Modular Construction of Shape-Numeric Analyzers

Xavier Rival
Joint work with Bor-Yuh Evan Chang

INRIA

October 2013

Goals in the design of static analyzers

- **Precision** : ability to infer **complex properties**
  i.e., produce **few false warnings** for safety verification
- **Efficiency** : ability to analyze **large software**
- Also **ability to extend** the analysis : more properties, target codes

We need some work on static analysis engineering

- How to cover a **wide set of properties** ? (pointers, values)
- How to set up an **extensible** analysis framework ?

Example : Astrée analyzer

- **very precise** analysis for safety critical embedded softwares
- **industrial fly-by-wire avionic softwares verified**
- the analyzer's scope was **extended** multiple time

Inter dependences between properties

Example : **balanced search trees**, with complex **recursive invariants**

- **dependence of pointer properties on values properties** : balancing invariants
- **dependence of values properties on pointer properties** : binary search tree property

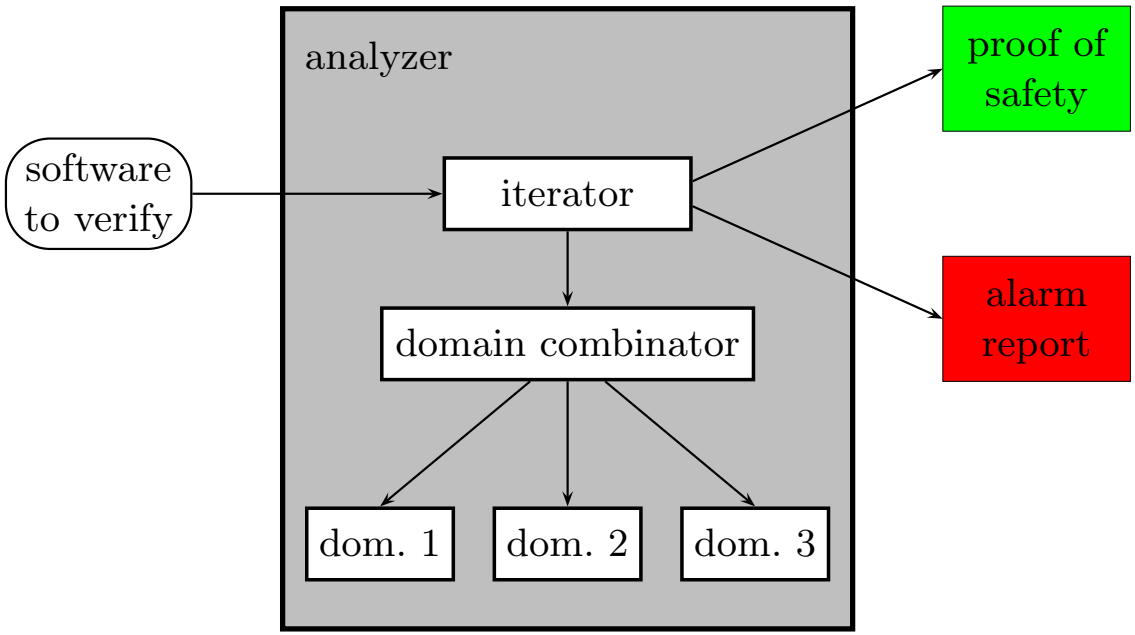
Combined analyses are required

- sequences of analyses would **fail to exchange** important properties
- combined analysis allows for better precision, but are **harder to design**

Modular design of static analyzers

Advantages of a modular abstract domain design

- Easier to **design**, **prove correct** and **implement**
- Allows for a better **precision tuning**
- Open to **abstractions** that were not initially planned

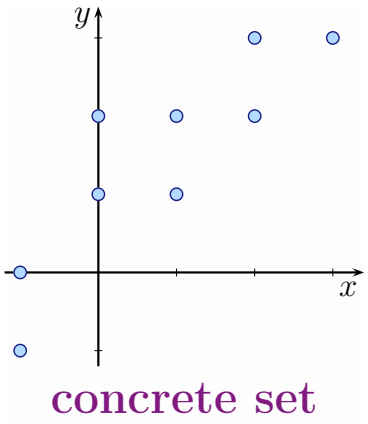


Example : construction of numeric abstractions

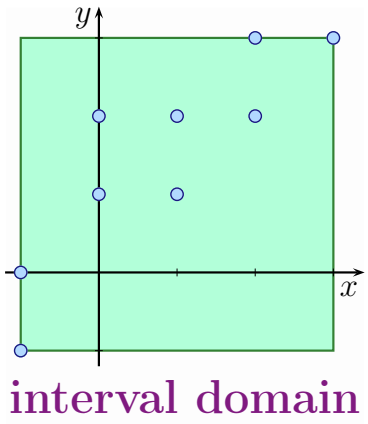
Abstract domains

- Families of **abstract predicates** adapted to static analysis
- **Operations** for the static analysis of concrete operations

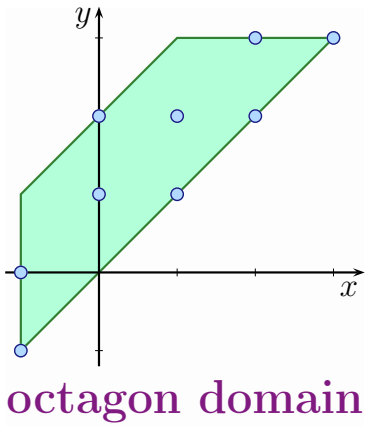
Ex., numeric abstractions : abstraction of **sets of pairs of integers**



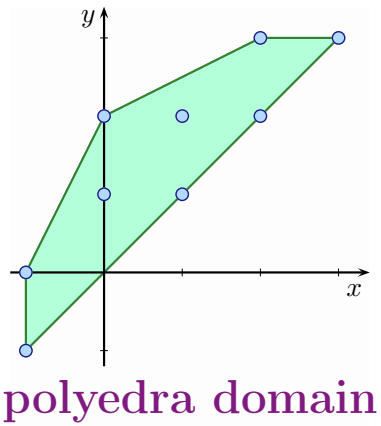
concrete set



interval domain



octagon domain



polyedra domain

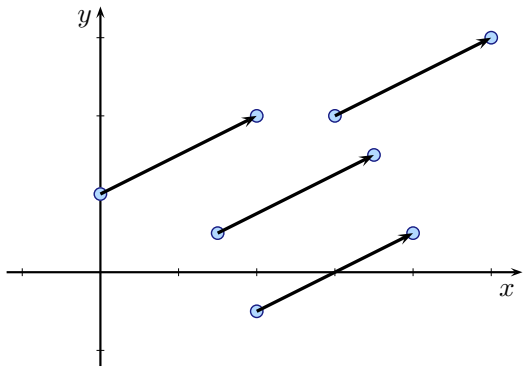
- **Packing** : utilize expensive abstraction only when needed
- **Reduced product** with other numeric abstraction, e.g., **congruences**

Abstraction of execution steps

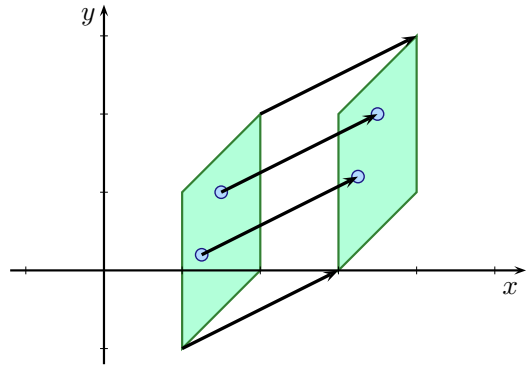
Computing sound abstract transformer

- **Conservative analysis** of concrete execution steps in the abstract
  e.g., **assignments**, **condition tests**...
- Balance between **cost** and **precision**

Example : analysis of a **translation** with **octagons**



concrete transformation



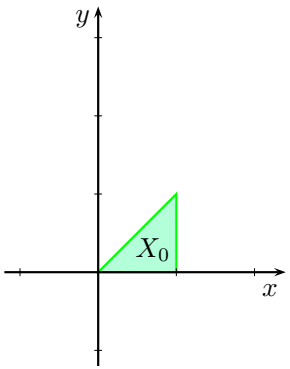
abstract transformation

Abstraction of infinite computations

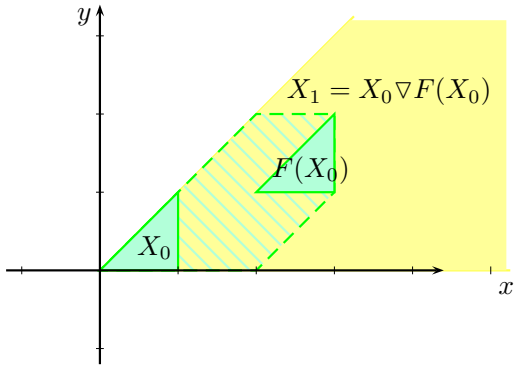
Computing invariants about infinite executions with widening ∇

- **Widening** ∇ over-approximates $\cup$: **soundness guarantee**
- **Widening** ∇ guarantees the **termination of the analyses**

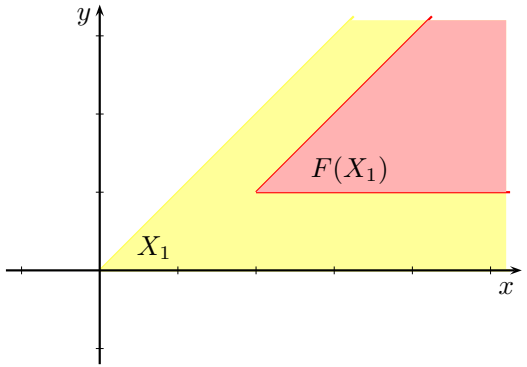
Example : iteration of the translation $(2, 1)$, with **octagons**



initial



iteration 1



iteration 2 : stable !

