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 $\mathcal{P}(\text{Environment} \times \text{Stack} \times \text{Heap})$
- 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 few memory abstractions, with value information

A generic memory abstraction step by step

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

\[
\begin{align*}
&(x \cdot n) = 0x\ldots a_0 & \text{17} \\
&(x \cdot d) = 0x\ldots a_4 & 0x\ldots b_0 \\
&(y \cdot n) = 0x\ldots b_0 & 17 \\
&(y \cdot d) = 0x\ldots b_4 & 0x0
\end{align*}
\]
A few memory abstractions, with value information

A generic memory abstraction step by step

- **Concrete memory states**

- **Abstraction of values into symbolic variables** (nodes)

  \[
  \begin{array}{c|c|c}
  \text{0x...a0} & 17 & \alpha_0 \\
  \hline
  \text{0x...b0} & & \alpha_1 \\
  \hline
  \end{array}
  \quad
  \begin{array}{c|c|c}
  \text{0x...b0} & 17 & \alpha_2 \\
  \hline
  \text{0x0} & & \alpha_3 \\
  \end{array}
  \quad
  \begin{array}{c|c|c}
  \text{0x...a0} & 0x...a0 & \nu(\alpha_0) = 0x...a0 \\
  \hline
  \text{17} & 17 & \nu(\alpha_1) = 17 \\
  \hline
  \text{0x0} & 0x0 & \nu(\alpha_4) = 0x0 \\
  \end{array}
  \quad
  \begin{array}{c|c|c}
  \text{17} & 0x...b0 & \nu(\alpha_2) = 0x...b0 \\
  \hline
  \end{array}
  \quad
  \begin{array}{c|c|c}
  \text{17} & 17 & \nu(\alpha_3) = 17 \\
  \hline
  \end{array}
  \\

  \text{characterized by valuation } \nu \\
  \text{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

\[ \begin{align*}
\nu(\alpha_0) &= 0x...a0 \\
\nu(\alpha_1) &= 17 \\
\nu(\alpha_2) &= 0x...b0 \\
\nu(\alpha_3) &= 17 \\
\nu(\alpha_4) &= 0x0
\end{align*} \]
A few memory abstractions, with value information

A generic memory abstraction step by step

- Concrete memory states

- Abstraction of values into symbolic variables / nodes

- Abstraction of regions into points-to edges

- Shape graph concretization

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

valuation \( \nu \) plays an important role in the combined domain

\[ \nu(\alpha_0) = \text{0x...a0} \]
\[ \nu(\alpha_1) = 17 \]
\[ \nu(\alpha_2) = \text{0x...b0} \]
\[ \nu(\alpha_3) = 17 \]
\[ \nu(\alpha_4) = \text{0x0} \]
A few memory abstractions, with value information

Adding value information (here, numeric)

- **Concrete numeric values** appear in the valuation: abstract $\nu$!

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

- **Memory abstraction:**

- **Abstraction of valuations:** $\nu(\alpha_1) = \nu(\alpha_3)$, (constraint $\alpha_1 = \alpha_3$)
  - combined abstraction: product of shape and numeric over nodes
  - $\gamma(G, N) = \{(h, \nu) | (h, \nu) \in \gamma(G) \land \nu \in \gamma(N)\}$
A few memory abstractions, with value information

Adding environment abstraction

Addresses can be read in the valuations

\[ \begin{align*}
&x = \& (x \cdot n) = 0x...a0 \\
&x = \& (x \cdot d) = 0x...a4
\end{align*} \]

\[ \begin{align*}
&y = \& (y \cdot n) = 0x...b0 \\
&y = \& (y \cdot d) = 0x...b4
\end{align*} \]

Abstraction of environments:
- mapping \( E \) from variables to symbolic nodes
- \( \gamma(E, G, N) = \{(e, h, \nu) \mid (h, \nu) \in \gamma(G, N) \land e = \nu \circ E\} \)
Adding summarization

Example:
set of all lists of any length

List inductive predicate
\[
\alpha \cdot \text{list} ::= \\
(\text{emp} \land \alpha = 0x0) \\
\lor \left( \alpha \cdot \text{d} \mapsto \beta_0 \land \alpha \cdot \text{n} \mapsto \beta_1 \land * \beta_1 \cdot \text{list} \land \alpha \neq 0x0 \right)
\]

well-founded predicate

Inductive summary predicates

Concretization based on unfolding and least-fixpoint:

