Construction of abstract domains for heterogeneous memory properties

Xavier Rival
Joint work with Bor-Yuh Evan Chang,
Huisong Li, Jiangchao Liu, Pascal Sotin, Antoine Toubhans

CNRS, ENS and INRIA

TUM, February, 2015

Memory abstraction difficulties

Memory abstraction

- **Abstraction** of $\mathcal{P}($ **Environment** $\times$ **Stack** $\times$ **Heap** $)$
- **Operations to analyze**: C statements
  including **pointer arithmetic**, **memory management**...
- **Operations to verify**: **memory safety**, **structure preservation**...

Difficulties:

1. **Complex concrete semantics**: memory states, pointers...
2. **Huge variety of properties to abstract**:
   - **linked structures**: lists, trees...
   - **arrays**, **buffers**, **strings**...
   - structures involving **relations** (sorted lists, balanced trees) or **sharing**
   - **combination** of numerical and memory properties
3. **Expensive analysis algorithms** to infer such abstractions

Existing approaches

Many existing analyses for specific structures:

- **Inductive structures** (lists, trees, ASTs...):
  TVLA, **separation logic** based static analyses...
- **Complex aggregates** (unions and structs):
  precise **abstractions of blocks**, account for alignment and sizes
- **Arrays**:
  smashing based abstractions,
  partitioning based abstractions (array split into several regions)
- **Buffers** and **strings**:
  abstractions of **size** and **positions of zeros**

But abstracting stores with heterogeneous structures remains hard

Main ideas of the MemCAD project

Towards a library of abstract domains

Common abstract domain interfaces:

- Concretization to the **same domain**
- **Similar** transfer functions / lattice operations

A few basic abstract domains (implemented as **ML modules**):

- **Lightweight** abstraction of **fixed size blocks**
- **Parametric** abstraction for **inductive structures**...

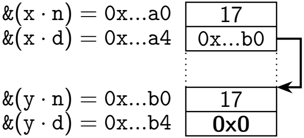
Combination operations (implemented as **ML functors**):

- **Product** with a **value** abstraction (e.g., numeric)
- **Separating product** of memory abstraction
- **Reduced, non separating product** of memory abstraction...

- Combinations of abstract domains can be tuned for precision/efficiency
- Abstract domains are not analyzer specific

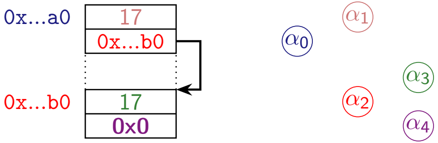
A generic memory abstraction step by step

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



A generic memory abstraction step by step

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

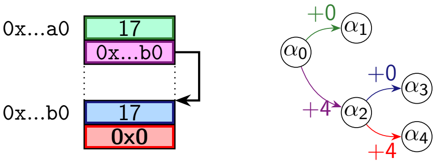


$\nu(\alpha_0) = \texttt{0x...a0}$
$\nu(\alpha_1) = 17$
$\nu(\alpha_2) = \texttt{0x...b0}$
$\nu(\alpha_3) = 17$
$\nu(\alpha_4) = \mathbf{0x0}$

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

A generic memory abstraction step by step

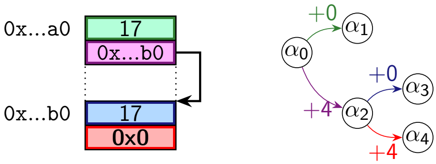
- **Concrete memory states**
- **Abstraction of values into symbolic variables** / nodes
- **Abstraction of regions into points-to edges**



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

A generic memory abstraction step by step

- **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**

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

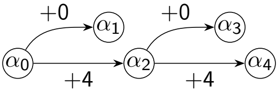
valuation ν plays an important role in the combined domain

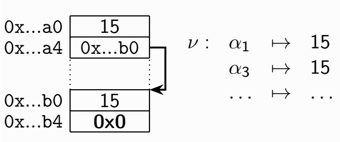
Adding value information (here, numeric)

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

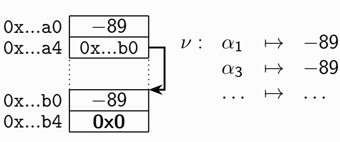
Example: all lists of length 2, with equal data fields

- **Memory abstraction**:





$$\nu : \begin{array}{ccc} \alpha_1 & \mapsto & 15 \\ \alpha_3 & \mapsto & 15 \\ \ldots & \mapsto & \ldots \end{array}$$

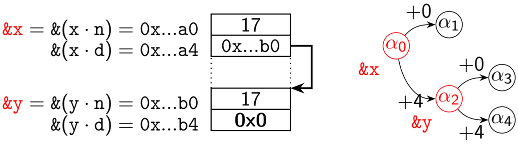


$$\nu : \begin{array}{ccc} \alpha_1 & \mapsto & -89 \\ \alpha_3 & \mapsto & -89 \\ \ldots & \mapsto & \ldots \end{array}$$

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

Adding environment abstraction

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



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

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

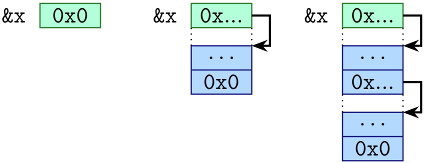
Abstraction of environments:

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

Adding summarization

Example:

set of all lists of any length

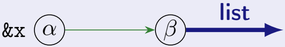


List inductive predicate

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

well-founded predicate

Inductive summary predicates



Concretization based on **unfolding** and **least-fixpoint**:

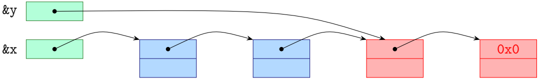
- $\rightsquigarrow$ replaces **an $\alpha \cdot$ list predicate** with **one of its premises**
- $\gamma(\mathbf{G}, \mathbf{N}) = \bigcup \{\gamma(\mathbf{G}', \mathbf{N}') \mid (\mathbf{G}, \mathbf{N}) \rightsquigarrow (\mathbf{G}', \mathbf{N}')\}$
  (all inductive predicates are required to be well-founded)

Summarization of incomplete structures

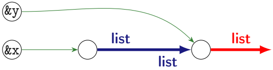
Structure fragments

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

- Example with **lists**:

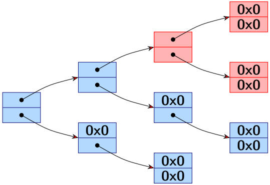


can be abstracted by:



- Works also for **trees**:

abstracts

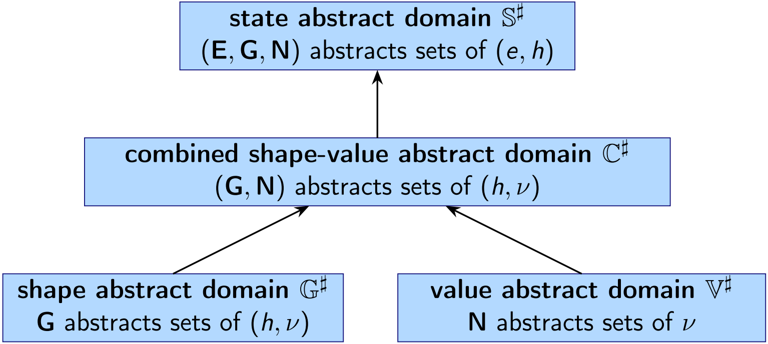


Overall abstract domain structure

Modular structure

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

Makes the implementation **a lot simpler**



state abstract domain $\mathbb{S}^\sharp$
$(\mathbf{E}, \mathbf{G}, \mathbf{N})$ abstracts sets of (e, h)

combined shape-value abstract domain $\mathbb{C}^\sharp$
$(\mathbf{G}, \mathbf{N})$ abstracts sets of (h, ν)

shape abstract domain $\mathbb{G}^\sharp$
$\mathbf{G}$ abstracts sets of (h, ν)

value abstract domain $\mathbb{V}^\sharp$
$\mathbf{N}$ abstracts sets of ν

Outline

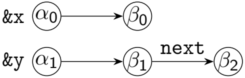
1. Memory abstraction
2. A few memory abstractions, with value information
3. Abstract operations
4. Instantiations
5. Separating product of abstractions
6. Reduced product of abstractions
7. Abstraction of arrays and hierarchical abstraction
8. Project status

Analysis of an assignment in the graph domain

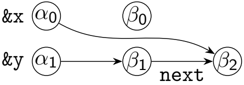
Steps for analyzing `x = y -> next` (local reasoning)

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** $\alpha \mapsto \beta'$

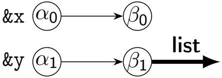
With pre-condition:



- 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**

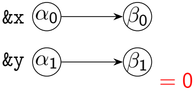
- Abstract state **too abstract**
- We need to **refine it**

Analysis of an assignment, with unfolding

Principle

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

Disjunct 1:

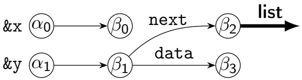


- Step 1 produces $\alpha_0 \mapsto \beta_0$
- **Step 2 fails:**
  Null pointer dereference !

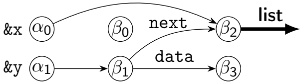
Case of a correct code...

Would be ruled out by a **condition** $\mathtt{y} \neq 0$ i.e., $\beta_1 \neq 0$ in numerics

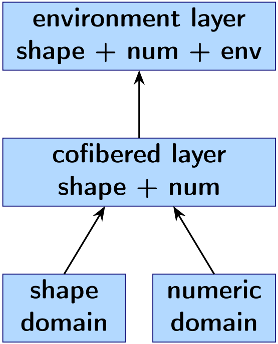
Disjunct 2:



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



Analysis of an assignment in the combined domain

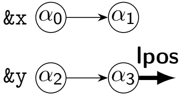


environment layer
shape + num + env

cofibered layer
shape + num

shape domain

numeric domain

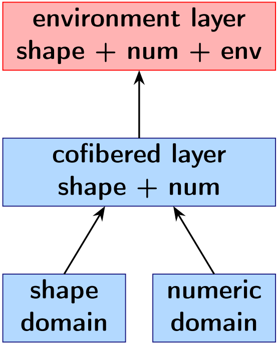


$$\mathbf{N} = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0}$$

```
y -> d = x + 1
```

Abstract post-condition ?

Analysis of an assignment in the combined domain

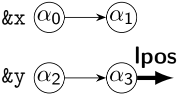


environment layer
shape + num + env

cofibered layer
shape + num

shape domain

numeric domain



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

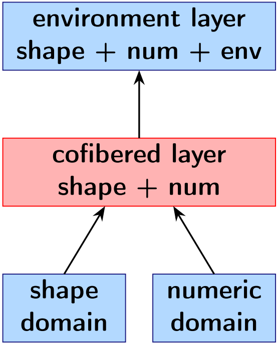
$$\mathtt{y\text{-}>d} = \mathtt{x} + 1 \quad \Rightarrow \quad (\star\alpha_2)\cdot\mathtt{d} = (\star\alpha_0) + 1$$

Abstract post-condition ?

Stage 1: environment resolution

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

Analysis of an assignment in the combined domain

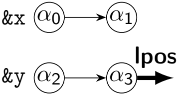


environment layer
shape + num + env

cofibered layer
shape + num

shape domain

numeric domain



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

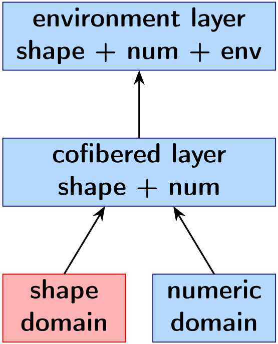
$$(\star\alpha_2) \cdot \mathtt{d} = (\star\alpha_0) + 1$$

Abstract post-condition ?

Stage 2: propagate into the shape + numerics domain

- only symbolic nodes appear

Analysis of an assignment in the combined domain

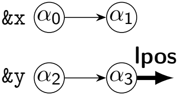


environment layer
shape + num + env

cofibered layer
shape + num

shape
domain

numeric
domain



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

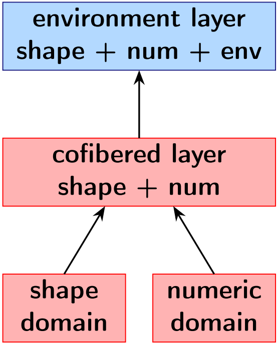
$$(\star\alpha_2) \cdot \mathtt{d} = (\star\alpha_0) + 1$$

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

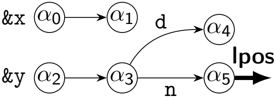
Analysis of an assignment in the combined domain


environment layer
shape + num + env

cofibered layer
shape + num

shape domain

numeric domain



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

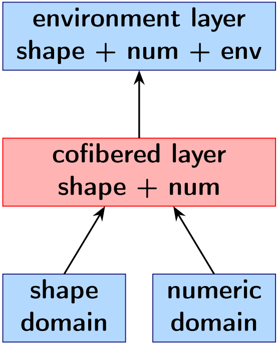
$$(\star\alpha_2) \cdot \mathtt{d} = (\star\alpha_0) + 1$$

Abstract post-condition ?