Outline

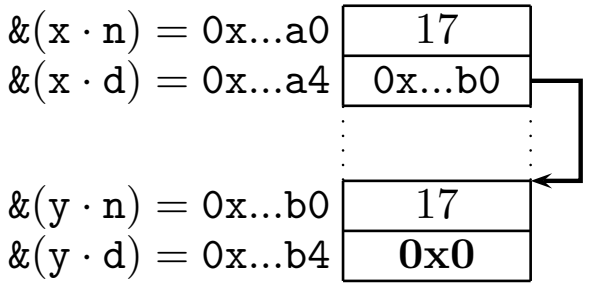
1. **Abstract domain**
   - for **both shape and values**
   - with a **modular** structure
   - closely **matches programs semantics**
2. **Abstract operations**
   - computation of **post-conditions**
   - over-approximation of **join** (abstract join, widening)

Outline

From concrete states to abstraction of sets of states

... without summarization (for now)

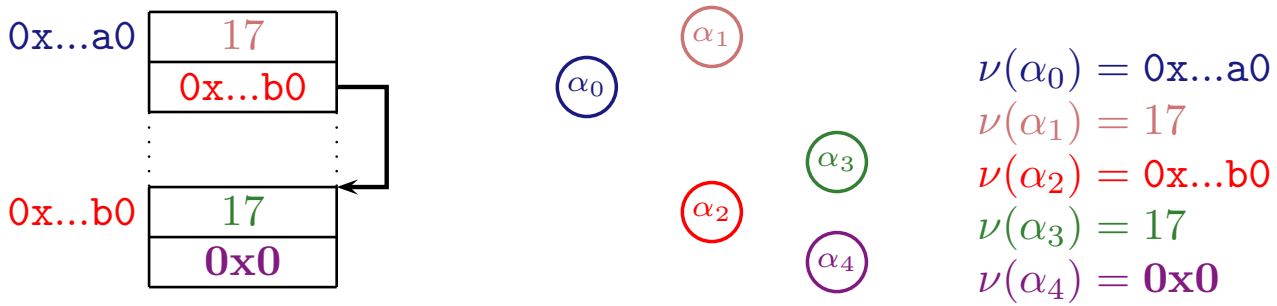
- **Concrete memory states**
  - very **low level** description
  - pointers, numeric values : **raw sequences of bits**



From concrete states to abstraction of sets of states

... without summarization (for now)

- **Concrete memory states**
- **Abstraction of values into symbolic variables** (nodes)

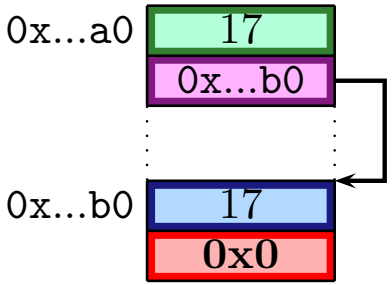


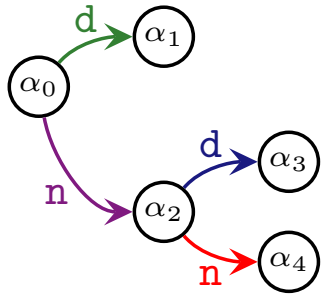
  - characterized by **valuation** ν
  - ν maps **symbolic variables** into **concrete addresses**

From concrete states to abstraction of sets of states

... without summarization (for now)

- **Concrete memory states**
- **Abstraction of values into symbolic variables** / nodes
- **Abstraction of regions into points-to edges**



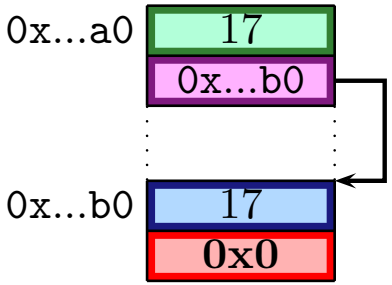


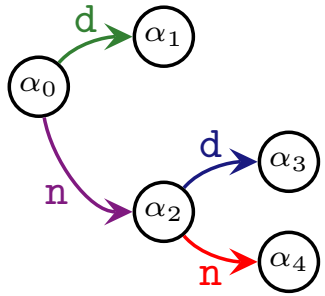
$$\nu(\alpha_0) = \texttt{0x...a0}$$
$$\nu(\alpha_1) = 17$$
$$\nu(\alpha_2) = \texttt{0x...b0}$$
$$\nu(\alpha_3) = 17$$
$$\nu(\alpha_4) = \mathbf{0x0}$$

From concrete states to abstraction of sets of states

... without summarization (for now)

- **Concrete memory states**
- **Abstraction of values into symbolic variables** / nodes
- **Abstraction of regions into points-to edges**





$\nu(\alpha_0) = \texttt{0x...a0}$
$\nu(\alpha_1) = 17$
$\nu(\alpha_2) = \texttt{0x...b0}$
$\nu(\alpha_3) = 17$
$\nu(\alpha_4) = \mathbf{0x0}$

- **Shape graph concretization**

$$\gamma(G) = \{(h, \nu) \mid \ldots\}$$

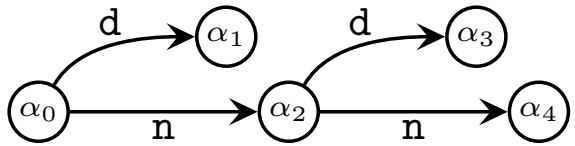
valuation ν plays an important role in the combined domain

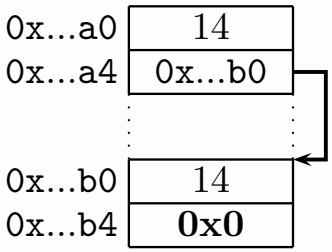
Product with a value abstraction

- **Concrete numeric values** appear in the **valuation** : **abstract ν !**

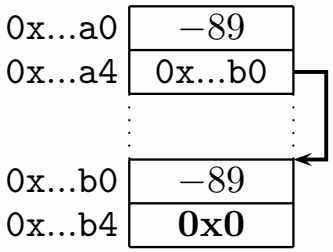
Example : lists of length 2, with equal data fields

- **Memory abstraction** :





ν : $\alpha_1 \mapsto 14$, $\alpha_3 \mapsto 14$, $\ldots \mapsto \ldots$

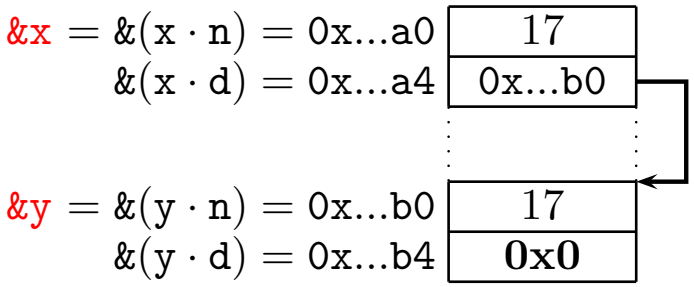


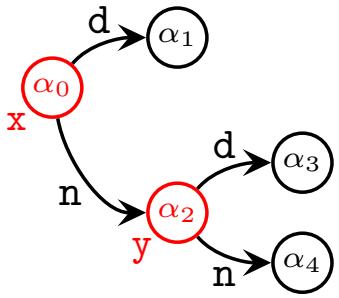
ν : $\alpha_1 \mapsto -89$, $\alpha_3 \mapsto -89$, $\ldots \mapsto \ldots$

- **Abstraction of valuations** : $\alpha_1 = \alpha_3$
  - combined abstraction : **product of shape and numerics** over nodes
  - $\gamma(\mathbf{G}, \mathbf{N}) = \{(h, \nu) \mid (h, \nu) \in \gamma(\mathbf{G}) \wedge \nu \in \gamma(\mathbf{N})\}$

Environment abstraction

- **Addresses** can be read in the **valuations**





$$\nu : \begin{array}{lcl} \alpha_0 & \mapsto & \texttt{0x...a0} \\ \alpha_2 & \mapsto & \texttt{0x...b0} \\ \ldots & \mapsto & \ldots \end{array}$$

$$\mathbf{E} : \begin{array}{ll} \texttt{x} \mapsto \alpha_0 & (\overset{\nu}{\mapsto} \texttt{0x...a0}) \\ \texttt{y} \mapsto \alpha_2 & (\overset{\nu}{\mapsto} \texttt{0x...b0}) \end{array}$$

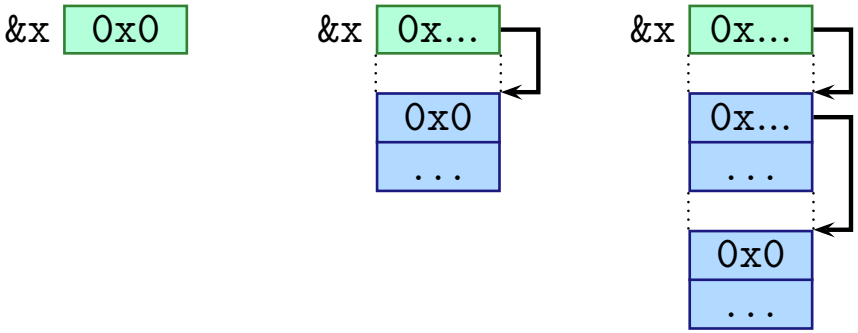
Abstraction of environments :

- mapping **E** from **variables** to **symbolic nodes**
- $\gamma(\mathbf{E}, \mathbf{G}, \mathbf{N}) = \{(e, h) \mid (h, \nu) \in \gamma(\mathbf{G}, \mathbf{N}) \wedge e = \nu \circ \mathbf{E}\}$

Summarization of unbounded heap regions

Example :

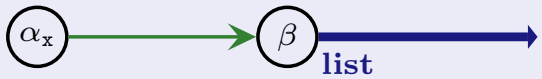
set of all lists of any length



List inductive predicate

$$\begin{aligned}\alpha \cdot \textbf{list} := \\ & (\textbf{emp} \wedge \alpha = \textbf{0x0}) \\ \vee \quad & (\alpha \cdot \mathtt{d} \mapsto \beta_0 * \alpha \cdot \mathtt{n} \mapsto \beta_1 \\ & * \beta_0 \cdot \textbf{list} \wedge \alpha \neq \textbf{0x0})\end{aligned}$$

Inductive summary predicates



Concretization based on **unfolding** :