- \( \leadsto \) replaces an \( \alpha \cdot \text{list} \) predicate with one of its premises

\[
\gamma(G, N) = \bigcup \{ \gamma(G', N') \mid (G, N) \leadsto (G', N') \}
\]

(all inductive predicates are required to be well-founded)
A few memory abstractions, with value information

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** $S^\#$
  - $(E, G, N)$ abstracts sets of $(e, h)$

- **combined shape-value abstract domain** $C^\#$
  - $(G, N)$ abstracts sets of $(h, \nu)$

- **shape abstract domain** $G^\#$
  - $G$ abstracts sets of $(h, \nu)$

- **value abstract domain** $\nabla^\#$
  - $N$ abstracts sets of $\nu$
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 $x = y \rightarrow \text{next}$ (local reasoning)

1. Evaluate \textit{l-value} $x$ into \textbf{points-to edge} $\alpha \mapsto \beta$
2. Evaluate \textit{r-value} $y \rightarrow \text{next}$ into \textbf{node} $\beta'$
3. Replace \text{points-to edge} $\alpha \mapsto \beta$ with \textbf{points-to edge} $\alpha \mapsto \beta'$

\textbf{With pre-condition:}

- Step 1 produces $\alpha_0 \mapsto \beta_0$
- Step 2 produces $\beta_2$
- End result:

\textbf{With pre-condition:}

- Step 1 produces $\alpha_0 \mapsto \beta_0$
- Step 2 fails
- Abstract state \textit{too abstract}
- We need to \textbf{refine it}
Abstract operations

Analysis of an assignment, with unfolding

**Principle**
- We have \( \gamma_G(\alpha \cdot \iota) = \bigcup \{ \gamma_G(G) \mid \alpha \cdot \iota \xrightarrow{\text{unfold}} G \} \)
- Replace \( \alpha \cdot \iota \) with a finite number of disjuncts and continue

**Disjunct 1:**
\[
\begin{align*}
\&x \quad \alpha_0 & \quad \beta_0 \\
\&y \quad \alpha_1 & \quad \beta_1 \\
\end{align*}
\]
- Step 1 produces \( \alpha_0 \mapsto \beta_0 \)
- **Step 2 fails:** Null pointer dereference!

**Disjunct 2:**
\[
\begin{align*}
\&x \quad \alpha_0 & \quad \beta_0 & \xrightarrow{\text{next}} \beta_2 & \text{list} \\
\&y \quad \alpha_1 & \quad \beta_1 & \xrightarrow{\text{data}} \beta_3 \\
\end{align*}
\]
- Step 1 produces \( \alpha_0 \mapsto \beta_0 \)
- Step 2 produces \( \beta_2 \)
- **End result:**

**Case of a correct code...**
Would be ruled out by a condition \( y \neq 0 \) i.e., \( \beta_1 \neq 0 \) in numerics
Abstract operations

Analysis of an assignment in the combined domain

environment layer
shape + num + env

cofibered layer
shape + num

shape domain
numeric domain

\& x \begin{array}{c} \alpha_0 \\ \rightarrow \alpha_1 \end{array}

\& y \begin{array}{c} \alpha_2 \\ \rightarrow \alpha_3 \end{array}

lpos

N = \alpha_1 \geq 0 \land \alpha_3 \neq 0x0

y \rightarrow 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

**Stage 1: environment resolution**
- Replaces $x$ with $\star E(x)$

**Abstract post-condition**

\[
y \rightarrow d = x + 1 \implies (\star \alpha_2) \cdot d = (\star \alpha_0) + 1
\]

**Equation**

\[
N = \alpha_1 \geq 0 \land \alpha_3 \neq 0x0
\]
Abstract operations

Analysis of an assignment in the combined domain

environment layer

shape + num + env

cobibered layer

shape + num

shape domain

numeric domain

Stage 2: propagate into the shape + numerics domain

only symbolic nodes appear

Abstract post-condition?

\[
&x \quad (\alpha_0 \rightarrow \alpha_1)
\]

\[
&y \quad (\alpha_2 \rightarrow \alpha_3)
\]

\[
N = \alpha_1 \geq 0 \land \alpha_3 \neq 0x0
\]

\[
(\ast \alpha_2) \cdot d = (\ast \alpha_0) + 1
\]
Abstract operations

Analysis of an assignment in the combined domain

\[ N = \alpha_1 \geq 0 \land \alpha_3 \neq 0x0 \]

\[ (\star \alpha_2) \cdot d = (\star \alpha_0) + 1 \]