Stage 4: unfolding (several steps, skipped here)

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

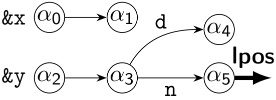
Analysis of an assignment in the combined domain


environment layer
shape + num + env

cofibered layer
shape + num

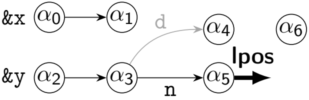
shape
domain

numeric
domain



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

create node α_6

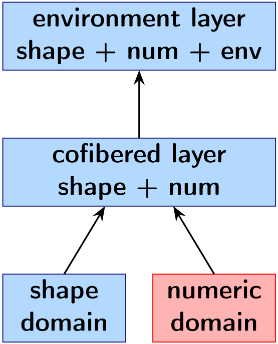


$\mathbf{N} = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0} \wedge \alpha_4 \geq 0$

Stage 5: create a new node

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

Analysis of an assignment in the combined domain

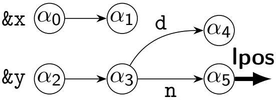


environment layer
shape + num + env

cofibered layer
shape + num

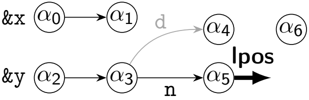
shape domain

numeric domain



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

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

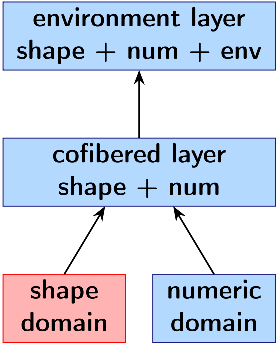


$$\mathbf{N} = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0} \wedge \alpha_4 \geq 0 \wedge \alpha_6 \geq 1$$

Stage 6: perform numeric assignment

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

Analysis of an assignment in the combined domain



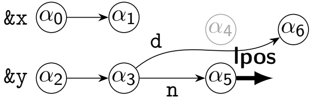
environment layer
shape + num + env

cofibered layer
shape + num

shape domain

numeric domain

$$\text{mutate } (\alpha_3 \cdot \mathtt{d}) \mapsto \alpha_4 \text{ into } \alpha_6$$



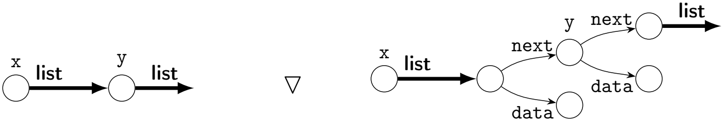
$$\mathsf{N} = \alpha_1 \geq 0 \wedge \alpha_3 \neq \mathbf{0x0} \wedge \alpha_4 \geq 0 \wedge \alpha_6 \geq 1$$

Stage 7: perform the update in the graph

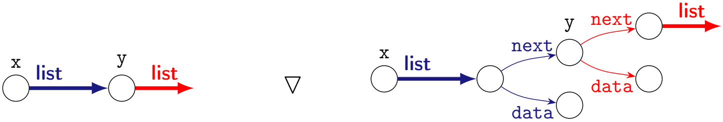
- classic **strong update** in a pointer aware domain
- symbolic node α_4 becomes redundant and can be removed

Widening in the graph domain

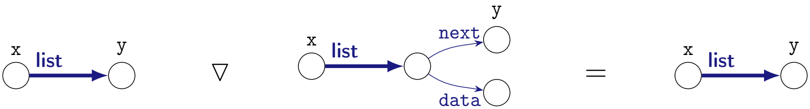
- Takes **two abstract values as inputs**



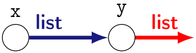
- **Region matching** (non unique choice: use of **strategies**)



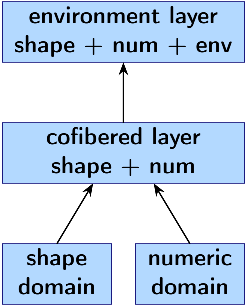
- **Semantic rules** for **per region weakening**



- **Widening:**



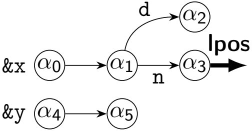
Widening / join in the combined domain


environment layer
shape + num + env

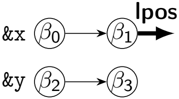
cofibered layer
shape + num

shape
domain

numeric
domain

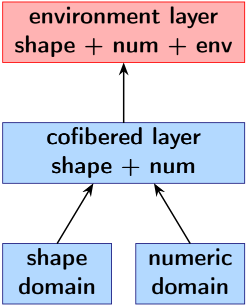


$\mathsf{N} = \alpha_2 \geq \alpha_5 \geq 2$



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

Widening / join in the combined domain

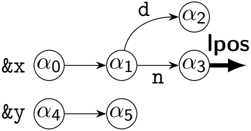


environment layer
shape + num + env

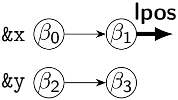
cofibered layer
shape + num

shape domain

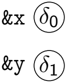
numeric domain



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



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

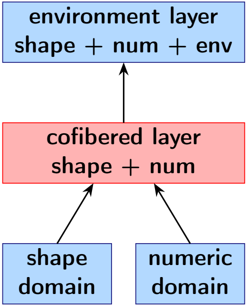


$$\begin{array}{rcl}\delta_0 & \equiv & (\alpha_0, \beta_0) \\ \delta_1 & \equiv & (\alpha_4, \beta_2)\end{array}$$

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

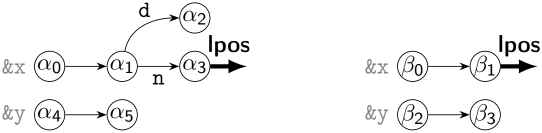


environment layer
shape + num + env

cofibered layer
shape + num

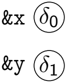
shape domain

numeric domain



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

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



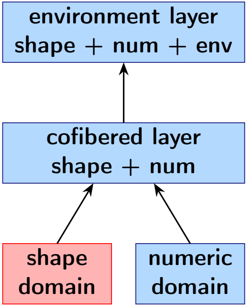
$$\begin{array}{lcl} \delta_0 & \equiv & (\alpha_0, \beta_0) \\ \delta_1 & \equiv & (\alpha_4, \beta_2) \end{array}$$

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

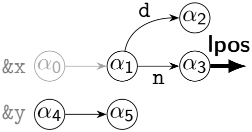


environment layer
shape + num + env

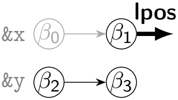
cofibered layer
shape + num

shape domain

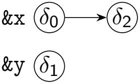
numeric domain



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



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

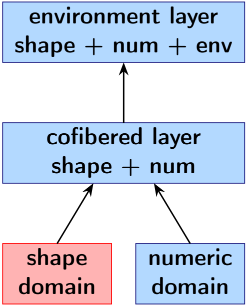


$$\begin{aligned} \delta_0 &\equiv (\alpha_0, \beta_0) \\ \delta_1 &\equiv (\alpha_4, \beta_2) \\ \delta_2 &\equiv (\alpha_1, \beta_1) \end{aligned}$$

Stage 2: graph join

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

Widening / join in the combined domain

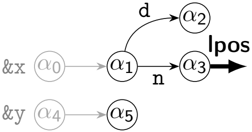


environment layer
shape + num + env

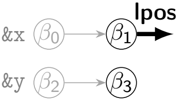
cofibered layer
shape + num

shape domain

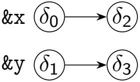
numeric domain



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



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

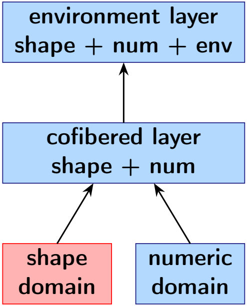


$$\begin{array}{lcl} \delta_0 & \equiv & (\alpha_0, \beta_0) \\ \delta_1 & \equiv & (\alpha_4, \beta_2) \\ \delta_2 & \equiv & (\alpha_1, \beta_1) \\ \delta_3 & \equiv & (\alpha_5, \beta_3) \end{array}$$

