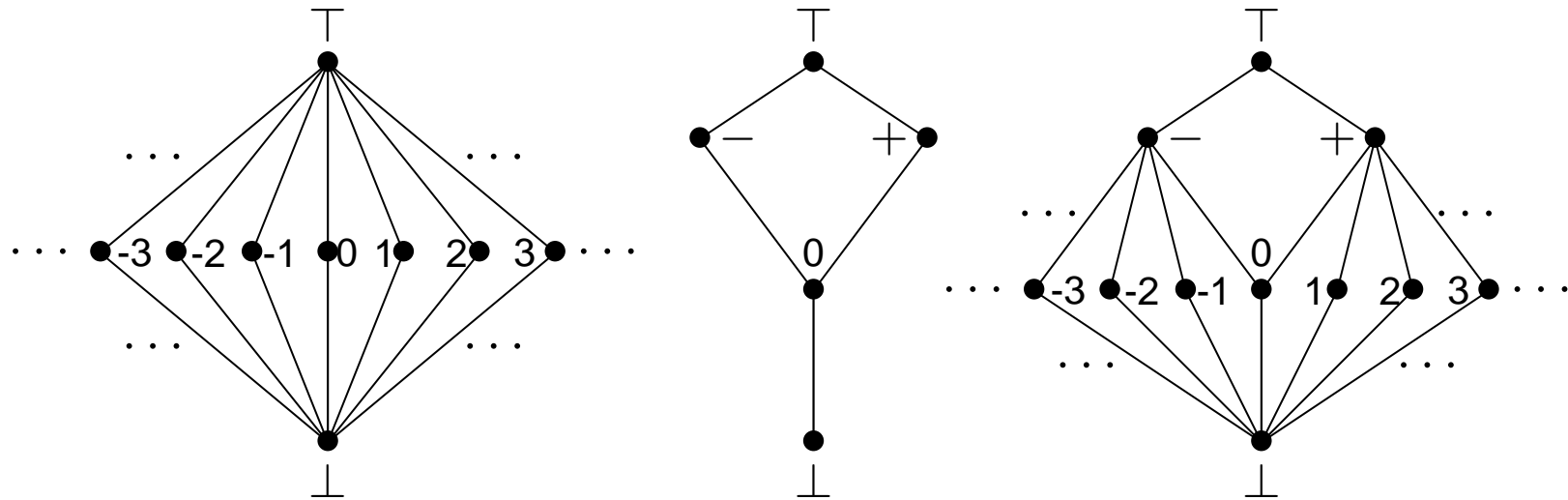


Observe that  $Sign$  has two new elements with respect to  $positive$  and  $negative$ :  $0$  and  $\emptyset$ .

These elements are obtained by combining, by set intersection of the corresponding sets of integers, respectively,  $0+$  with  $-0$  and  $+$  with  $-$ .

In this example, reduction is necessary only for identifying distinct pairs of elements denoting  $\emptyset$ , as, for instance, the pairs  $\langle +, - \rangle$  and  $\langle +, -0 \rangle$ .

## The reduced product: another example



The first domain is the one used for [constant propagation analysis](#).

The second domain is the one used for [sign analysis](#).

The third one is the [reduced product](#) of the other two.

## Reduced Product Operations

---

It is possible to specify optimal abstract operations for the reduced domain.

If  $\alpha_i$  is an abstraction function for  $D_i$  ( $1 \leq i \leq 2$ ), an optimal abstraction of a concrete operation  $OP_c$  can be specified as

$$OP(\langle d_1, d_2 \rangle) = reduce(\langle \alpha_1(r), \alpha_2(r) \rangle)$$

$$\text{where } r = OP_c(\gamma(\langle d_1, d_2 \rangle)).$$

However, this definition is a theoretical concept, since it uses the (non-computable) concretization functions.

The implementation of abstract operations along this specification would necessitate the revision of the original design phase.

## Pseudo-Reduced Product

---

Several papers refer to a simpler version of the reduced product, called the *pseudo-reduced product*.

In the pseudo-reduced product, the domain remains the same as in the direct product but the abstract operations are defined as

$$\text{OP}(\langle d_1, d_2 \rangle) = \text{reduce}(\langle \text{OP}_1(d_1), \text{OP}_2(d_2) \rangle).$$

The abstract operations are, in general, non-optimal.

On one hand, the definition still relies on the concretization function (a semantic notion).

On the other hand, additional accuracy can be obtained by defining new operations where the operations on  $D_1$  and  $D_2$  interact.

