Hierarchical Shape Abstraction of Dynamic Structures in Static Blocks

Pascal Sotin and Xavier Rival

INRIA

4 novembre 2013

Context of this talk

Previous talks :

- Static analysis of **embedded softwares**, with **Astrée**
  mostly **numeric and boolean properties** + **control**
- **Shape analysis** with **Xisa / MemCAD**
  inference of memory invariants, mixing value properties

This talk

Towards an application of shape abstraction
to the analysis of embedded softwares

Outline

Verification of safety critical embedded softwares

- **Synchronous softwares**, mostly **numeric**, **few**, rather **flat** data-structures
  Vérification of several industrial size applications by **Astrée** :
  - **flight-by-wire** software, around 1 MLOC
    no dynamic structures, no malloc
  - Analyzer designed since 2001 at ENS :
    B. Blanchet, P. Cousot, R. Cousot, J. Feret,
    L. Mauborgne, A. Miné, D. Monniaux, X. Rival
- **Beyond synchronous softwares :**
  - e.g., **flight Warning System** : gathers info about aircraft systems
  - **asynchronous** : Miné (ESOP'11)
  - uses **a few, tricky data structures**
    dynamic, but no malloc

How to verify the code using those structures
by abstract interpretation based static analysis ?

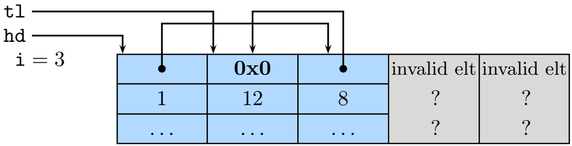
An example data-structure from a critical embedded code

- "Normal software" **uses malloc**
- In **highly critical, embedded code**, **malloc** should not be used
  - it **may fail**
    returns 0, e.g. if no long enough, contiguous block
  - no control on localization

Static array, dynamic list

```
typedef struct Cell {
    struct Cell *next;
    int prio;
    /* other fields */
} Cell;
Cell free_pool[100];
```

Example, with **length** 5



A common pattern in avionics and aerospace softwares

In FWS : list of messages to send to a **display**

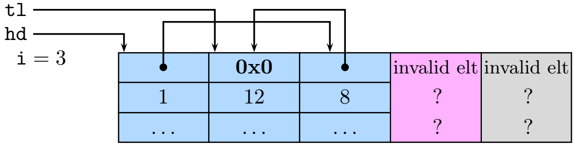
A code fragment

Display control

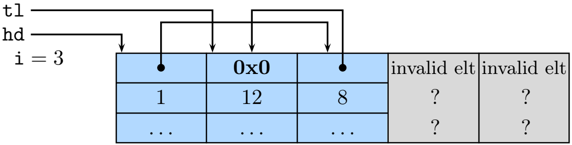
- **Navigation** in the list of messages : structure traversal
- **Recomputation** of the list of **active** messages, in order

```
for(int i = 0, i < 100, i++) {
    int priority = ...;
    if(priority < hd -> prio) { /*insert at first position */
    } else if(priority >= tlprio){ /*insert at last position*/
    } else { /*locate position (loop over cursor cr) and insert*/ }
```

next insertion
at position 3



Issues to resolve

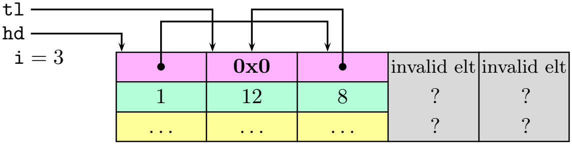


1. **Abstract the array**, in **two zones**
2. **Abstract the dynamic structure**, using **precise shape invariants**
3. **Analyze memory accesses** using **array indexes** and **list pointers**

Array abstractions

First approach : **array abstraction** [GRS'05,HP'08,CCL'11]

- **partition** of the array into **zones**
- **specific invariants** over each zone



Abstraction over the list zone

- $\mathtt{fp}[i].\mathtt{next} \in \{\mathtt{fp} + k \cdot \textbf{sizeof}(\textbf{Cell}) \mid k \in \mathbb{N}\}$
- $\mathtt{fp}[i].\mathtt{prio} \geq 0$

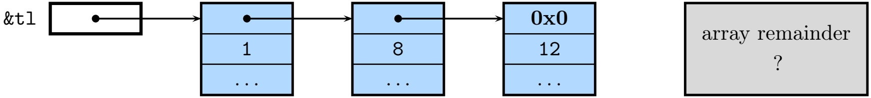
But...