Stage 3: resolve cells in the shape graph abstract domain

- \( \star \alpha_0 \) evaluates to \( \alpha_1 \); \( \star \alpha_2 \) evaluates to \( \alpha_3 \)
- \((\star \alpha_2) \cdot d\) fails to evaluate: no points-to out of \( \alpha_3 \)
Abstract operations

Analysis of an assignment in the combined domain

environment layer
shape + num + env

cofibered layer
shape + num

shape domain numeric domain

N = α₁ ≥ 0 ∧ α₃ ≠ 0x0 ∧ α₄ ≥ 0

(⋆α₂) · d = (⋆α₀) + 1

Abstract post-condition ?

Stage 4: unfolding (several steps, skipped here)

- locally materialize α₃ · lpos; update keys / relations in the numerics
- l-value (⋆α₂) · d now evaluates into edge α₃ → d ↦ α₄
Abstract operations

Analysis of an assignment in the combined domain

Stage 5: create a new node

- new node $\alpha_6$ denotes a new value
  will store the new value

$$N = \alpha_1 \geq 0 \land \alpha_3 \neq 0x0 \land \alpha_4 \geq 0$$
Abstract operations

Analysis of an assignment in the combined domain

environment layer
shape + num + env

cofibered layer
shape + num

shape domain
numeric domain

Stage 6: perform numeric assignment

- numeric assignment completely ignores pointer structures
to the new node

\[ N = \alpha_1 \geq 0 \land \alpha_3 \neq 0x0 \land \alpha_4 \geq 0 \]

\[ \alpha_6 \leftarrow \alpha_1 + 1 \text{ in numerics} \]

\[ N = \alpha_1 \geq 0 \land \alpha_3 \neq 0x0 \land \alpha_4 \geq 0 \land \alpha_6 \geq 1 \]
Abstract operations

Analysis of an assignment in the combined domain

- environment layer: shape + num + env
- cofibered layer: shape + num
- shape domain
- numeric domain

Stage 7: perform the update in the graph
- classic **strong update** in a pointer aware domain
- symbolic node $\alpha_4$ becomes redundant and can be removed

\[\text{mutate } (\alpha_3 \cdot d) \mapsto \alpha_4 \text{ into } \alpha_6\]

\[N = \alpha_1 \geq 0 \land \alpha_3 \neq 0\times0 \land \alpha_4 \geq 0 \land \alpha_6 \geq 1\]
Abstract operations

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

N = \alpha_2 \geq \alpha_5 \geq 2

N' = \beta_3 \geq 1
Widening / join in the combined domain

Stage 1: abstract environment
- compute new abstract environment and initial node relation
  e.g., $\alpha_0, \beta_0$ both denote $\&x$

\[ N = \alpha_2 \geq \alpha_5 \geq 2 \]

\[ N' = \beta_3 \geq 1 \]

$\delta_0 \equiv (\alpha_0, \beta_0)$
$\delta_1 \equiv (\alpha_4, \beta_2)$
Widening / join in the combined domain

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
Abstract operations

Widening / join in the combined domain

environment layer
shape + num + env

cofibered layer
shape + num

shape domain
numeric domain

Stage 2: graph join
- apply local join rules
  - ex: points-to matching, weakening to inductive...
- incremental algorithm

N = α_2 ≥ α_5 ≥ 2

N' = β_3 ≥ 1

δ_0 ≡ (α_0, β_0)
δ_1 ≡ (α_4, β_2)
δ_2 ≡ (α_1, β_1)
Abstract operations

Widening / join in the combined domain

Stage 2: graph join
- apply local join rules
  - ex: points-to matching, weakening to inductive...
- incremental algorithm
Widening / join in the combined domain

Stage 2: graph join
- apply local join rules
  - ex: points-to matching, weakening to inductive...
- incremental algorithm
Abstract operations

Widening / join in the combined domain

Stage 3: conversion function application in numerics
- remove nodes that were abstracted away
- rename other nodes

$N = \alpha_2 \geq \alpha_5 \geq 2$

$N' = \beta_3 \geq 1$

$\delta_0 \equiv (\alpha_0, \beta_0)$
$\delta_1 \equiv (\alpha_4, \beta_2)$
$\delta_2 \equiv (\alpha_1, \beta_1)$
$\delta_3 \equiv (\alpha_5, \beta_3)$