## Granger's Product

---

Granger's Product is an elegant solution to the problem of computing a good approximation of the *reduce* function. The key idea is to define two new operations  $\sigma_1$  and  $\sigma_2$  on the product  $D_1 \times D_2$  that “reduce” each of the components respectively, and to iterate the application of these two operations.

More precisely,

- let  $\sigma_1 : D_1 \times D_2 \rightarrow D_1$  such that  $\sigma_1(d_1, d_2) \leq d_1$  and  $\gamma(\langle \sigma_1(d_1, d_2), d_2 \rangle) = \gamma(\langle d_1, d_2 \rangle)$
- let  $\sigma_2 : D_1 \times D_2 \rightarrow D_2$  such that  $\sigma_2(d_1, d_2) \leq d_2$  and  $\gamma(\langle d_1, \sigma_2(d_1, d_2) \rangle) = \gamma(\langle d_1, d_2 \rangle)$ .
- *Granger's product* is defined as the fixpoint of a decreasing iteration sequence  $(\langle \eta_1^n, \eta_2^n \rangle)_{n \in \mathbb{N}}$  defined as follows:

$$\begin{aligned} \langle \eta_1^0, \eta_2^0 \rangle &= \langle d_1, d_2 \rangle \\ \langle \eta_1^{n+1}, \eta_2^{n+1} \rangle &= \langle \sigma_1(\eta_1^n, \eta_2^n), \sigma_2(\eta_1^n, \eta_2^n) \rangle. \end{aligned}$$

## Granger's Product

---

The main benefit of this approach is that the designer must only implement (an approximation of) the  $\sigma_i$  functions relating  $D_i$  with the cartesian product  $D_1 \times D_2$ .

Granger's approach imposes reasonable effort and reuses the existing implementation.

However it has some limitations.

- ↪ The first limitation is that the domains only interact after completion of the individual operations; letting them interact *during* the individual operations may lead to additional accuracy that cannot be recovered by Granger's product
- ↪ The new operations are defined in terms of the product, which we would like to avoid in order to provide a fully automated combination of domains, to guide the design phase, and to hide irrelevant implementation details.

## Open Product

---

The *open product* is an attempt to remedy these two limitations. More precisely, the open product aims at providing a framework to

- ~> implement more precise approximations of the reduced product by letting domains interact during and after the operations;
- ~> provide methodological guidance on how to construct abstract domains that lead to effective and precise combinations;
- ~> support an encapsulation of the representation and implementation of each component, thus avoiding operations that manipulate several domains simultaneously.

## Queries and Open Operations

---

An *open abstract interpretation* differs from a traditional abstract interpretation (domain and operations) by introducing the notion of *queries* and *open operations*.

↪ A query is simply a function giving information about the properties captured by the domain.

Let  $Arg$  be a set. A test is a boolean function  $\mathcal{T} : Arg \rightarrow Bool$ .

A query on the domain  $D$  wrt a given set  $Arg$  is a monotone function  $Q : D \rightarrow Test_{Arg}$  which maps elements of the domain  $D$  onto tests.

↪ An open operation is an abstract operation except that it receives one or more boolean functions describing additional properties of the concrete objects (e.g., properties not captured by the domain).

An open operation on the domain  $D$  is a function

$OP : \langle Arg \rightarrow Bool \rangle_{j \in I} \rightarrow D \rightarrow D$  ( $|I| \geq 0$ ) which maps a tuple of tests onto an operation.

## Supporting Queries

---

- ↪ Domains may support some queries  $Q_i$  (acting as servers)
- ↪ Operations defined on a domain may require info from the environment (acting like clients).

The open product  $D_1 \otimes D_2$  is defined as follows.

- ↪ the domain is the cartesian product  $D_1 \times D_2$ ;
- ↪ the partial ordering is the product of  $\leq_1$  and  $\leq_2$ ;
- ↪ the query  $Q_i$  is defined as  $Q_i(d_1, d_2) = Q_i^1(d_1) \vee Q_i^2(d_2)$
- ↪ operations  $OP : (D_1 \times D_2) \rightarrow (D_1 \times D_2)$  are defined by:
 
$$OP(d_1, d_2) =$$

$$(OP^1(\langle Q_i(d_1, d_2) \rangle_{i \in I})(d_1), OP^2(\langle Q_i(d_1, d_2) \rangle_{i \in I})(d_2))$$

## Concrete Queries

---

Consider the following queries, defined in the concrete domain  $\wp(Subst)$ .