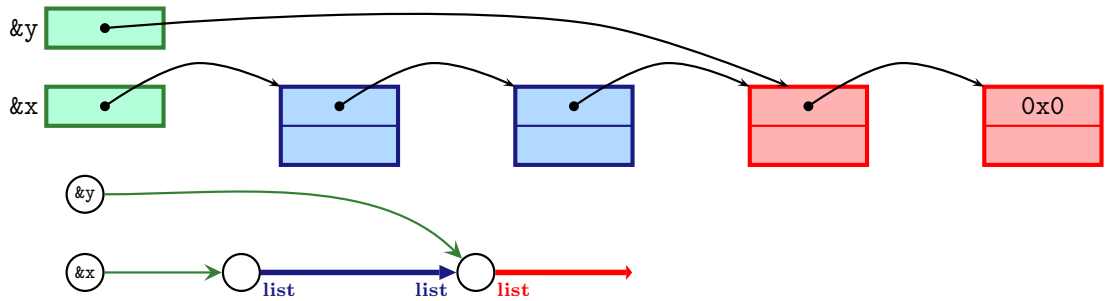
- $\rightsquigarrow$ replaces **an $\alpha \cdot$ list predicate** with **one of its premisces**
- $\gamma(\mathbf{G}, \mathbf{N}) = \bigcup\{\gamma(\mathbf{G}', \mathbf{N}') \mid (\mathbf{G}, \mathbf{N}) \rightsquigarrow (\mathbf{G}', \mathbf{N}')\}$

Summarization of incomplete structures

Structure fragments

- **Common pattern** : structure **traversal** using a **cursor**
- **Segment** abstract predicate : incomplete structure

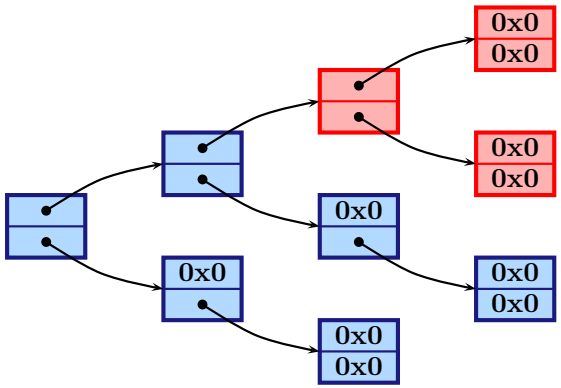
- Example with **lists** :



can be abstracted by :

- Works also for **trees** :

abstracts



Concretization

Denotation of abstract elements

- Function $\gamma_{\mathrm{S}} : \mathbb{D}^{\sharp}_{\mathrm{S}} \to \mathcal{P}(\mathbb{M} \times (\textbf{nodes} \to \textbf{values}))$
- To be used for the **proof of correctness of the analysis**

Separating conjunction :

$$\gamma_{\mathrm{S}}(\mathbf{G}_0 * \mathbf{G}_1) \quad = \quad \{(h_0 \uplus h_1, \nu) \mid (h_0, \nu) \in \gamma_{\mathrm{S}}(\mathbf{G}_0) \wedge (h_1, \nu) \in \gamma_{\mathrm{S}}(\mathbf{G}_1)\}$$

Points-to predicates :

$$\gamma_{\mathrm{S}}(\alpha \cdot \mathtt{f} \mapsto \beta) \quad = \quad \{([\nu(\alpha) + \textbf{offset}(\mathtt{f}) \mapsto \nu(\beta)], \nu) \mid \nu : \textbf{nodes} \to \textbf{values}\}$$

Inductive predicates :

$$\gamma_{\mathrm{S}}(\alpha \cdot \iota) \quad = \quad \bigcup\nolimits_{\mathrm{fin}} \{\gamma_{\mathrm{S}}(\mathbf{G}) \mid \alpha \cdot \iota \rightsquigarrow_{\mathrm{unfold}} \mathbf{G}\}$$

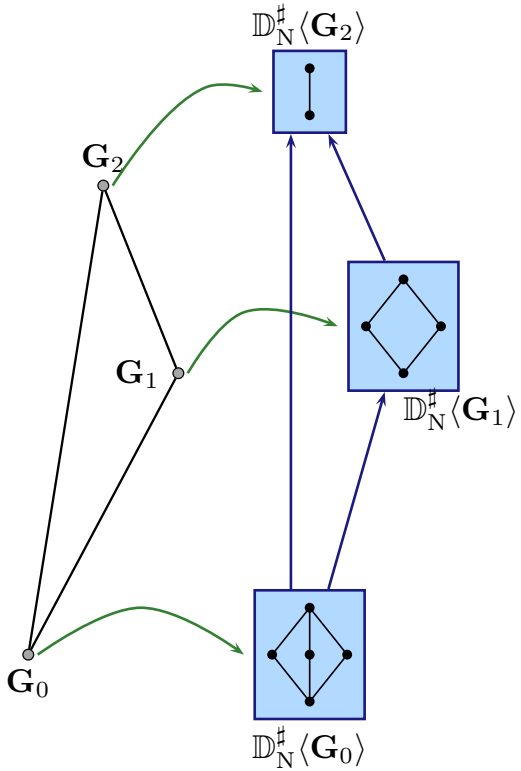
An assymetric product of abstract domains

Dependence of the numeric lattice on the graph

- **How to account for shape updates in numerics ?**
- The set of **numeric dimensions depends on the graph**

Cofibered domain structure [Venet, 1996]

- a **lattice $\mathbb{D}^\sharp_S$ of shape graphs**
- for each shape graph $\mathbf{G} \in \mathbb{D}^\sharp_S$, a distinct **numerical lattice** $\mathbb{D}^\sharp_N\langle \mathbf{G} \rangle$
- abstract values are **pairs** $(\mathbf{G}, \mathbf{N})$ where $\mathbf{N} \in \mathbb{D}^\sharp_N\langle \mathbf{G} \rangle$



Abstract operations in a cofibered abstract domain

Conversion operations :

When $\mathbf{G}_0 \sqsubseteq \mathbf{G}_1$, $\Pi_{0\to 1} : \mathbb{D}^\sharp_N\langle \mathbf{G}_0\rangle \to \mathbb{D}^\sharp_N\langle \mathbf{G}_1\rangle$ preserves concretization :

$$(\mathbf{G}_0 \sqsubseteq \mathbf{G}_1 \wedge \Pi_{0\to 1}(\mathbf{N}_0) \sqsubseteq \mathbf{N}_1) \Longrightarrow \gamma(\mathbf{G}_0, \mathbf{N}_0) \subseteq \gamma(\mathbf{G}_1, \mathbf{N}_1)$$

Abstract post-condition $\mathfrak{post}(\mathbf{G}_0, \mathbf{N}_0)$

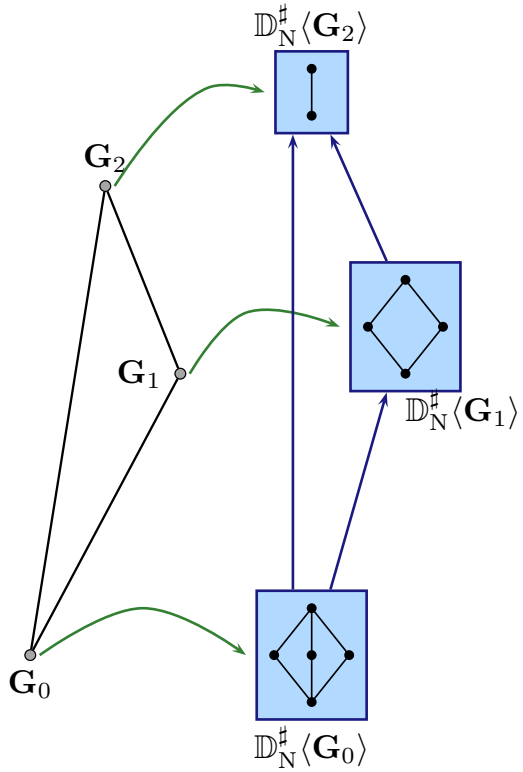
Typical algorithm :

1. evaluate $\mathbf{G}_0 = \mathfrak{post}_G(\mathbf{G}_0)$
2. compute $\mathbf{N}_1 = \mathfrak{post}_N(\Pi_{0\to 1}(\mathbf{N}_0))$