Stage 2: graph join

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

Widening / join in the combined domain

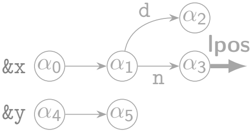


environment layer
shape + num + env

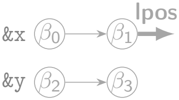
cofibered layer
shape + num

shape domain

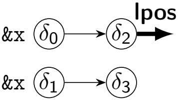
numeric domain



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



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

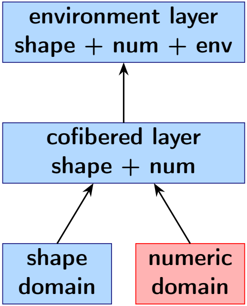


$$\begin{array}{lcl} \delta_0 & \equiv & (\alpha_0, \beta_0) \\ \delta_1 & \equiv & (\alpha_4, \beta_2) \\ \delta_2 & \equiv & (\alpha_1, \beta_1) \\ \delta_3 & \equiv & (\alpha_5, \beta_3) \end{array}$$

Stage 2: graph join

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

Widening / join in the combined domain

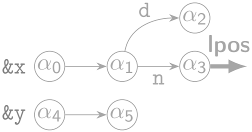


environment layer
shape + num + env

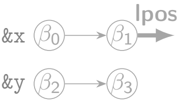
cofibered layer
shape + num

shape domain

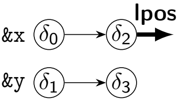
numeric domain



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



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



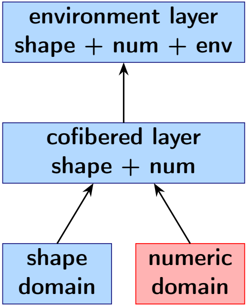
$$\begin{array}{lcl}\delta_0 & \equiv & (\alpha_0, \beta_0)\\ \delta_1 & \equiv & (\alpha_4, \beta_2)\\ \delta_2 & \equiv & (\alpha_1, \beta_1)\\ \delta_3 & \equiv & (\alpha_5, \beta_3)\end{array}$$

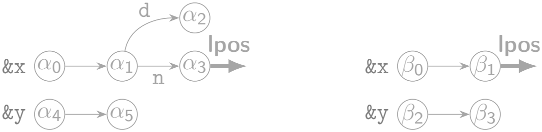
$$\mathsf{N}^{\sqcup} = [\delta_3 \geq 2] \sqcup [\delta_3 \geq 1]$$

Stage 3: conversion function application in numerics

- remove nodes that were abstracted away
- rename other nodes

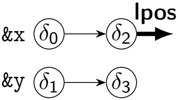
Widening / join in the combined domain





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

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



$$\begin{array}{lcl}
\delta_0 & \equiv & (\alpha_0, \beta_0) \\
\delta_1 & \equiv & (\alpha_4, \beta_2) \\
\delta_2 & \equiv & (\alpha_1, \beta_1) \\
\delta_3 & \equiv & (\alpha_5, \beta_3)
\end{array}$$

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

Stage 4: join in the numeric domain

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

Memory abstract domain interfaces

- **Symbolic variables** represent **concrete values** $\mathbf{Val}^\sharp$
- They appear in $\mathbb{C}^\sharp$, in $\mathbb{S}^\sharp$, in $\mathbb{G}^\sharp$, and in $\mathbb{V}^\sharp$

Concretizations all involve **valuations** (values of symbolic variables)

$$\begin{array}{lrcll}
\text{for } \mathbb{G}^\sharp : & \gamma_{\mathbb{G}} : & \mathbb{G}^\sharp & \rightarrow & \mathcal{P}((\mathbf{Val}^\sharp \rightarrow \mathbf{Val}) \times \mathbf{Mem}) \\
\text{for } \mathbb{V}^\sharp : & \gamma_{\mathbb{V}} : & \mathbb{V}^\sharp & \rightarrow & \mathcal{P}(\mathbf{Val}^\sharp \rightarrow \mathbf{Val}) \\
\text{for } \mathbb{S}^\sharp : & \gamma_{\mathbb{C}} : & \mathbb{C}^\sharp & \rightarrow & \mathcal{P}((\mathbf{Val}^\sharp \rightarrow \mathbf{Val}) \times \mathbf{Mem})
\end{array}$$

Abstract operations in $\mathbb{G}^\sharp$ (simplified excerpt):

$$\begin{array}{lrcl}
\text{unfold :} & \mathbb{G}^\sharp \times \mathbf{Val}^\sharp & \rightarrow & \mathcal{P}_{\mathrm{fin}}(\mathbb{G}^\sharp) \\
\text{eval-lvalue :} & \mathbb{G}^\sharp \times \mathbf{Lvalue}[\mathbf{Val}^\sharp] & \rightarrow & \mathbf{Val}^\sharp \\
\text{eval-expr :} & \mathbb{G}^\sharp \times \mathbf{Expr}[\mathbf{Val}^\sharp] & \rightarrow & \mathbf{Val}^\sharp \\
\text{read :} & \mathbb{G}^\sharp \times \mathbf{Val}^\sharp \times \text{offset} \times \text{size} & \rightarrow & \mathbb{G}^\sharp \times \mathbf{Val}^\sharp \\
\text{write :} & \mathbb{G}^\sharp \times \mathbf{Val}^\sharp \times \text{offset} \times \text{size} & \rightarrow & \mathbb{G}^\sharp
\end{array}$$

Outline

1. Memory abstraction
2. A few memory abstractions, with value information
3. Abstract operations
4. Instantiations
5. Separating product of abstractions
6. Reduced product of abstractions
7. Abstraction of arrays and hierarchical abstraction
8. Project status

Instantiations of $\mathbb{G}^\sharp$

A general $\mathbb{G}^\sharp$ definition; needs to be instantiated

- It is **parameterized** by **a set of inductive definitions**
- Inductive definitions could be **built-in** in the implementation, pre-defined by the user, inferred...