- **The list structure** is **lost**
- **List accesses cannot be analyzed**

Shape abstractions for dynamic structures

Second approach : **shape abstraction** [SRW'99,DOY'06,CR'08]

- use **shape graphs** to describe **unbounded regions**
- rely on e.g., **separation logic** to **fragment** the heap



Dynamic structure

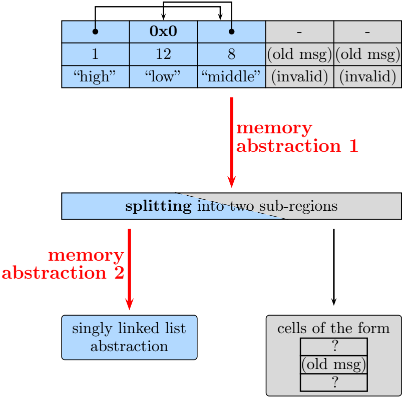
- **The list structure** is well described
- It can be **summarized** partially or fully

But...

- The **contiguousness** is **lost**
- **Accesses via indexes** or via **field arithmetics cannot be analyzed**

Outline

Abstraction principle



A composite abstraction

- **Array level** : fragmentation, partitions
- **Sub-memory level** : shape abstraction

Both components share most of their implementation

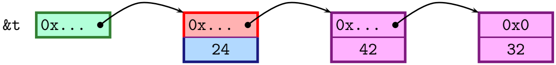
Contributions :

- a hierarchical shape abstraction
- integration on top of a regular shape abstraction
- extension of shape analysis operations

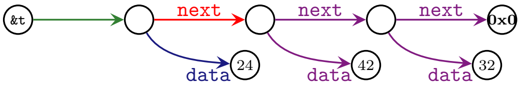
Abstraction of memory states : shape abstraction

- **Shape graphs** with **points-to** edges, and **inductive** edges
- **Nodes** denote **concrete values**, **edges** denote **memory regions**

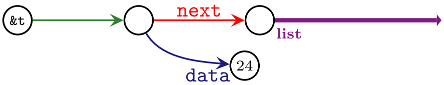
- **Memory splitting into regions**



- **Graph abstraction :** values, addresses $\longrightarrow$ nodes; cells $\longrightarrow$ edges



- **Region summarization :**



  - abstraction **parameterized** by a **set of inductive definitions**

Inductive definitions and segments

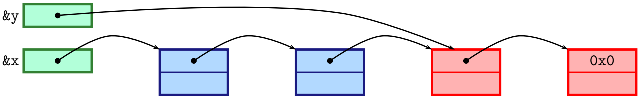
Inductive definitions in separation logic

- **Example :**

$$\alpha \cdot \mathbf{list} \ ::= \ (\mathbf{emp}, \alpha = 0) \ | \ (\alpha \cdot \mathtt{next} \mapsto \beta_0 * \alpha \cdot \mathtt{prio} \mapsto \beta_1 * \alpha \cdot \ldots \mapsto * \ \beta_0 \cdot \mathbf{list}, \alpha \neq 0)$$

- **User-supplied** (could be inferred automatically)

- **Full inductive edges** : complete structures
- **Segment edges** : structure segments



can be abstracted by :

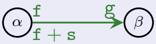
Abstraction of an atomic memory block

Shape graphs introduced in Laviron, Chang, Rival (ESOP'10) :

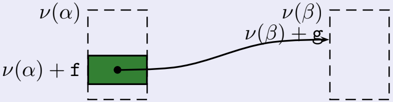
- describe **pointer arithmetics**
- represent **pointers to fields**, using a **(base,offset)** abstraction

Abstraction of a cell with a points-to edge $\alpha \cdot \mathtt{f} \mapsto \beta \cdot \mathtt{g}$

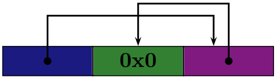
Valuation : ν : **Nodes** $\rightarrow$ **Values**



represents



Example : chain of pointers to fields in a given block



$$\begin{cases} \alpha \cdot 0 & \mapsto & \alpha \cdot 8 \\ \alpha \cdot 4 & \mapsto & \beta = \mathbf{0x0} \\ \alpha \cdot 8 & \mapsto & \alpha \cdot 4 \end{cases}$$

Abstraction of memory states and numeric contents

Product of shape graphs and **numerical** in Chang, Rival (POPL'08) :

- utilize a **numerical invariant** $\mathcal{N} \in \mathbb{D}_{\mathbf{num}}$ tied to the **nodes** of $\mathcal{G}$:

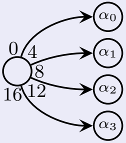
$$\gamma(\mathcal{G}, \mathcal{N}) = \{\hbar \mid \exists \nu : \textbf{Nodes} \to \textbf{Values},\ \nu \in \gamma(\mathcal{N}) \wedge (\hbar, \nu) \in \gamma(\mathcal{G})\}$$