For some operations : **decomposition needed**

Widening $(\mathbf{G}_0, \mathbf{N}_0)\nabla(\mathbf{G}_1, \mathbf{N}_1)$

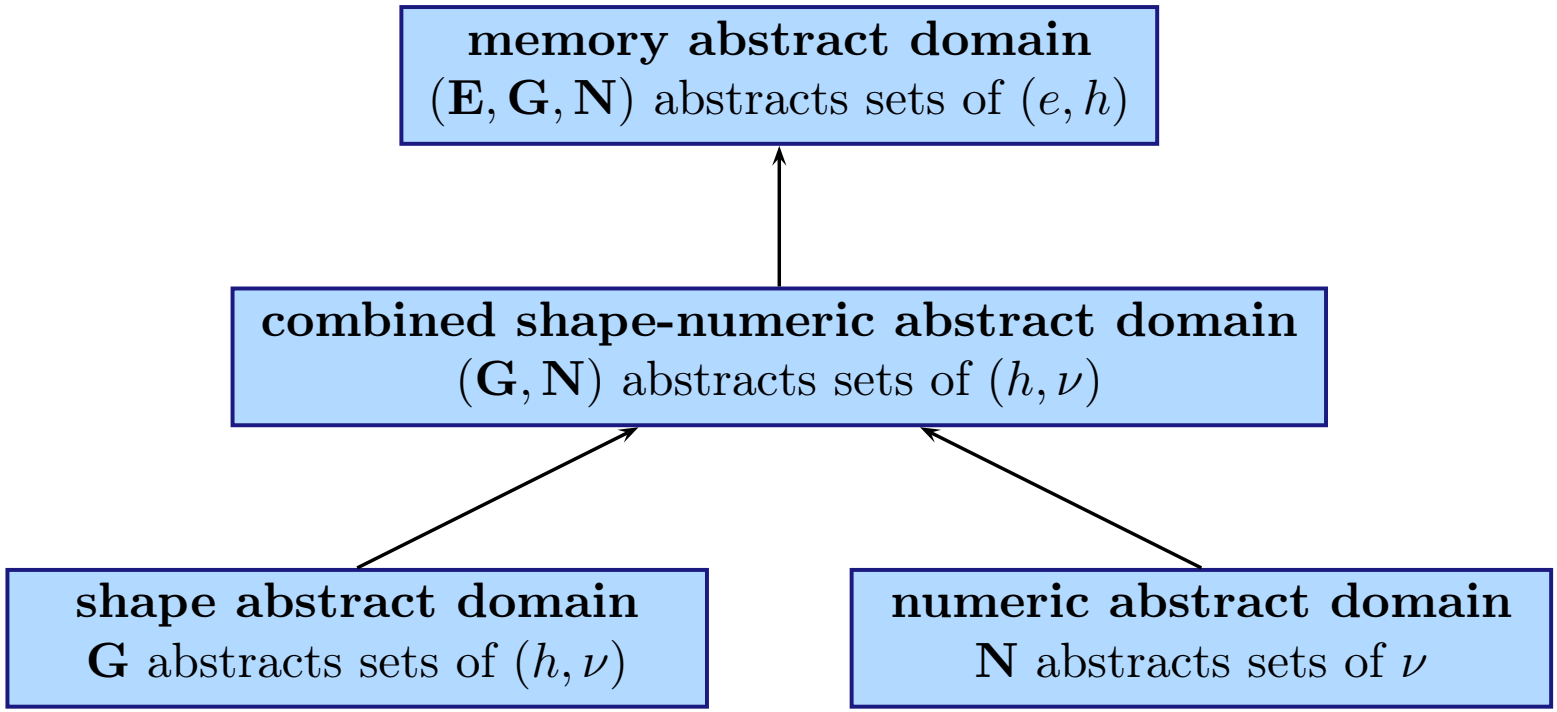
- evaluate $\mathbf{G}_2 = \mathbf{G}_0 \nabla \mathbf{G}_1$
- compute $\mathbf{N}_2 = \Pi_{0\to 2}(\mathbf{N}_0) \nabla \Pi_{1\to 2}(\mathbf{N}_1)$



Abstract domain layout

Modular structure

- **Each layer** accounts for one **aspect of the concrete states**
- **Each layer** boils down to a **module or functor in ML**



- Also a **disjunctive abstraction layer** (trace partitioning) for unfolding

Outline

Abstract operations

Denotational style abstract interpreter

- Concrete **denotational semantics** $[\![p]\!] : s \mapsto \mathcal{P}(s')$
- **Abstract semantics** $[\![p]\!]^\sharp(\mathbf{S}) = \mathbf{S}'$, computed by the analysis :

$$s \in \gamma(\mathbf{S}) \Longrightarrow [\![p]\!](s) \subseteq [\![p]\!]^\sharp(\mathbf{S})$$

Analysis by induction on the syntax using **domain operators**

$$\begin{array}{rcl}
[\![p_0; p_1]\!]^\sharp(\mathbf{S}) & = & [\![p_1]\!]^\sharp \circ [\![p_0]\!]^\sharp(\mathbf{S}) \\
[\![\ell = e]\!]^\sharp(\mathbf{S}) & = & \mathfrak{assign}(\ell, e, \mathbf{S}) \\
[\![\ell = \mathbf{malloc}(n)]\!]^\sharp(\mathbf{S}) & = & \mathfrak{alloc}(\ell, n, \mathbf{S}) \\
[\![\mathbf{free}(\ell)]\!]^\sharp(\mathbf{S}) & = & \mathfrak{free}(\ell, n, \mathbf{S}) \\
[\![\mathbf{if}(e)\ p_t\ \mathbf{else}\ p_f]\!]^\sharp(\mathbf{S}) & = & \left\{ \begin{array}{l} \mathfrak{join}([\![p_t]\!]^\sharp(\mathfrak{guard}(e, \mathbf{S})), \\ \qquad [\![p_f]\!]^\sharp(\mathfrak{guard}(e = \mathbf{false}, \mathbf{S}))) \end{array} \right. \\
[\![\mathbf{while}(e)p]\!]^\sharp(\mathbf{S}) & = & \mathfrak{guard}(e = \mathbf{false}, \mathbf{lfp}^\sharp_{\mathbf{S}} \mathcal{F}^\sharp) \\
\multicolumn{3}{c}{\text{where, } \mathcal{F}^\sharp : \mathbf{S}_0 \mapsto [\![p]\!]^\sharp(\mathfrak{guard}(e, \mathbf{S}_0))}
\end{array}$$

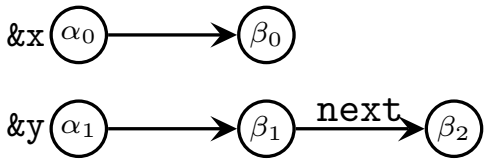
$\ldots$

Analysis of an assignment

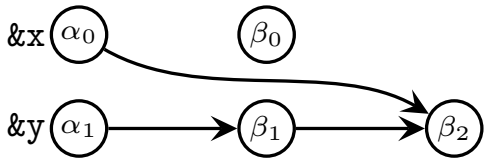
Steps for analyzing `x = y -> next` (local reasoning)

1. Evaluate **l-value** `x` into **points-to edge** $\alpha \mapsto \beta$
2. Evaluate **r-value** `y -> next` into **node** β'
3. Replace points-to edge $\alpha \mapsto \beta$ with **points-to edge** $\alpha \mapsto \beta'$

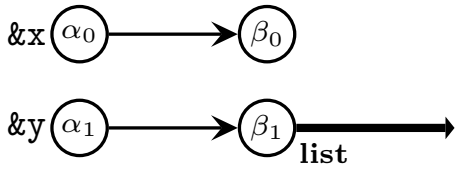
With pre-condition :



- Step 1 produces $\alpha_0 \mapsto \beta_0$
- Step 2 produces β_2
- End result :



With pre-condition :



- Step 1 produces $\alpha_0 \mapsto \beta_0$
- **Step 2 fails**

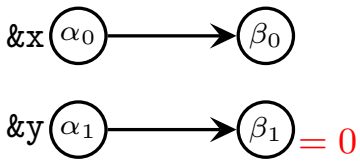
- **Too abstract abstract state**
- We need to **refine it**

Analysis of an assignment, with unfolding

Principle

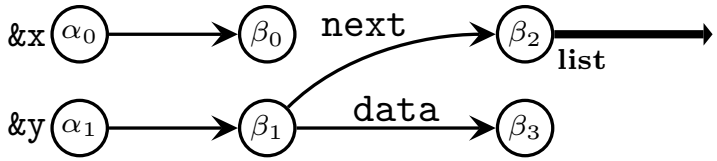
- We have $\gamma_S(\alpha \cdot \iota) = \bigcup\{\gamma_S(\mathbf{G}) \mid \alpha \cdot \iota \rightsquigarrow_{\text{unfold}} \mathbf{G}\}$
- Replace $\alpha \cdot \iota$ with a finite number of disjuncts and continue

Disjunct 1 :

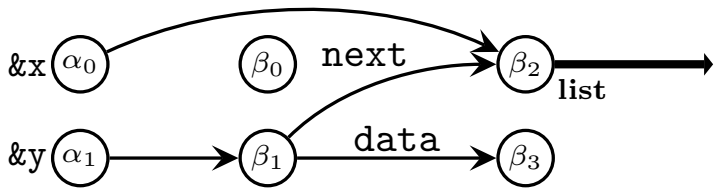


- Step 1 produces $\alpha_0 \mapsto \beta_0$
- **Step 2 fails :**
  Null pointer dereference !

Disjunct 2 :



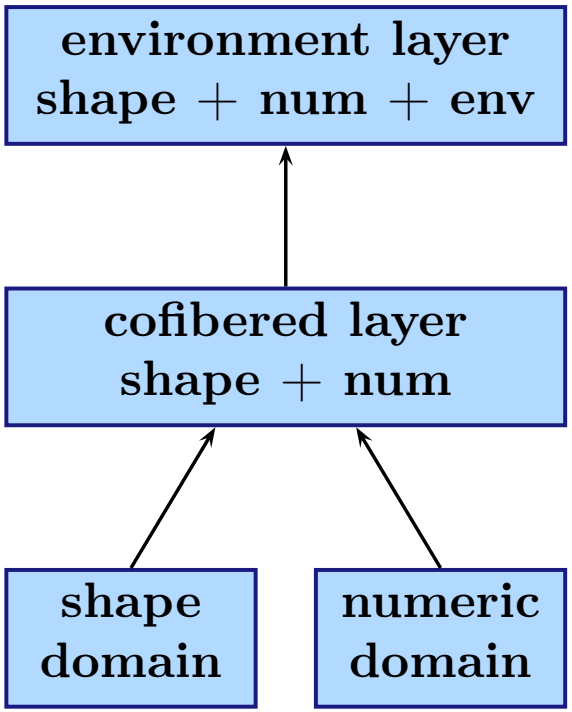
- Step 1 produces $\alpha_0 \mapsto \beta_0$
- Step 2 produces β_2
- **End result** :

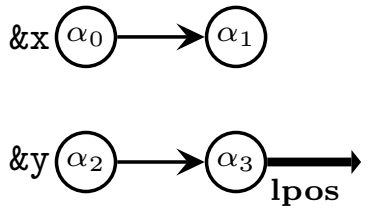


Case of a correct code...