$\mathcal{C}$ -GROUND and  $\mathcal{C}$ -NOSHARING, which provide information on groundness and sharing, respectively.

$$\mathcal{C}\text{-GROUND}(\Theta)(x_i) = \begin{cases} \text{true} & \text{if } \forall \theta \in \Theta : x_i\theta \text{ is a ground term} \\ \text{false} & \text{otherwise} \end{cases}$$

$$\mathcal{C}\text{-NOSHARING}(\Theta)(x_i, x_j) = \begin{cases} \text{true} & \text{if } \forall \theta \in \Theta : x_i\theta, x_j\theta \text{ do not share} \\ \text{false} & \text{otherwise} \end{cases}$$

## Abstract Queries in Pos

---

In Pos, the queries are abstracted by the functions

- Pos-GROUND:  $\text{Pos} \rightarrow \text{Var} \rightarrow \text{Bool}$
- Pos-NOSHARING:  $\text{Pos} \rightarrow \text{Var} \times \text{Var} \rightarrow \text{Bool}$

whose definitions are as follows

$$\text{Pos-GROUND}(f)(x_i) \quad \Leftrightarrow \quad (f \rightarrow x_i)$$

$$\text{Pos-NOSHARING}(f)(x_i, x_j) \quad \Leftrightarrow \quad (f \rightarrow x_i) \vee (f \rightarrow x_j)$$

## Abstract Queries in Sharing

---

In `Sharing`, the queries are abstracted by the functions

- `Sharing-GROUND`:  $\text{Sharing} \rightarrow \text{Var} \rightarrow \text{Bool}$
- `Sharing-NOSHARING`:  $\text{Sharing} \rightarrow \text{Var} \times \text{Var} \rightarrow \text{Bool}$

whose definitions are as follows

$$\text{Sharing-GROUND}(S)(x_i) \quad \Leftrightarrow \quad x_i \notin \text{Var}(S)$$

$$\text{Sharing-NOSHARING}(S)(x_i, x_j) \quad \Leftrightarrow \quad \exists A \in S \text{ such that } \{x_i, x_j\} \subseteq A$$

## Open Unification in Pos

---

The basic unification on Pos can be defined as

$$\mathbf{u}(\text{GROUND})(f, x_i, x_j) = \begin{cases} f \wedge x_i \wedge x_j & \text{if } \text{GROUND}(x_i) \vee \text{GROUND}(x_j) \\ f \wedge (x_i \leftrightarrow x_j) & \text{otherwise} \end{cases}$$

## Open Unification in Sharing

---

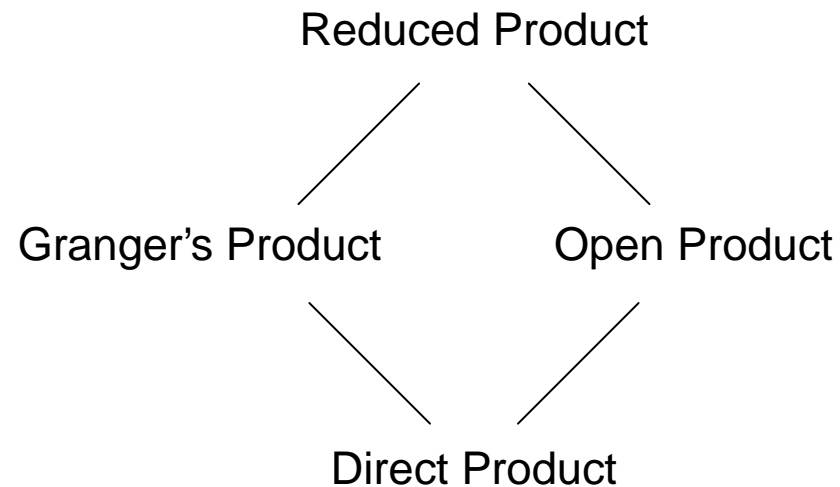
The basic unification on `Sharing` can be defined as

$$\mathbf{u}(\mathbf{GROUND})(S, x_i, x_j) = \left\{ \begin{array}{l} (S - (rel(x_i, S) \cup rel(x_j, S))) \\ \quad \text{if } \mathbf{GROUND}(x_i) \vee \mathbf{GROUND}(x_j) \\ \\ (S - (rel(x_i, S) \cup rel(x_j, S))) \cup (rel(x, B) \otimes rel(t, B))^* \\ \quad \text{otherwise} \end{array} \right.$$

## Summary Picture

---

The following figure depicts the relations among product definitions with respect to the accuracy of the operations, the Reduced Product being the most precise and the Direct Product the least accurate.