$N \sqcup = [\delta_3 \geq 2] \sqcup [\delta_3 \geq 1]$
Widening / join in the combined domain

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 $\text{Val}^\#$
- They appear in $\mathcal{C}^\#$, in $\mathcal{S}^\#$, in $\mathcal{G}^\#$, and in $\mathcal{V}^\#$

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

- for $\mathcal{G}^\#$: $\gamma_G : \mathcal{G}^\# \to \mathcal{P}(\mathcal{Val}^\# \to \mathcal{Val}) \times \mathcal{Mem}$
- for $\mathcal{V}^\#$: $\gamma_V : \mathcal{V}^\# \to \mathcal{P}(\mathcal{Val}^\# \to \mathcal{Val})$
- for $\mathcal{S}^\#$: $\gamma_C : \mathcal{C}^\# \to \mathcal{P}(\mathcal{Val}^\# \to \mathcal{Val}) \times \mathcal{Mem}$

**Abstract operations** in $\mathcal{G}^\#$ (simplified excerpt):

- unfold : $\mathcal{G}^\# \times \mathcal{Val}^\# \to \mathcal{P}_\text{fin}(\mathcal{G}^\#)$
- eval-lvalue : $\mathcal{G}^\# \times \text{Lvalue}[\mathcal{Val}^\#] \to \mathcal{Val}^\#$
- eval-expr : $\mathcal{G}^\# \times \text{Expr}[\mathcal{Val}^\#] \to \mathcal{Val}^\#$
- read : $\mathcal{G}^\# \times \mathcal{Val}^\# \times \text{offset} \times \text{size} \to \mathcal{G}^\# \times \mathcal{Val}^\#$
- write : $\mathcal{G}^\# \times \mathcal{Val}^\# \times \text{offset} \times \text{size} \to \mathcal{G}^\#$
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

Xavier Rival (CNRS, ENS and INRIA)

Composite memory abstract domains

February, 2015
Instantiations of $G^\#$

A general $G^\#$ 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:

- $G^\#_{\langle \iota_0, \ldots, \iota_k \rangle}$, parameterized by a set of inductive definitions: inputs a finite set of inductive predicates, very expressive

- $G^\#_{\text{flat}}$, **no inductive definitions** [Miné, LCTES’06]

- $G^\#_{\text{list}}$, a single, built-in definition (e.g., list)
Instantiations of $\mathcal{G}^\#$: impact on scalability

**Data-structures** to represent abstract values:

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

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

- $\mathcal{G}^\#_{\langle \iota_0, \ldots, \iota_k \rangle}$:
  needs to traverse the graphs (almost never physically equal)
- $\mathcal{G}^\#_{\text{flat}}$, **no inductive definitions**:
  can be done in $O(1)$ when physically equal
- $\mathcal{G}^\#_{\text{list}}$: graph join, as in $\mathcal{G}^\#_{\langle \iota_0, \ldots, \iota_k \rangle}$,
  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**: $\mathcal{G}_{\text{flat}}$
- **More precise abstraction** for region **Heap**: $\mathcal{G}_{\langle \text{tree} \rangle}$
The separating product [SAS’14]

A concrete state:

Abstract state:

Abstract domain and concretization

\[ C^\#_\bullet = C^\#_1 \times C^\#_2 \times E^\# \] where \( E^\# \) abstracts equalities over \( \text{Val}^\# \times \text{Val}^\# \)

\[ \gamma_\bullet(G_0, G_1, R) = \{ (h_0 \uplus h_1, \nu) \mid \forall i, (h_i, \nu) \in \gamma_C(G_i) \land \nu \in \gamma_\equiv(R) \} \]
Use of the separating product

Resulting abstract domain structure:

A set of ML modules:
- Lightweight $\mathcal{G}_{\text{flat}}$
- Expressive $\mathcal{G}_{\text{tree}}$, applied locally
- Combination functor + equalities

Alternate composition order:
- Product with $\mathcal{V}$ above $\mathcal{C}_*$
- More expressive
- More expensive
Separating product of abstractions

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 \( t \rightarrow 1 \) in

1. \( \&t \) evaluates into \( \beta_0 \) in \( G^\#:_{flat} \)
2. \( \sharp(\&t) \) evaluates into \( \beta_1 \), also in \( G^\#:_{flat} \)
3. But then \( t \rightarrow 1 \) needs to evaluate inside \( G^\#:_{tree} \)