Would be ruled out by a **condition** $y \neq 0$ i.e., $\beta_1 \neq 0$ in $\mathbb{D}^\sharp_N$

Analysis of an assignment



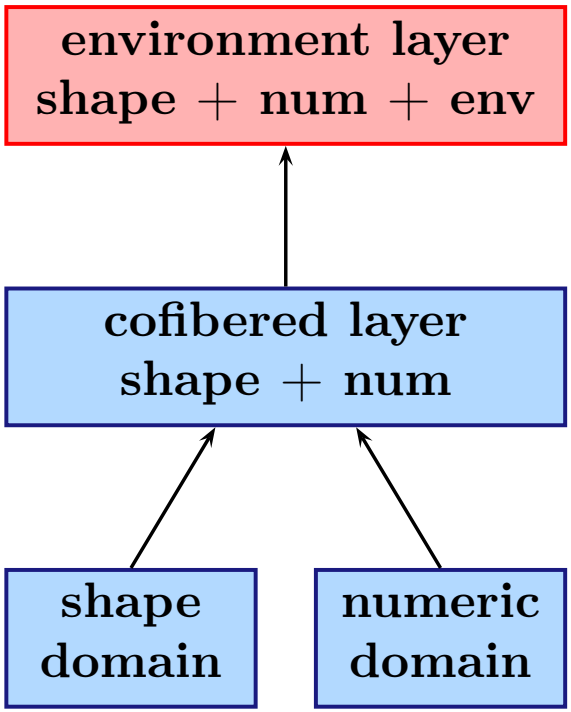


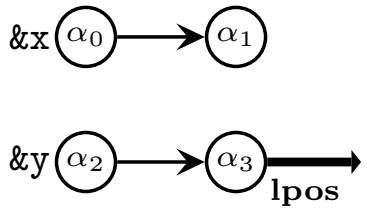
$$\mathbf{N} = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0}$$

$$\texttt{y -> d} = \texttt{x} + 1$$

Abstract post-condition ?

Analysis of an assignment





$$\mathbf{N} = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0}$$

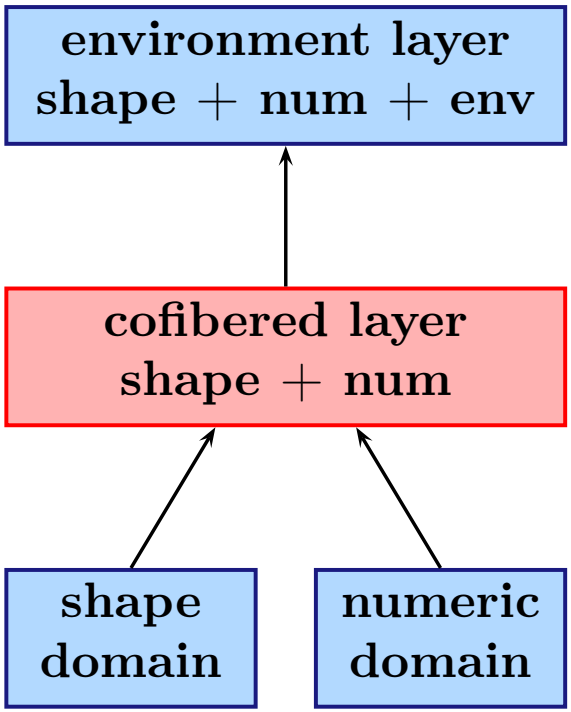
$$\texttt{y->d} = \texttt{x} + 1 \quad \Rightarrow \quad (\star\alpha_2) \cdot \texttt{d} = (\star\alpha_0) + 1$$

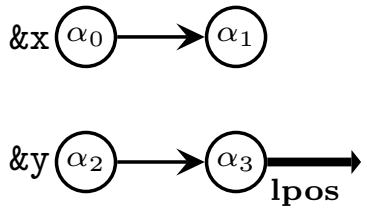
Abstract post-condition ?

Stage 1 : environment resolution

- replaces x with $\star\mathbf{E}(\texttt{x})$

Analysis of an assignment





$$\mathbf{N} = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0}$$

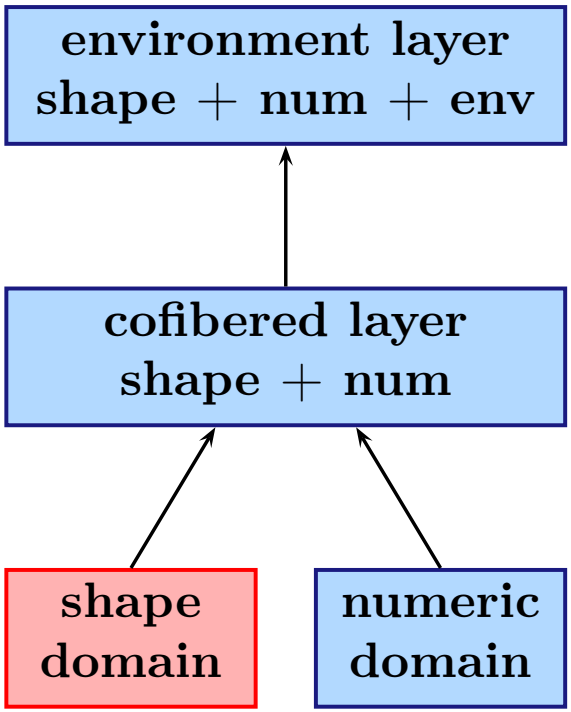
$$(\star\alpha_2) \cdot \mathtt{d} = (\star\alpha_0) + 1$$

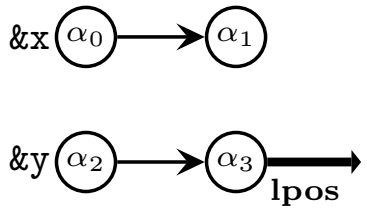
Abstract post-condition ?

Stage 2 : propagate into the shape + numerics domain

- only symbolic nodes appear

Analysis of an assignment





$$\mathbf{N} = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0}$$

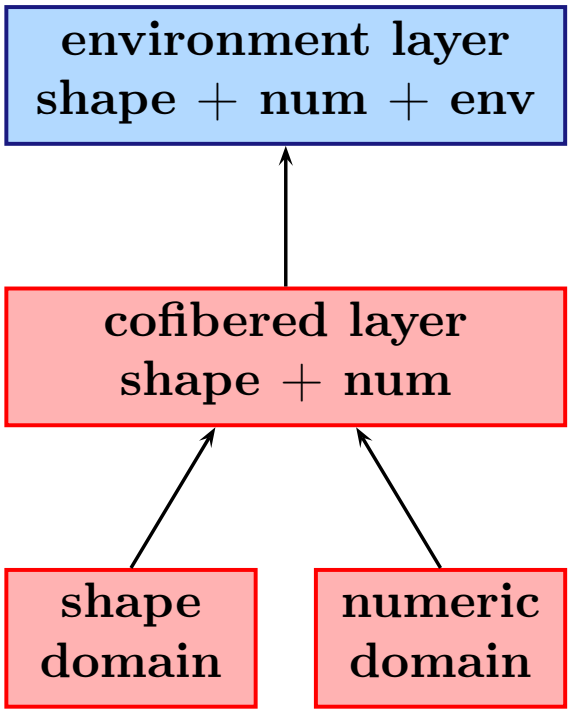
$$(\star\alpha_2) \cdot \mathtt{d} = (\star\alpha_0) + 1$$

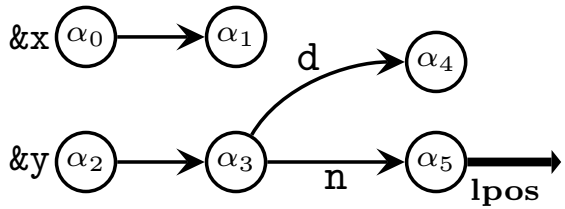
Abstract post-condition ?

Stage 3 : resolve cells in the shape graph abstract domain

- $\star\alpha_0$ evaluates to α_1 ; $\star\alpha_2$ evaluates to α_3
- $(\star\alpha_2) \cdot \mathtt{d}$ **fails to evaluate** : **no points-to out of** α_3

Analysis of an assignment





$$\mathbf{N} = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0} \wedge \alpha_4 \geq 0$$

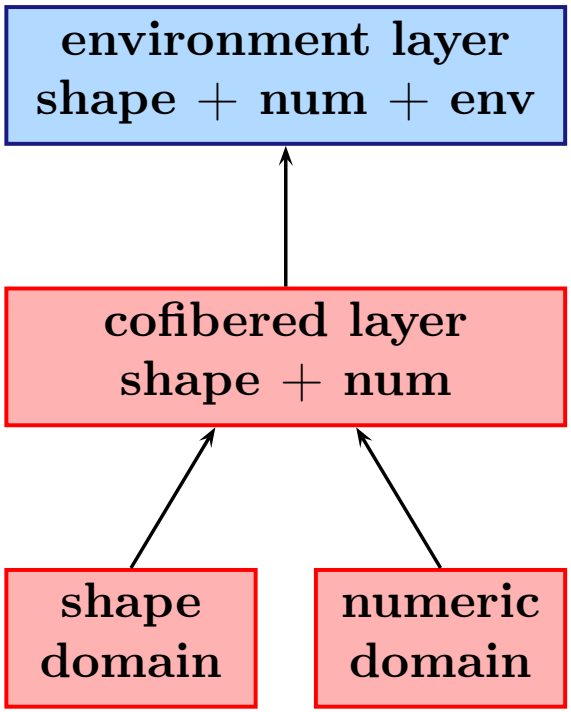
$$(\star\alpha_2) \cdot \mathtt{d} = (\star\alpha_0) + 1$$

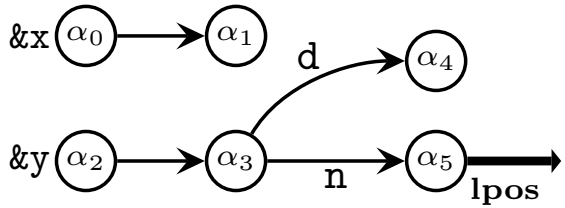
Abstract post-condition ?