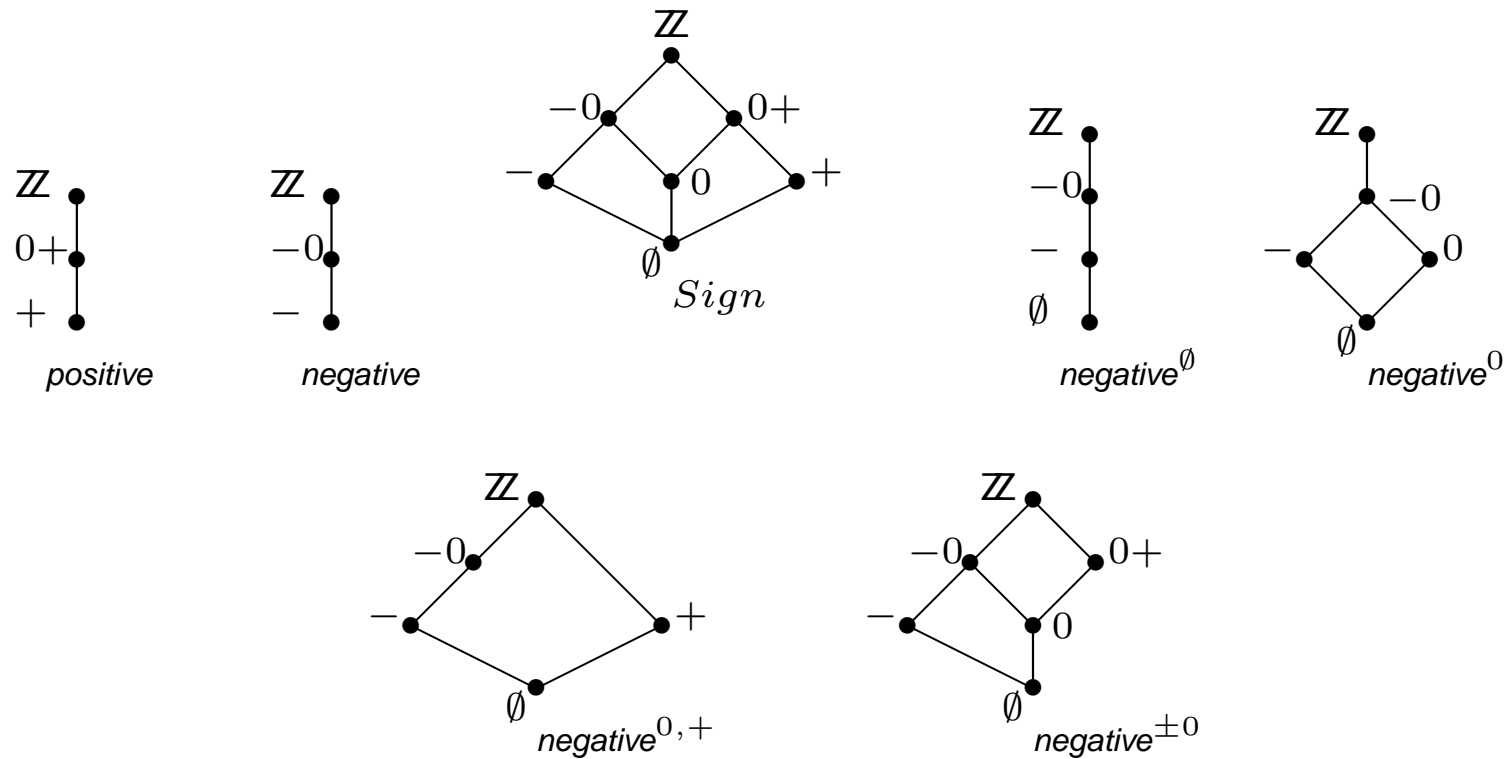
## Decomposing Domains

---

- ↪ Reduced product of abstract domains is the most precise operation for domain composition in abstract interpretation.
- ↪ We are interested also in its inverse operation: domain complementation. Starting from any two domains  $C$  and  $D$ , with  $D$  more abstract than  $C$ , complementation operation gives as result the most abstract domain  $C \sim D$ , whose reduced product with  $D$  is exactly  $C$ .

Complementation provides a systematic way to design new abstract domains, and it allows to systematically decompose domains. Also, such an operation allows to simplify domain verification problems, and it yields space-saving representations for complex domains.

## Example



All the domains *Sign*, *negative*, *negative <sup>$\emptyset$</sup>* , *negative<sup>0</sup>*, *negative<sup>0,+</sup>*, and *negative <sup>$\pm 0$</sup>*  are all and only those abstractions of *Sign* which, once combined with *positive* by reduced product, give *Sign* back. Hence, being the smallest,  $\text{negative} = \text{Sign} \sim \text{positive}$ .