- relies on a **cofibered abstraction** (Venet, SAS'96)

Example : abstraction of a sorted array of length 4

a_0	a_1	a_2	a_3

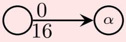
One abstract cell per concrete cell



$$\begin{cases} & \alpha_0 \leq \alpha_1 \\ \wedge & \alpha_1 \leq \alpha_2 \\ \wedge & \alpha_2 \leq \alpha_3 \end{cases}$$

Octagon numeric abstract domain

Alternate abstraction



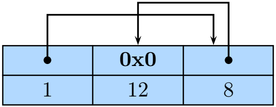
$\mathfrak{P}_{\texttt{sortedarray}}(\alpha)$

- A single **"fat"** points-to edge
- **Array specific abstraction**

Sub-memory predicate

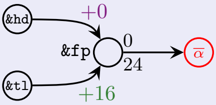
Principle

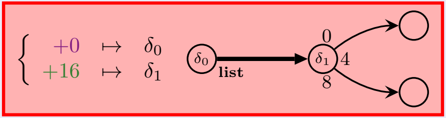
- The array zone : a **single, "fat"** points-to edge
- The contents : **a sub-memory** described by a **shape invariant**



(additional fields dropped for the sake of concision)

A two layers memory abstraction





- Node $\overline{\alpha}$ describes a **24 bytes long** value
- **Sub-memory predicate**, enclosing :
  - a **shape graph** : array contents viewed as a **memory** in itself
  - a **sub-environment** : mapping **main offsets** into **sub-nodes**

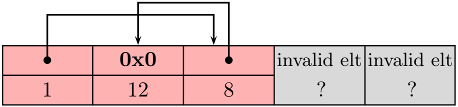
Hierarchical memory abstraction

Shape domain as an underlying numerical abstract domain

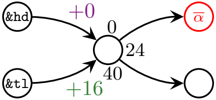
- **Abstract elements of $\mathbb{D}_{\textbf{sub}}$** are of the form $\mathbb{S}_{\overline{\alpha}}$**(Sub-env,Sub-graph)**
- The **concretization** of the whole domain is still of the form :

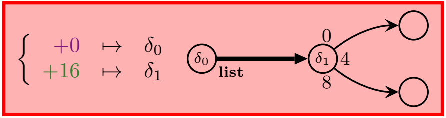
$$\gamma(\mathcal{G},\mathcal{N}) = \{\hbar \mid \exists \nu : \textbf{Nodes} \to \textbf{Values}, \nu \in \gamma(\mathcal{N}) \wedge (\hbar,\nu) \in \gamma(\mathcal{N})\}$$

Example, putting it all together :



is abstracted by :





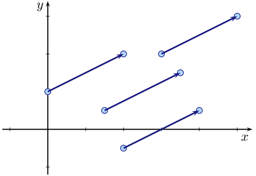
Outline

Abstract interpretation of a statement

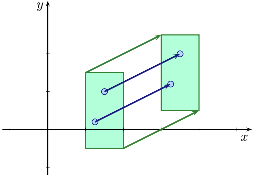
Computing sound abstract transfer functions

- **Conservative analysis** of concrete execution steps in the abstract
  e.g., **assignments**, **condition tests**...
- May **lose precision**, will **never forget any behavior**

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



concrete computation step



abstract transfer function

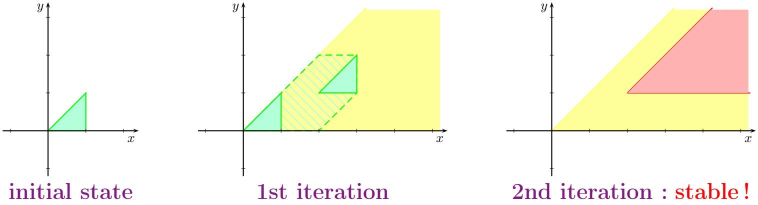
Soundness : all concrete behaviors are accounted for !

Abstract interpretation of a loop

Computing invariants about infinite executions with widening $\triangledown$

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

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

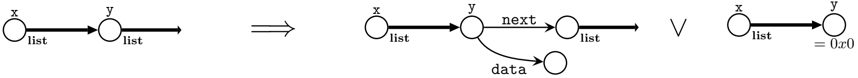


Soundness : all concrete behaviors are accounted for !