Stage 4 : unfolding

- locally materialize $\alpha_3 \cdot$ **lpos** ; update keys / relations in the numerics
- l-value $(\star\alpha_2) \cdot \mathtt{d}$ **now evaluates into edge** α_3 -> d $\mapsto \alpha_4$

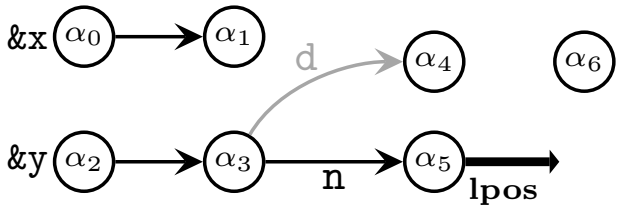
Analysis of an assignment





$$\mathbf{N} = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0} \wedge \alpha_4 \geq 0$$

create node α_6

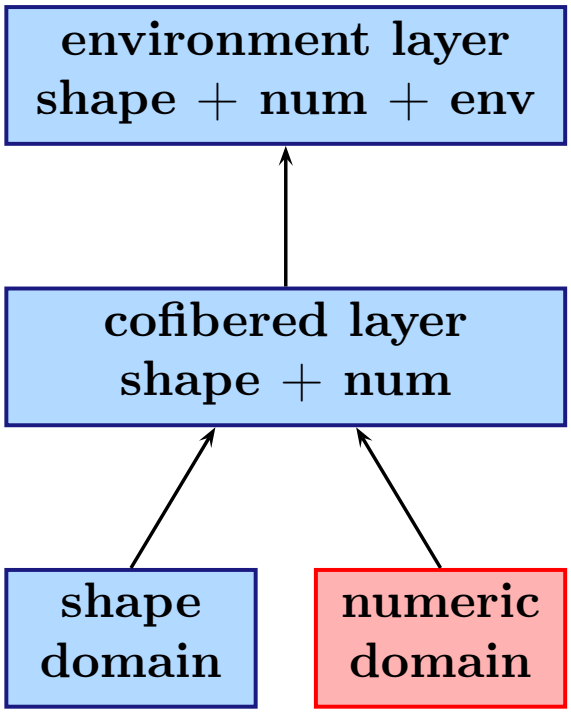


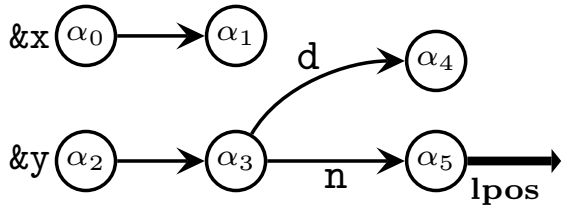
$$\mathbf{N} = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0} \wedge \alpha_4 \geq 0 \wedge \alpha_6 \geq 1$$

Stage 5 : create a new node

- new node α_6 denotes a new value
  will store the new value

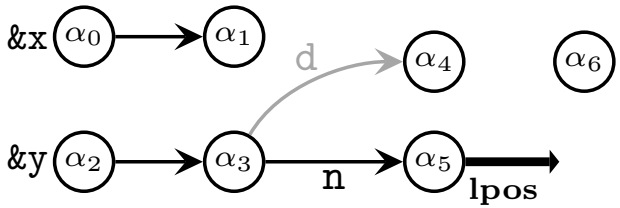
Analysis of an assignment





$$\mathbf{N} = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0} \wedge \alpha_4 \geq 0$$

$\alpha_6 \leftarrow \alpha_1 + 1$ in numerics

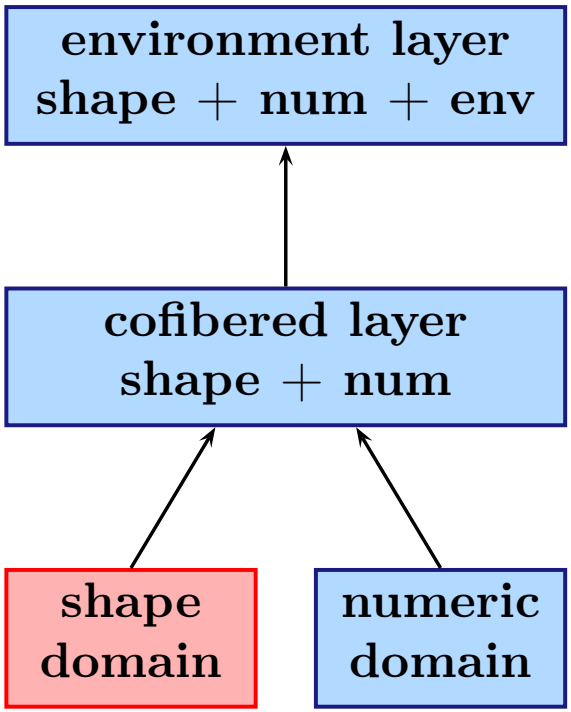


$$\mathbf{N} = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0} \wedge \alpha_4 \geq 0 \wedge \alpha_6 \geq 1$$

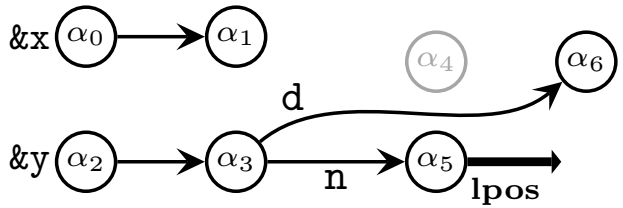
Stage 6 : perform numeric assignment

- numeric assignment **completely ignores pointer structures** to the new node

Analysis of an assignment



mutate $(\alpha_3 \cdot \mathtt{d}) \mapsto \alpha_4$ into α_6



$\mathbf{N} = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0} \wedge \alpha_4 \geq 0 \wedge \alpha_6 \geq 1$

Stage 7 : perform the update in the graph

- classic **strong update** in a pointer aware domain
- symbolic node α_4 becomes redundant and can be removed

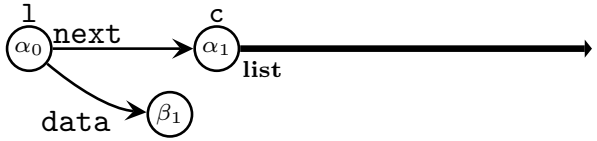
Widening I : need for a folding operation

- A minimal example...

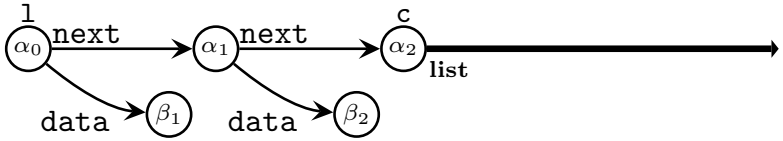
```
assume(l points to a list)
c = l;
while(c ≠ NULL){
  c = c → next;
}
```

- **First iterates** in the loop :
  - at **iteration 0** (before entering the loop) :

  - at **iteration 1** :



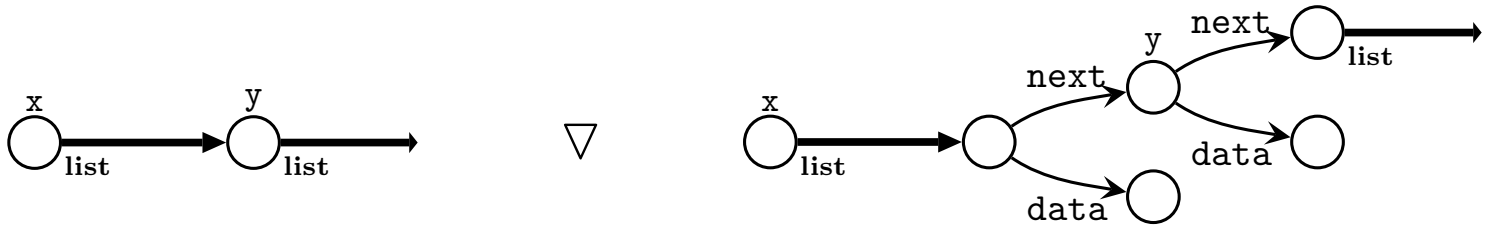
  - at **iteration 2** :



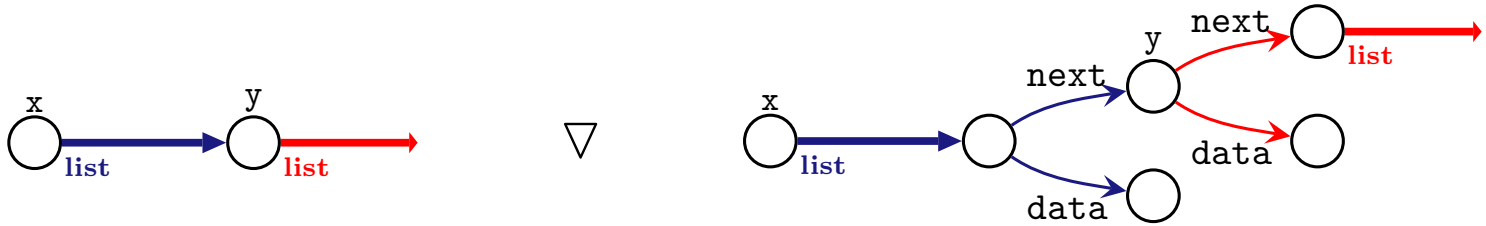
- How to guarantee the **termination** of the analysis ?
- How to **introduce segment edges** / perform **abstraction** ?

Widening II : algorithm overview

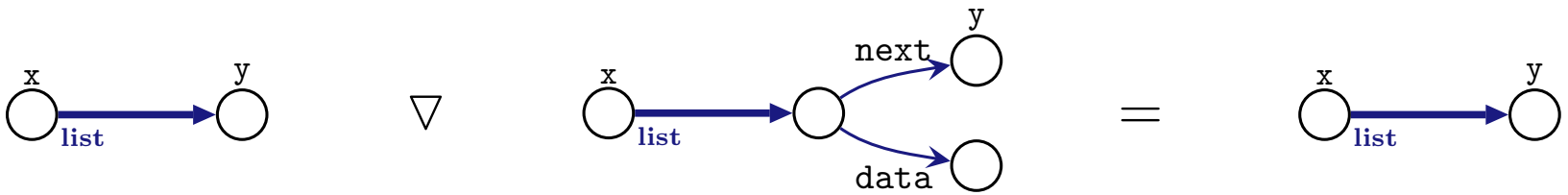
- Takes **two abstract values as inputs**



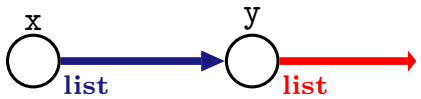
- **Region matching** (non unique choice : use of **strategies**)