To do this, we need to use **equalities:** \( \beta_1 = \beta_2 \)
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
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 $\mathbb{C}_0^\#$:
- Inductive $\text{tree\_parent}$: tracks parent pointers
- $\mathbb{C}_0^\#$ is $G^\#_{\langle\text{tree\_parent}\rangle}$

\[ \beta_0 = \text{NULL} \]

\[ \text{tree\_parent}(\beta_0) \]

Domain $\mathbb{C}_1^\#$:
- Inductive $\text{tree\_static}$: tracks pointers to static fields
- $\mathbb{C}_1^\#$ is $G^\#_{\langle\text{tree\_static}\rangle}$

\[ \text{tree\_static}(\beta_1) \]

Abstract domain and concretization
- $\mathbb{C}_\wedge^\# = \mathbb{C}_1^\# \times \mathbb{C}_2^\#$
- $\gamma_\wedge(G_0, G_1) = \gamma_C(G_0) \cap \gamma_C(G_1)$
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

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
Array abstraction

The shape abstract domain can abstract arrays in several ways

Example **concrete state:**

<table>
<thead>
<tr>
<th></th>
<th>v0</th>
<th>v1</th>
<th>v2</th>
<th>v3</th>
<th>v4</th>
<th>v5</th>
<th>v6</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Abstractions:**

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

   ![Diagram](attachment:image.png)

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]

   ![Diagram](attachment:image.png)
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

<table>
<thead>
<tr>
<th>fs</th>
<th>mm</th>
<th>init</th>
<th>usr0</th>
</tr>
</thead>
</table>

- Records of system processes are permanent
  - mm : memory management process
  - fs : file system process
  - init : the ancestor of all user processes
- User processes usri, dynamically created
Abstraction of arrays and hierarchical abstraction

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

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>fs</td>
<td>mm</td>
<td>init</td>
<td>usr0</td>
<td>usr1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>fs</th>
<th>mm</th>
<th>init</th>
<th>usr0</th>
<th>usr1</th>
<th>usr2</th>
</tr>
</thead>
</table>

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

User processes \( \text{usr}i \), 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

<table>
<thead>
<tr>
<th>fs</th>
<th>mm</th>
<th>init</th>
<th>usr0</th>
<th>usr1</th>
<th>usr2</th>
<th>usr3</th>
</tr>
</thead>
</table>

- 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  usr3
```

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

Cells with similar properties are not contiguous
A Process Table in Minix 1.1: Data-Type

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

<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
</tr>
</thead>
<tbody>
<tr>
<td>flag</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>parent</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>?</td>
<td>4</td>
<td>?</td>
</tr>
</tbody>
</table>
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

![Diagram showing process table and fields]

- `mm` process
- `init` process
- `fs` process
- `usr0` process
- `usr1` process
- `usr2` process
- `free` slots

valid process
A Code Segment from Fork