Many possible instantiations, result in **different implementations**:

- $\mathbb{G}^\sharp_{\langle \iota_0, \ldots, \iota_k \rangle}$, **parameterized by a set of inductive definitions**: **inputs a finite set of inductive predicates**, **very expressive**
- $\mathbb{G}^\sharp_{\mathrm{flat}}$, **no inductive definitions** [Miné, LCTES'06]
- $\mathbb{G}^\sharp_{\mathrm{list}}$, **a single, built-in definition** (e.g., list)

Instantiations of $\mathbb{G}^\sharp$: impact on scalability

Data-structures to represent abstract values:

- $\mathbb{G}^\sharp_{\langle \iota_0, \ldots, \iota_k \rangle}$: based on **graphs**
- $\mathbb{G}^\sharp_{\mathrm{flat}}$, **no inductive definitions**: based on **functional arrays** (maps)
- $\mathbb{G}^\sharp_{\mathrm{list}}$, **a single, built-in definition**: also graphs

Algorithms, e.g. for **join**:

- $\mathbb{G}^\sharp_{\langle \iota_0, \ldots, \iota_k \rangle}$:
  needs to traverse the graphs (almost never physically equal)
- $\mathbb{G}^\sharp_{\mathrm{flat}}$, **no inductive definitions**:
  can be done in $\mathcal{O}(1)$ when physically equal
- $\mathbb{G}^\sharp_{\mathrm{list}}$: graph join, as in $\mathbb{G}^\sharp_{\langle \iota_0, \ldots, \iota_k \rangle}$,
  but much simpler: **no inductive inference in folding**

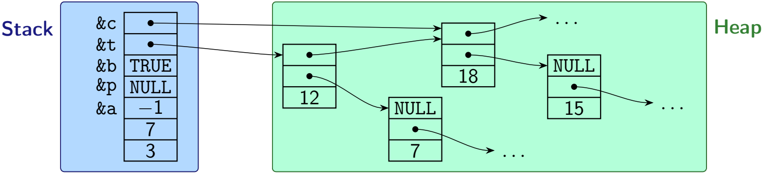
Outline

1. Memory abstraction
2. A few memory abstractions, with value information
3. Abstract operations
4. Instantiations
5. **Separating product of abstractions**
6. Reduced product of abstractions
7. Abstraction of arrays and hierarchical abstraction
8. Project status

Applying different abstractions to different regions

- Stores often contain (large) **trivial** regions (e.g., local variables)
- More complex structures may be **limited to precise areas** (heap)

Expensive abstraction should not be applied uniformly

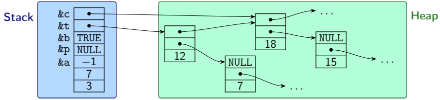


Different local abstractions on different regions

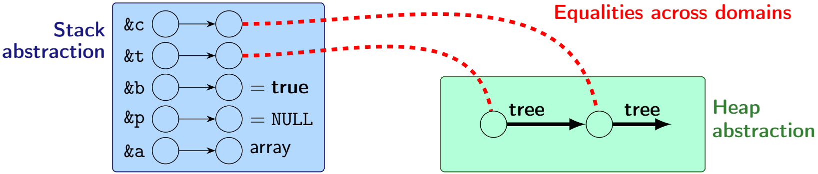
- **Separating conjunction** is applied **at the abstract domain level**
- **Lightweight abstraction** for region **Stack**: $\mathbb{G}^\sharp_{\mathrm{flat}}$
- **More precise abstraction** for region **Heap**: $\mathbb{G}^\sharp_{\langle \mathbf{tree} \rangle}$

The separating product [SAS'14]

A concrete state:



Abstract state:

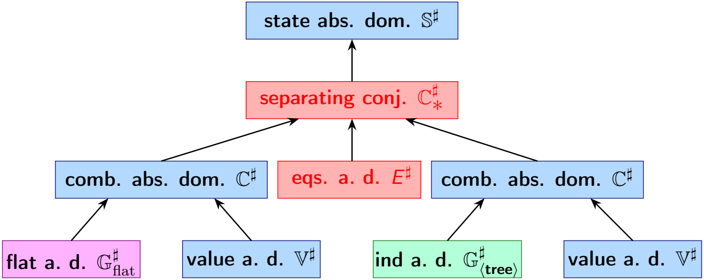


Abstract domain and concretization

- $\mathbb{C}^\sharp_* = \mathbb{C}^\sharp_1 \times \mathbb{C}^\sharp_2 \times E^\sharp$ where $E^\sharp$ abstracts **equalities over $\mathbf{Val}^\sharp \times \mathbf{Val}^\sharp$**
- $\gamma_*(\mathbf{G}_0, \mathbf{G}_1, R) = \{(h_0 \uplus h_1, \nu) \mid \forall i,\ (h_i, \nu) \in \gamma_{\mathbb{C}}(\mathbf{G}_i) \wedge \nu \in \gamma_{\equiv}(R)\}$

Use of the separating product

Resulting abstract domain structure:



A set of ML modules:

- Lightweight $\mathbb{G}^\sharp_{\mathrm{flat}}$
- Expressive $\mathbb{G}^\sharp_{\langle\mathbf{tree}\rangle}$, applied locally
- Combination functor + equalities

Alternate composition order:

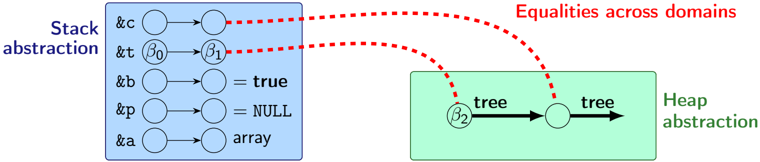
- Product with $\mathbb{V}^\sharp$ above $\mathbb{C}^\sharp_*$
- More expressive
- More expensive

Computation of transfer functions

Extension of abstract operations

- L-values may need to **evaluate over several sub-domains**
- **Equalities** also have to be used, in order to evaluate paths

Example: read `t -> l` in



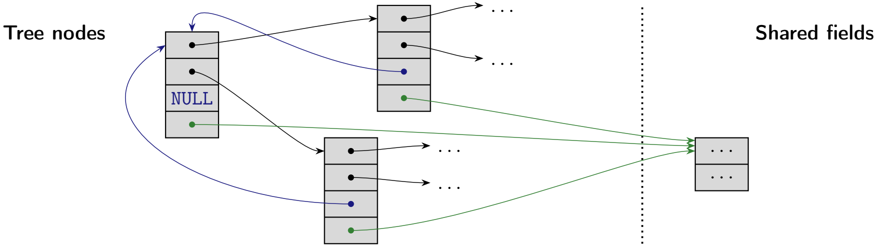
1. `&t` evaluates into β_0 in $\mathbb{G}^\sharp_{\mathrm{flat}}$
2. $\star$(`&t`) evaluates into β_1, also in $\mathbb{G}^\sharp_{\mathrm{flat}}$
3. But then `t -> l` needs to evaluate inside $\mathbb{G}^\sharp_{\langle \textbf{tree} \rangle}$
   To do this, we need to use **equalities**: $\beta_1 = \beta_2$

Outline

Conjoining heap abstractions

An overlaid data-structure:

- **Tree** with parent pointers
- All nodes have a shared pointer to a common record



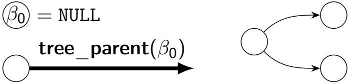
A global conjunction of properties

- Calls for a **reduced product** of abstract domains
- **Issue:** conjunction in memory abstractions can be expensive