## A method to generate the complement

---

It is well known the correspondence between Galois insertions and closure operators ( $\rho = \gamma \circ \alpha$ ).

Given a complete lattice  $\langle C, \preceq \rangle$ , and  $X \subseteq C$ ,

- $\text{maxs}(X)$  denotes the set of maximal elements of  $X$ , viz., the objects  $x \in X$  such that  $\forall y \in X. x \preceq y \Rightarrow x = y$
- $\text{Cl}(X)$  denotes the greatest closure operator containing  $X$ .

Observe that, if  $X, Y$  are (sets of fixpoints of) closure operators on  $C$ , then  $\text{Cl}(X \cup Y) = X \sqcap Y = \{x \wedge y \mid x \in X \text{ and } y \in Y\}$ , and that  $\text{Cl}(X)$  can be computed by adding to  $X$  all the greatest lower bounds of subsets of  $X$ .

## A method to generate the complement

---

If  $C$  is finite, we may build  $C \sim D$  as the saturation point of an increasing sequence of closure operators.

Let  $\{X_n\}_{n \in \mathbb{N}}$  be the following family:

$$\begin{aligned}
 X_0 &= \{\top\} \\
 X_1 &= Cl(X_0 \cup maxs(C \setminus Cl(X_0 \cup D))) \\
 &\vdots \\
 X_n &= Cl(X_{n-1} \cup maxs(C \setminus Cl(X_{n-1} \cup D))) \\
 &\vdots
 \end{aligned}$$

If  $C$  is finite, then the sequence  $\{X_n\}_{n \in \mathbb{N}}$  converges in a finite number of steps to the complement  $C \sim D$ .

If  $C$  is infinite, then the same construction can be extended so as to obtain a transfinite sequence.

## The complement of Def wrt Sharing

---

We compute the complement of Def with respect to itt Sharing by using that methodology.

The domain Def can be represented as the closure operator  $\gamma \circ \alpha$  on Sharing. This closure operator is the function that, for any  $S \in \text{Sharing}$ , gives the closure of  $S$  under set union. Formally,  $\gamma \circ \alpha = \text{clu}$ , where

$$\text{clu}(S) = \{A \mid \exists A_1, \dots, A_n \in S. A = A_1 \cup \dots \cup A_n\}.$$

We construct a chain  $X_0, X_1, X_2, \dots$  of subsets of Sharing.

According to such definitions, we have that

$$X_0 = \{\emptyset(\text{Var})\}.$$

## The complement of Def wrt Sharing

---

In order to construct  $X_1$ , consider the set  $X = \mathit{maxs}(\mathit{Sharing} \setminus \mathit{Def})$ .

$$X = \{\wp(\mathit{Var}) \setminus \{A\} \mid A \subseteq \mathit{Var} \text{ and } |A| \geq 2\}.$$

The set  $X_1$  is defined as the closure of  $X_0 \cup X$  under the *glb* on  $\mathit{Sharing}$ , namely under set intersection.

Observe that every element of  $\mathit{Sharing}$  which contains the empty set and all singletons of  $\mathit{Var}$  either is  $\wp(\mathit{Var})$ , or it can be obtained by set intersection of suitable elements of  $X$ .

We therefore have

$$X_1 = \{S \in \mathit{Sharing} \mid \forall x \in \mathit{Var}, \{x\} \in S\}.$$

## The complement of Def wrt Sharing

---

The closure of  $X_1 \cup \text{Def}$  under set intersection coincides with `Sharing`, which implies that we have already reached the limit of the construction, i.e.,  $X_2 = X_1$ .

Thus,

`Sharing`  $\sim$  `Def` =

$$\{S \in \text{Sharing} \mid \forall x \in \text{Var}, \{x\} \in S\}$$

## The complement of Def wrt Sharing

---

For instance, if  $Var = \{x, y\}$ , then  $Sharing \sim Def$  is the simple domain depicted below.

$$\{\emptyset, \{x\}, \{y\}, \{x, y\}\}$$

|

$$\{\emptyset, \{x\}, \{y\}\}$$

This formalizes the idea that it is just the presence of singletons that prevents expressing ground dependencies.

## Conclusions

---

- ~> We have seen a number of different operators to manipulate abstract domains, for combining, comparing and decomposing domains.
- ~> We applied these concepts to the case of two abstract domains for analysis of logic programs. However, the techniques are general, and have been successfully applied also to imperative and functional programming.
- ~> What we have seen is just part of the story. Other researchers have studied other operators like powerset of abstract domains, or refinement operators...
- ~> And there is another part of the story, the one about *implementing* abstract domains.