```c
{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}
```

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:

<table>
<thead>
<tr>
<th>flag</th>
<th>parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>?</td>
</tr>
</tbody>
</table>

Abstract memory state:
Towards an Abstraction of mproc

A concrete memory state:

<table>
<thead>
<tr>
<th>flag</th>
<th>parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>?</td>
</tr>
</tbody>
</table>

Abstract memory state:

Partitions and field properties will be computed automatically
Towards an Abstraction of mproc

A concrete memory state:

<table>
<thead>
<tr>
<th>flag</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>?</td>
<td>4</td>
<td>?</td>
</tr>
</tbody>
</table>

Abstract memory state:

\[ G_2 \]

\[ 0 \leq \text{Index}_2 \leq 23 \]

\[ \text{Index}_i : \text{the summary of indexes in group } i \]

Partitions and field properties will be computed automatically
Towards an Abstraction of mproc

A concrete memory state:

<table>
<thead>
<tr>
<th>flag</th>
<th>parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>?</td>
</tr>
</tbody>
</table>

Abstract memory state:

Size$_i$: the number of elements in group $i$

Partitions and field properties will be computed automatically
Towards an Abstraction of \texttt{mproc}

A concrete memory state:

\[
\begin{array}{cccccccc}
\text{flag} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\
\text{parent} & 0 & 0 & 0 & 2 & 2 & ? & 4 & ? \\
\end{array}
\]

Abstract memory state:

\[
G_2
\]

\[
0 \leq \text{Index}_2 \leq 23 \\
0 \leq \text{Size}_2 \leq 23 \\
\text{flag}_2 \geq 1
\]

flag$_i$: the summary of values on field flag in group $i$

Partition and field properties will be computed automatically
Towards an Abstraction of \texttt{mproc}

A concrete memory state:

<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
</tr>
</thead>
<tbody>
<tr>
<td>flag</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>parent</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>?</td>
<td>4</td>
<td>?</td>
</tr>
</tbody>
</table>

Abstract memory state:

\[ G_2 \]

\[ 0 \leq \text{Index}_2 \leq 23 \]
\[ 0 \leq \text{Size}_2 \leq 23 \]
\[ \text{flag}_2 \geq 1 \]
\[ 0 \leq \text{parent}_2 \leq 23 \]

\text{parent}_{i} : \text{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:

<table>
<thead>
<tr>
<th>flag</th>
<th>parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>?</td>
</tr>
</tbody>
</table>

Abstract memory state:

\[
\begin{align*}
G_1 & : 0 \leq \text{Index}_1 \leq 23 \\
& \text{and } 0 \leq \text{Size}_1 \leq 23 \\
& \text{flag}_1 = 0 \\
G_2 & : \ldots
\end{align*}
\]

Partitions and field properties will be computed automatically
Towards an Abstraction of mproc

A concrete memory state:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Abstract memory state:

\[ G_0 \]

- Index_0 = 2
- Size_0 = 1
- Flag_0 ≥ 1
- Parent_0 = 0

\[ G_1 \]

\[ G_2 \]

Partitions and field properties will be computed automatically
Towards an Abstraction of \texttt{mproc}

**A concrete memory state:**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>flag</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>parent</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>?</td>
<td>4</td>
</tr>
</tbody>
</table>

**Abstract memory state:**

Group Relations: parent\(_2 \prec G_0 \cup G_2\)

Partitions and field properties will be computed automatically
**Abstract States**

<table>
<thead>
<tr>
<th>$G_0$</th>
<th>$G_1$</th>
<th>$G_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Index}_0 = 2$</td>
<td>$0 \leq \text{Index}_1 \leq 23$</td>
<td>$0 \leq \text{Index}_2 \leq 23$</td>
</tr>
<tr>
<td>$\text{Size}_0 = 1$</td>
<td>$0 \leq \text{Size}_1 \leq 23$</td>
<td>$0 \leq \text{Size}_2 \leq 23$</td>
</tr>
<tr>
<td>$\text{flag}_0 \geq 1$</td>
<td>$\text{flag}_1 = 0$</td>
<td>$\text{flag}_2 \geq 1$</td>
</tr>
<tr>
<td>$\text{parent}_0 = 0$</td>
<td></td>
<td>$0 \leq \text{parent}_2 \leq 23$</td>
</tr>
</tbody>
</table>

**Group Relations:** $\text{parent}_2 \prec G_0 \cup G_2$

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

#### Finite set of group names

<table>
<thead>
<tr>
<th></th>
<th>0 ≤ Index₁ ≤ 23</th>
<th>0 ≤ Index₂ ≤ 23</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Index}_0$</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>$\text{Size}_0$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$\text{flag}_0$</td>
<td>≥ 1</td>
<td>0</td>
</tr>
<tr>
<td>parent₀</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

$0 \leq \text{Index}_0 \leq 23$

$0 \leq \text{Index}_1 \leq 23$

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

$0 \leq \text{Size}_0 \leq 23$

$0 \leq \text{Size}_1 \leq 23$

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

$\text{flag}_0 \geq 1$

$\text{flag}_1 = 0$

$\text{flag}_2 \geq 1$

$0 \leq \text{parent}_0 \leq 23$

$0 \leq \text{parent}_1 \leq 23$

$0 \leq \text{parent}_2 \leq 23$

**Group Relations:** parent₂ ⊲ $G₀ \cup G₂$

---

**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:** \( \text{parent}_2 \triangleleft G_0 \cup G_2 \)

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

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

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

- Inductive-parameterized $\mathcal{G}^\#\langle\ldots\rangle$
- List specific $\mathcal{G}^\#_{\text{flat}}$
- Flat $\mathcal{G}^\#_{\text{list}}$
- 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