The reduced product of memory abstract domains [VMCAI'13]

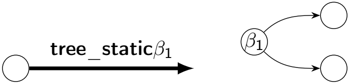
Domain $\mathbb{C}_0^\sharp$:

- Inductive **tree_parent**: tracks **parent pointers**
- $\mathbb{C}_0^\sharp$ is $\mathbb{G}^\sharp_{\langle \textbf{tree\_parent} \rangle}$



Domain $\mathbb{C}_1^\sharp$:

- Inductive **tree_static**: tracks **pointers to static fields**
- $\mathbb{C}_1^\sharp$ is $\mathbb{G}^\sharp_{\langle \textbf{tree\_static} \rangle}$

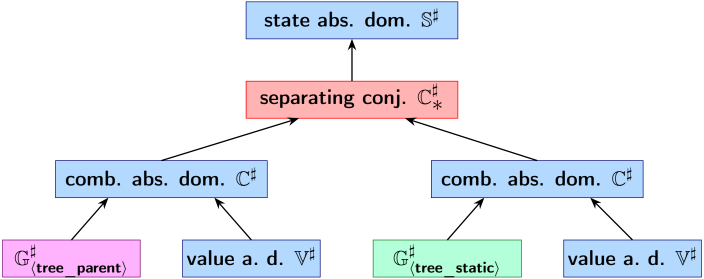


Abstract domain and concretization

- $\mathbb{C}_\wedge^\sharp = \mathbb{C}_1^\sharp \times \mathbb{C}_2^\sharp$
- $\gamma_\wedge(\mathbf{G}_0, \mathbf{G}_1) = \gamma_{\mathbb{C}}(\mathbf{G}_0) \cap \gamma_{\mathbb{C}}(\mathbf{G}_1)$

Use of the reduced product

Resulting abstract domain structure:



- **Reduction** at **assignment / tests**: exchange of **reachability** predicates, inferred from inductive definitions
- **Run-time:** about twice as long as the run-time of a single analysis
- **Gain:** **separation of concerns** in the inductive predicate specification

Outline

Array abstraction

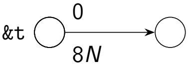
The shape abstract domain can abstract arrays in several ways

Example **concrete state:**

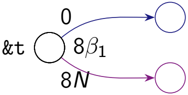
t	v_0	v_1	v_2	v_3	v_4	v_5	v_6

Abstractions:

1. A points-to edge may represent a **block of any size**: $\beta_0 \cdot [0, 8[\rightarrow \beta_1$
   Then, $\mathbb{V}^\sharp$ views β_1 an $8N$ bytes long block,
   can let a **dedicate array abstract domain** deal with it...



2. Offsets can be **expressions**, as in, e.g., $\beta_0 \cdot [8\beta_1, 8\beta_1 + 4[\rightarrow \beta_2$
   as in [Cousot,Cousot,Logozzo'11]



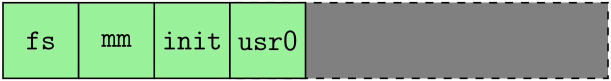
A Process Table in Minix 1.1

- Minix 1.1 is a **tiny operating system**
- It features **services** like **memory management**
- Memory management stores a **process table**, with one record per process

- Records of system processes are **permanent**
  - `mm` : memory management process
  - `fs` : file system process
  - `init` : the ancestor of all user processes

A Process Table in Minix 1.1

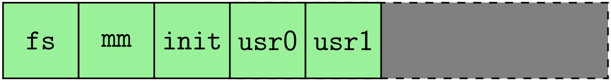
- Minix 1.1 is a **tiny operating system**
- It features **services** like **memory management**
- Memory management stores a **process table**, with one record per process



- Records of system processes are **permanent**
  - `mm` : memory management process
  - `fs` : file system process
  - `init` : the ancestor of all user processes
- User processes `usr`i, **dynamically created**

A Process Table in Minix 1.1

- Minix 1.1 is a **tiny operating system**
- It features **services** like **memory management**
- Memory management stores a **process table**, with one record per process

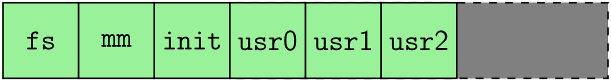


`fs` `mm` `init` `usr0` `usr1`

- Records of system processes are **permanent**
  - `mm` : memory management process
  - `fs` : file system process
  - `init` : the ancestor of all user processes
- User processes `usr`i, **dynamically created**

A Process Table in Minix 1.1

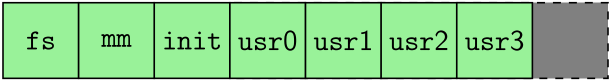
- Minix 1.1 is a **tiny operating system**
- It features **services** like **memory management**
- Memory management stores a **process table**, with one record per process



- Records of system processes are **permanent**
  - `mm` : memory management process
  - `fs` : file system process
  - `init` : the ancestor of all user processes
- User processes `usr`i, **dynamically created**

A Process Table in Minix 1.1

- Minix 1.1 is a **tiny operating system**
- It features **services** like **memory management**
- Memory management stores a **process table**, with one record per process

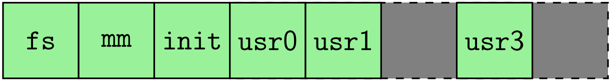


`fs`	`mm`	`init`	`usr0`	`usr1`	`usr2`	`usr3`	▨

- Records of system processes are **permanent**
  - `mm` : memory management process
  - `fs` : file system process
  - `init` : the ancestor of all user processes
- User processes `usr`i, **dynamically created**

A Process Table in Minix 1.1

- Minix 1.1 is a **tiny operating system**
- It features **services** like **memory management**
- Memory management stores a **process table**, with one record per process

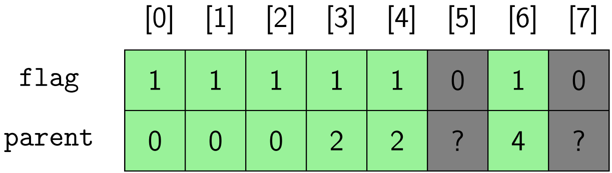


- Records of system processes are **permanent**
  - `mm` : memory management process
  - `fs` : file system process
  - `init` : the ancestor of all user processes
- User processes `usr`i, **dynamically created** or **removed**

Cells with similar properties are not contiguous

A Process Table in Minix 1.1: Data-Type

```
struct{
    int flag;
    \* valid is not 0 *\
    int parent;
    \* the index of parent *\
}mproc[24]
```



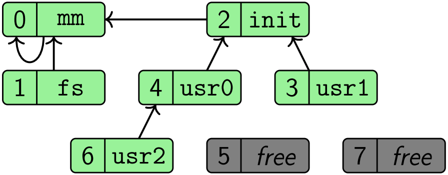
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
flag	1	1	1	1	1	0	1	0
parent	0	0	0	2	2	?	4	?