- **Semantic rules** for **per region weakening**

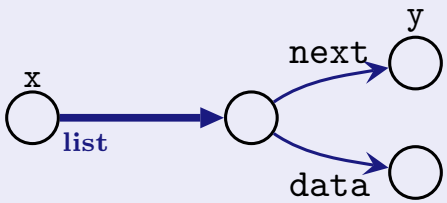


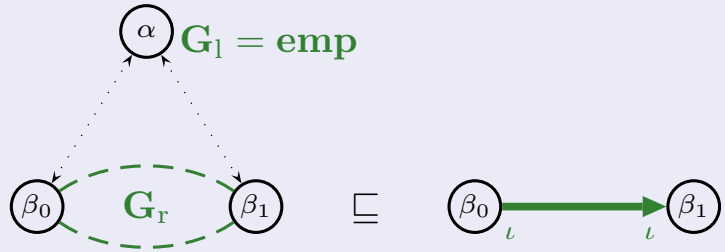
- **Widening :**



Widening III : an example of a widening meta-rule

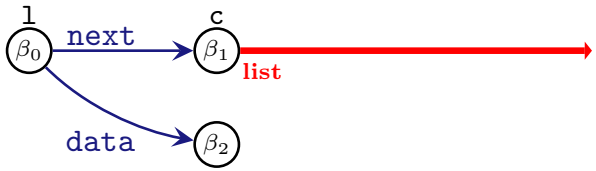
Segment introduction meta rule (*for all* ι)

if 

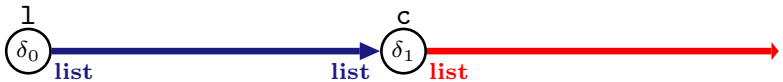
then $\mathbf{G}_\mathrm{l} \nabla \mathbf{G}_\mathrm{r} = \mathbf{G}_\mathrm{r} \sqsubseteq$ 

- **Application to list traversal**, at the end of iteration 1 :
  - **before iteration 0 :**

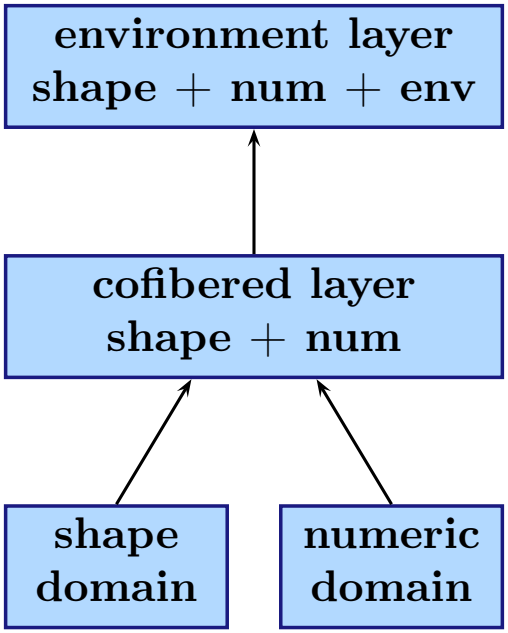
  - **end of iteration 0 :**

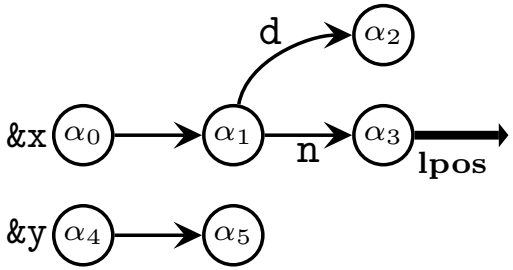


  - **join, before iteration 1 :**

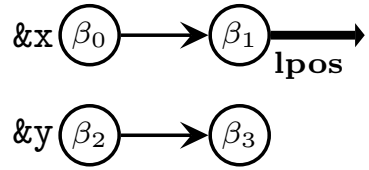


Widening / join in the combined domain



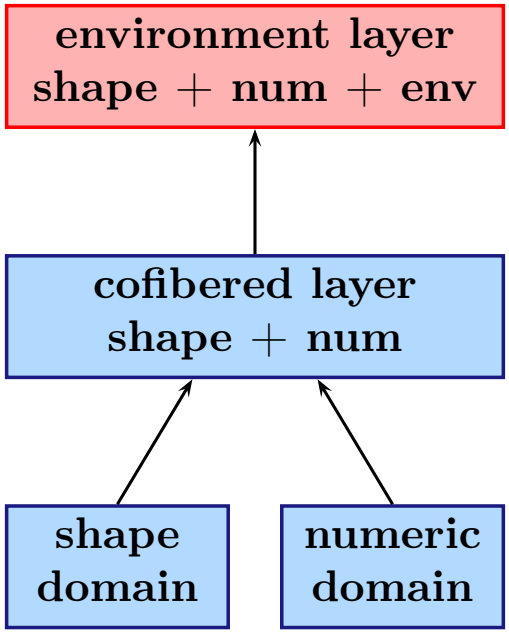


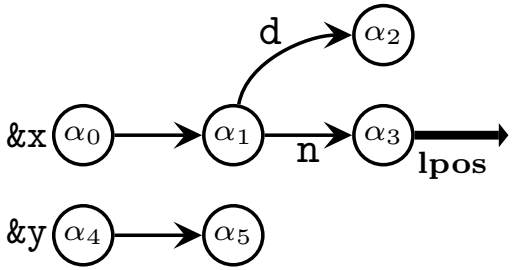
$\mathbf{N} = \alpha_2 \geq \alpha_5 \geq 2$



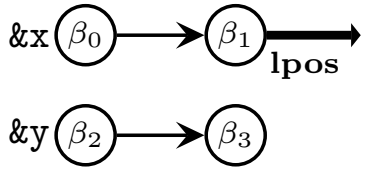
$\mathbf{N}' = \beta_3 \geq 1$

Widening / join in the combined domain

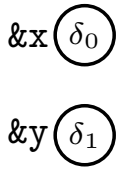




$\mathbf{N} = \alpha_2 \geq \alpha_5 \geq 2$



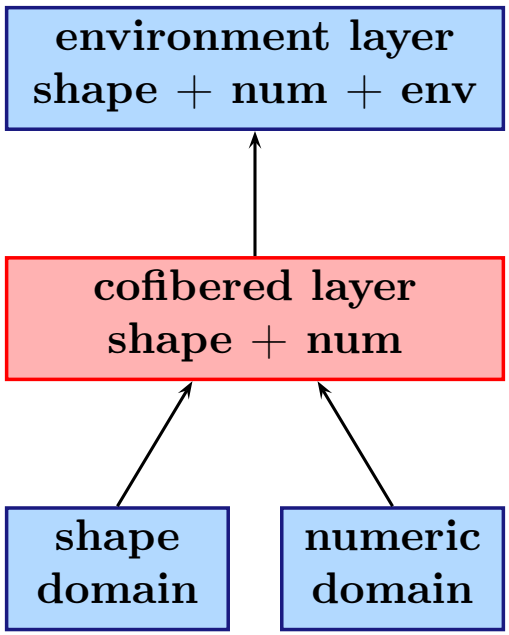
$\mathbf{N}' = \beta_3 \geq 1$

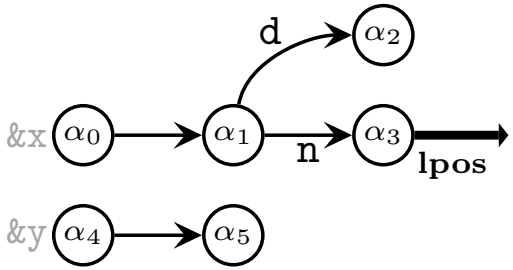


Stage 1 : abstract environment

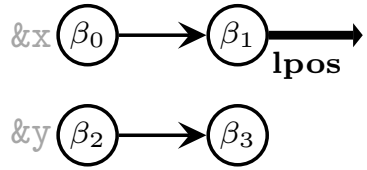
- compute new abstract environment and initial node relation
  e.g., α_0, β_0 both denote &x

Widening / join in the combined domain

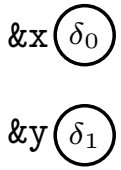




$\mathbf{N} = \alpha_2 \geq \alpha_5 \geq 2$



$\mathbf{N}' = \beta_3 \geq 1$

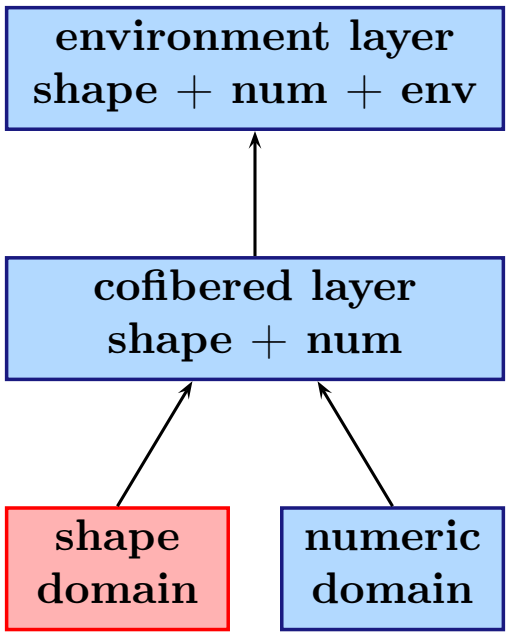


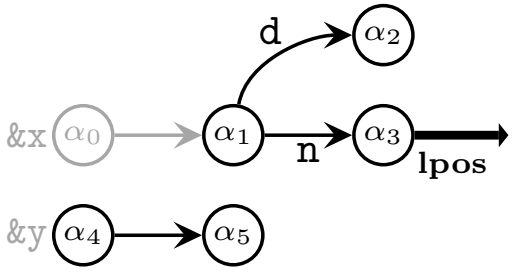
Stage 2 : join in the "cofibered" layer

operations to perform :

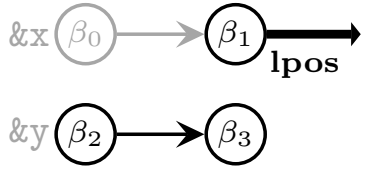
1. compute the join in the graph
2. convert value abstractions, and join the resulting lattice