A Process Table in Minix 1.1: Correctness Invariant

Structural correctness property `MMTPC`

Field `parent` of any valid process descriptor should be the index of a valid process descriptor

- i.e., **no dangling processes**
- should be **established after** `mm_init`
- should be **preserved by system calls**



🟩 valid process

▨ free slots

A Code Segment from Fork

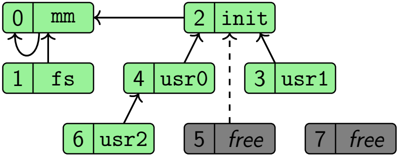
{Pre-condition: `MMTPC`}

```
int fork(int seed)
{
   ...
   for(i = 0; i < 24; i++){
      if(mproc[i].flag == 0)
         break;
   }
   mproc[i].flag = mproc[seed].flag;
   mproc[i].parent = seed;
   return 1;
}
```

{Post-condition: `MMTPC`}

① searches for a free slot
② allocates a record
③ update parent

🟩 valid process
⬛ free slots

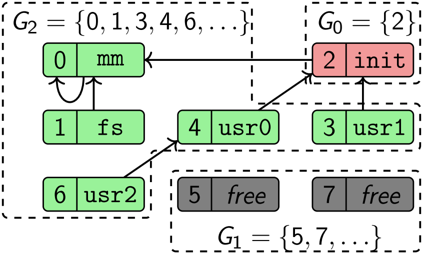


We want to verify `MMTPC` automatically
Thus, we need contiguous partitions

Towards an Abstraction of `mproc`

All cells in array `mproc` are partitioned into **three disjoint groups:**

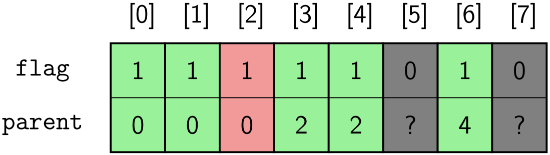
- **Group** G_0: `init` process
- **Group** G_1: free slots in array `mproc`
- **Group** G_2: all valid process descriptors except `init`



Key distinguishing factor with common array abstractions:
groups may be non-contiguous

Towards an Abstraction of `mproc`

A concrete memory state:

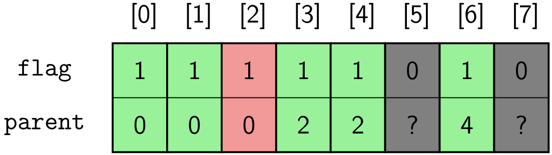


	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
`flag`	1	1	1	1	1	0	1	0
`parent`	0	0	0	2	2	?	4	?

Abstract memory state:

Towards an Abstraction of `mproc`

A concrete memory state:



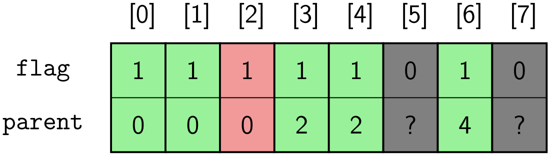
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
`flag`	1	1	1	1	1	0	1	0
`parent`	0	0	0	2	2	?	4	?

Abstract memory state:

Partitions and field properties will be computed automatically

Towards an Abstraction of `mproc`

A concrete memory state:



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
`flag`	1	1	1	1	1	0	1	0
`parent`	0	0	0	2	2	?	4	?

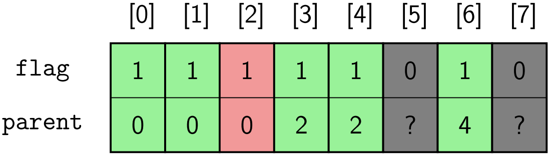
Abstract memory state:

$\underline{\mathrm{Index}}_i$: the summary of indexes in group i

Partitions and field properties will be computed automatically

Towards an Abstraction of `mproc`

A concrete memory state:



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
`flag`	1	1	1	1	1	0	1	0
`parent`	0	0	0	2	2	?	4	?

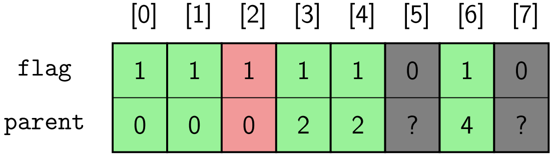
Abstract memory state:

$\underline{\mathrm{Size}}_i$: the number of elements in group i

Partitions and field properties will be computed automatically

Towards an Abstraction of `mproc`

A concrete memory state:



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
`flag`	1	1	1	1	1	0	1	0
`parent`	0	0	0	2	2	?	4	?

Abstract memory state:

G_2

$0 \leq \underline{\text{Index}}_2 \leq 23$

$0 \leq \underline{\text{Size}}_2 \leq 23$

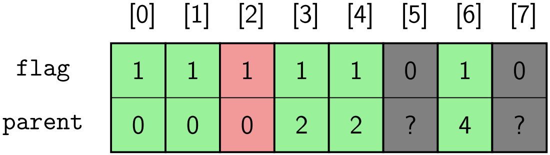
$\texttt{flag}_2 \geq 1$

$\texttt{flag}_i$: the summary of values on field `flag` in group i

Partitions and field properties will be computed automatically

Towards an Abstraction of `mproc`

A concrete memory state:



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
`flag`	1	1	1	1	1	0	1	0
`parent`	0	0	0	2	2	?	4	?

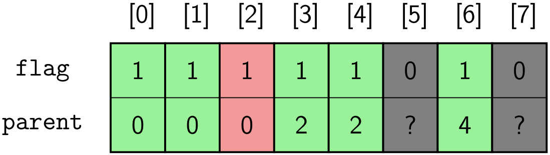
Abstract memory state:

$\mathtt{parent}_i$: the summary of values on field `parent` in group i

Partitions and field properties will be computed automatically

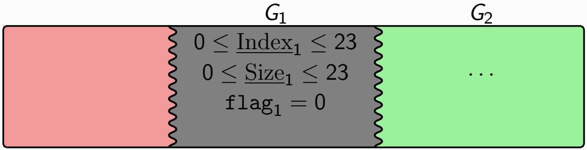
Towards an Abstraction of `mproc`

A concrete memory state:



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
`flag`	1	1	1	1	1	0	1	0
`parent`	0	0	0	2	2	?	4	?

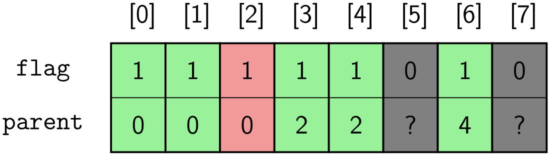
Abstract memory state:



Partitions and field properties will be computed automatically

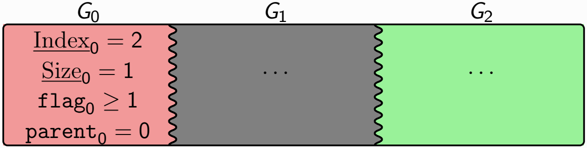
Towards an Abstraction of `mproc`

A concrete memory state:



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
`flag`	1	1	1	1	1	0	1	0
`parent`	0	0	0	2	2	?	4	?

Abstract memory state:



G_0

$\underline{\text{Index}_0} = 2$

$\underline{\text{Size}_0} = 1$

$\texttt{flag}_0 \geq 1$

$\texttt{parent}_0 = 0$

G_1

$\cdots$

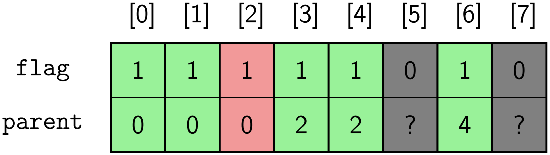
G_2

$\cdots$

Partitions and field properties will be computed automatically

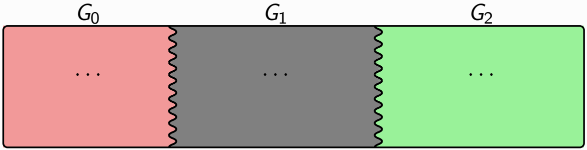
Towards an Abstraction of `mproc`

A concrete memory state:



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
`flag`	1	1	1	1	1	0	1	0
`parent`	0	0	0	2	2	?	4	?

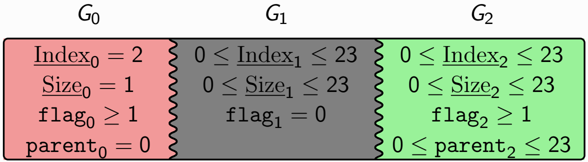
Abstract memory state:



Group Relations: $\mathtt{parent}_2 \lhd G_0 \cup G_2$

Partitions and field properties will be computed automatically

Abstract States

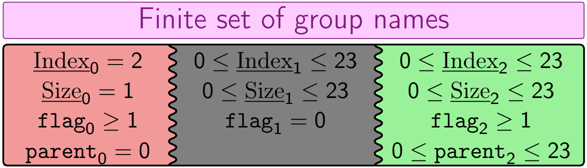


G_0	G_1	G_2
$\underline{\mathrm{Index}}_0 = 2$	$0 \leq \underline{\mathrm{Index}}_1 \leq 23$	$0 \leq \underline{\mathrm{Index}}_2 \leq 23$
$\underline{\mathrm{Size}}_0 = 1$	$0 \leq \underline{\mathrm{Size}}_1 \leq 23$	$0 \leq \underline{\mathrm{Size}}_2 \leq 23$
$\mathtt{flag}_0 \geq 1$	$\mathtt{flag}_1 = 0$	$\mathtt{flag}_2 \geq 1$
$\mathtt{parent}_0 = 0$		$0 \leq \mathtt{parent}_2 \leq 23$

Group Relations: $\mathtt{parent}_2 \lhd G_0 \cup G_2$

- **Concretization into sets of concrete states**: in the paper

Abstract States



Finite set of group names

$\underline{\mathrm{Index}}_0 = 2$	$0 \leq \underline{\mathrm{Index}}_1 \leq 23$	$0 \leq \underline{\mathrm{Index}}_2 \leq 23$
$\underline{\mathrm{Size}}_0 = 1$	$0 \leq \underline{\mathrm{Size}}_1 \leq 23$	$0 \leq \underline{\mathrm{Size}}_2 \leq 23$
$\mathtt{flag}_0 \geq 1$	$\mathtt{flag}_1 = 0$	$\mathtt{flag}_2 \geq 1$
$\mathtt{parent}_0 = 0$		$0 \leq \mathtt{parent}_2 \leq 23$

Group Relations: $\mathtt{parent}_2 \lhd G_0 \cup G_2$

An abstract state comprises:

- A **finite set of groups**

- **Concretization into sets of concrete states**: in the paper

Abstract States

Finite set of group names

Numerical constraints

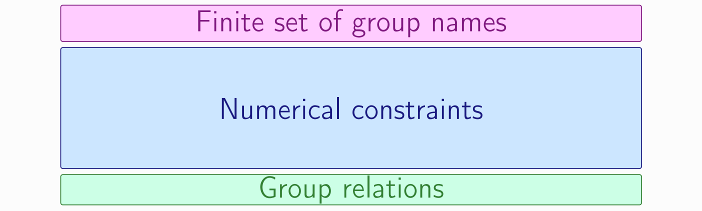
Group Relations: $\texttt{parent}_2 \lhd G_0 \cup G_2$

An abstract state comprises:

- A **finite set of groups**
- **Numerical constraints** over:
  - **base type variables**
  - group **sizes** and **cell indexes**
  - group **fields summary variables**

- **Concretization into sets of concrete states**: in the paper

Abstract States



An abstract state comprises:

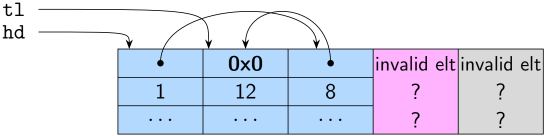
- A **finite set of groups**
- **Numerical constraints** over:
  - **base type variables**
  - group **sizes** and **cell indexes**
  - group **fields summary variables**
- **Group relations:** basic membership constraints

- **Concretization into sets of concrete states**: in the paper

Hierarchical abstraction [APLAS'12]

A common pattern in **embedded** software:

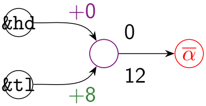
- No use of `malloc`, manual management of a **static zone** instead
- One particular implementation: **large array** of structures, allocated **from the beginning**, fully erased from time to time

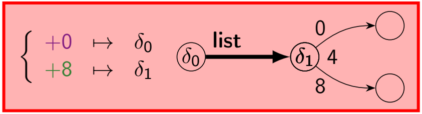


•	**0x0**	•	invalid elt	invalid elt
1	12	8	?	?
...	...	...	?	?

This makes the analysis **very challenging**:
both array and dynamic structures abstractions will fail...

Hierarchical abstraction: attaches a (shape) predicate to a node





$$\begin{cases} +0 \mapsto \delta_0 \\ +8 \mapsto \delta_1 \end{cases}$$

Outline

1. Memory abstraction
2. A few memory abstractions, with value information
3. Abstract operations
4. Instantiations
5. Separating product of abstractions
6. Reduced product of abstractions
7. Abstraction of arrays and hierarchical abstraction
8. Project status

Results and future works

Abstract domains

- Inductive-parameterized $\mathbb{G}^\sharp_{\langle\ldots\rangle}$
- List specific $\mathbb{G}^\sharp_{\mathrm{flat}}$
- Flat $\mathbb{G}^\sharp_{\mathrm{list}}$
- Array, non contiguous partitions
- Numeric (Apron)

Combination operations

- Separating product
- Reduced product
- Hierarchical functor

Ongoing and future works:

- **Additional domains** (arrays, strings)
- **Improvement of the analyzer** / **tuning of the operators**
- **Packaging of the domain** for integration in other tools, e.g. Astrée