Widening / join in the combined domain

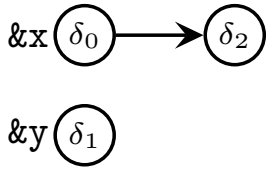




$\mathbf{N} = \alpha_2 \geq \alpha_5 \geq 2$



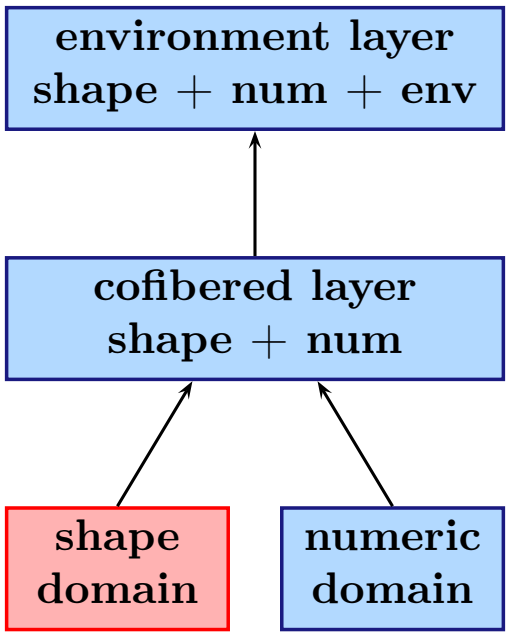
$\mathbf{N}' = \beta_3 \geq 1$

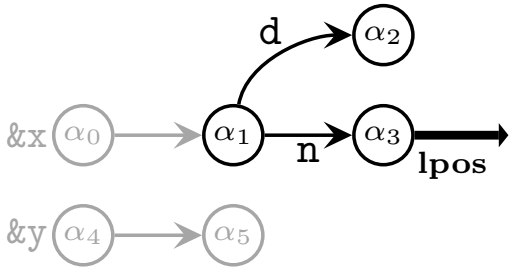


Stage 2 : graph join

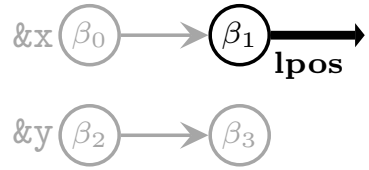
- apply local join rules
  ex : **points-to matching**, **weakening to inductive**...
- incremental algorithm

Widening / join in the combined domain

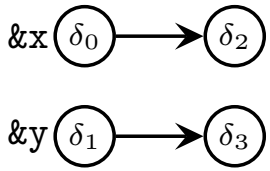




$\mathbf{N} = \alpha_2 \geq \alpha_5 \geq 2$



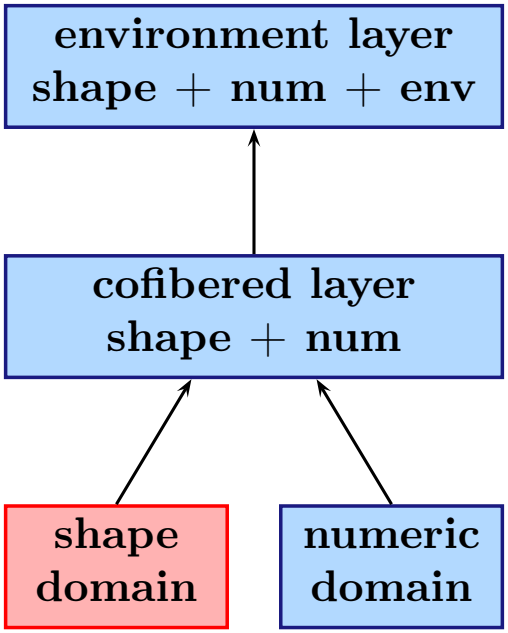
$\mathbf{N}' = \beta_3 \geq 1$

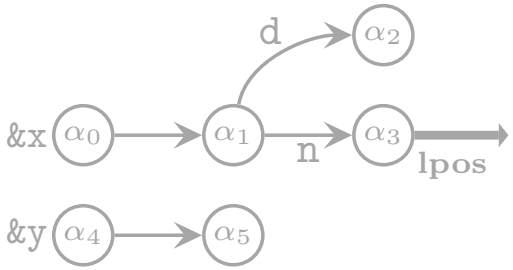


Stage 2 : graph join

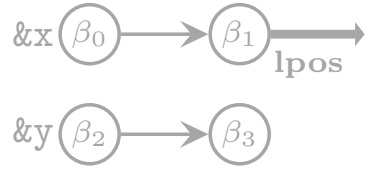
- apply local join rules
  ex : **points-to matching**, **weakening to inductive**...
- incremental algorithm

Widening / join in the combined domain

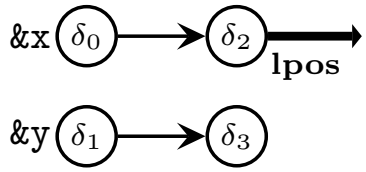




$\mathbf{N} = \alpha_2 \geq \alpha_5 \geq 2$



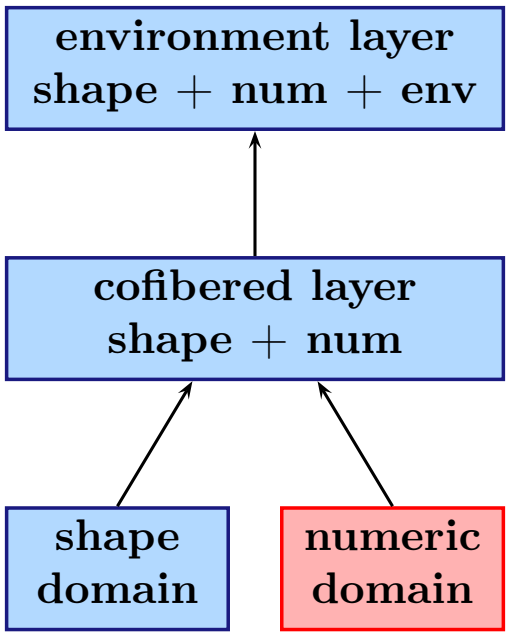
$\mathbf{N}' = \beta_3 \geq 1$

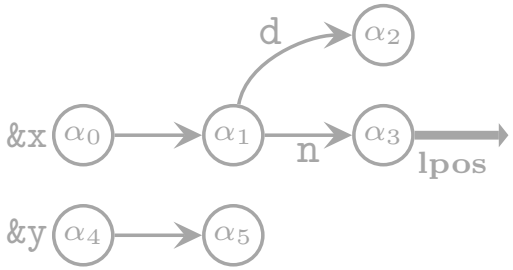


Stage 2 : graph join

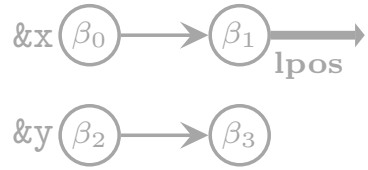
- apply local join rules
  ex : **points-to matching**, **weakening to inductive**...
- incremental algorithm

Widening / join in the combined domain

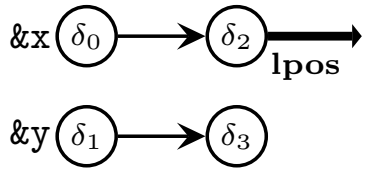




$\mathbf{N} = \alpha_2 \geq \alpha_5 \geq 2$



$\mathbf{N}' = \beta_3 \geq 1$

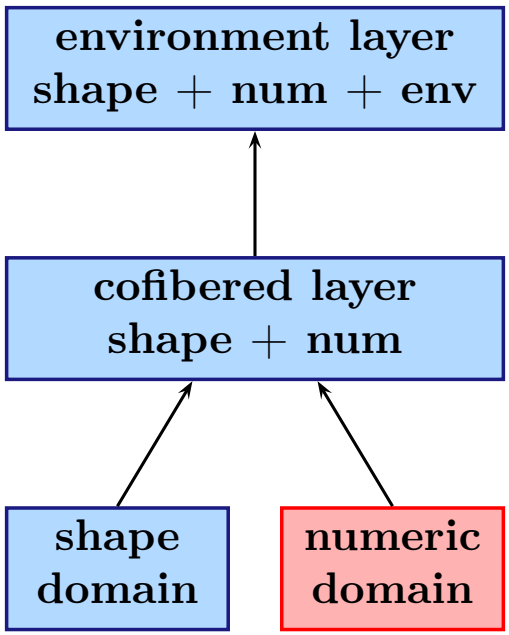


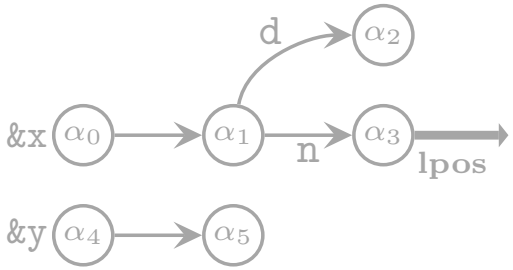
$\mathbf{N}^{\sqcup} = [\delta_3 \geq 2] \sqcup [\delta_3 \geq 1]$

Stage 3 : conversion function application in numerics

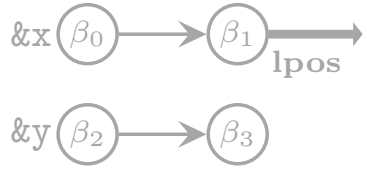
- remove nodes that were abstracted away
- rename other nodes

Widening / join in the combined domain

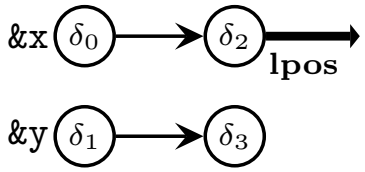




$\mathbf{N} = \alpha_2 \geq \alpha_5 \geq 2$



$\mathbf{N}' = \beta_3 \geq 1$



$\mathbf{N}^{\sqcup} = [\delta_3 \geq 1]$

Stage 4 : join in the numeric domain

- apply $\sqcup$ for regular join, $\triangledown$ for a widening

Concluding remarks

Advantages of modular abstract domain design

- **Intuitive** : follows the semantics structure
- **Easier** to **design**, **prove** and **implement**

Domain operators :

- shape abstract domain **reduced product** : **conjunction**
  Antoine Toubhans et al, VMCAI 2013
- **hierarchical shape abstraction** : **sub-memory inside a region**
  Pascal Sotin et al, APLAS 2012
- **spatial combination** : **different abstractions** for disjoint regions
  Antoine Toubhans (in progress) : better scalability / extensibility