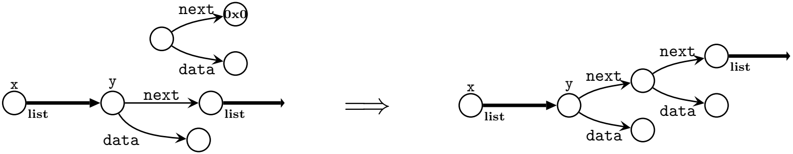
Algorithms underlying operations, shape abstraction

Foundation for transfer functions and widening

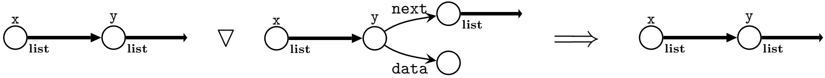
- **Unfolding** : cases analysis on summaries



- **Abstract postconditions**, on "exact" regions, e.g. insertion



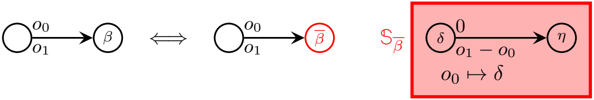
- **Folding** : builds summaries and ensures termination



Algorithms underlying operations, hierarchical abstraction

Foundation for transfer functions and widening

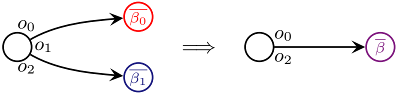
- **Introduction** of a sub-memory predicate :



$\mathcal{N} = [\ldots \wedge \eta = \beta]$

  - the fresh predicates says nothing new, but can be **later extended** (next slides)

- **Fusion** of *consecutive* sub-memory predicates



$$\left\{ \begin{array}{ll} & \mathbb{S}_{\overline{\beta_0}}(E_0, G_0) \\ \wedge & \mathbb{S}_{\overline{\beta_1}}(E_1, G_1) \end{array} \right. \qquad \mathbb{S}_{\overline{\beta}}(E_0 \uplus E_1, G_0 \uplus G_1)$$

  - merged predicates have to match **consecutive points-to edges**
  - result **joins** sub-graphs, and sub-environments

Analysis of an assignment

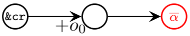
Next slides : **assignment transfer function** and **join / widening**

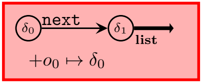
- Graphs are simplified : other fields than `next` are not shown

Inner loop traversal (**localisation** of the insertion position) :

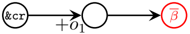
`cr = cr -> next ;`

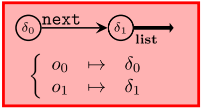
- **Pre-condition :**





- **Post-condition :**





Computation of the post

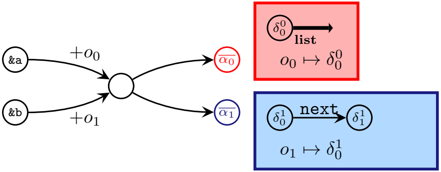
- **Modified** edge : inside the **main memory shape graph**
- **Effect** :
  - **update** of the **destination offset** into the new o_0'
  - o_0 could be dropped

Analysis of an assignment

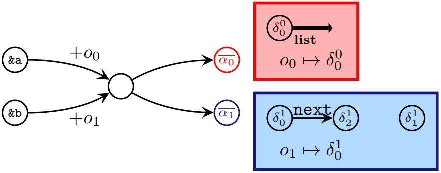
Assignment inside a sub-memory :

`b -> next = a;`

- **Pre-condition :**



- **Post-condition :**



$V = [\delta_0^0 = \delta_2^1]$

Computation of the post

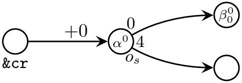
- **Modified** edge : inside the **sub-memory shape graph** attached to $\overline{\alpha_1}$
- **Effect** :
  - **edge replacement** towards **fresh node** δ_2^1
  - **equality constraint** to capture **equality across boundary**
  - δ_1^1 could be dropped

- If needed, **preliminary unfolding should apply**

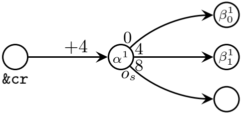
Hierarchical abstract join : introduction

Join after the **second iteration in the inner loop**, at the **first iteration in the outer loop** :

- **First argument :**

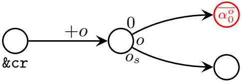


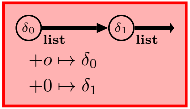
- **Second argument :**



$\beta_1^1 = \alpha^1$

- **Output :**





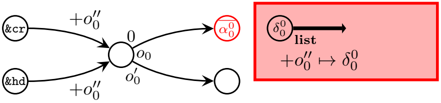
Computation

1. **Make elements compatible**
   - **introduction** at node β_0^0, in the first argument
   - **introductions** at nodes β_0^1, β_1^1, in the second argument, and fusion
2. **Main memory join :** shape abstract domain
3. **Sub-memory join :** also **classical** shape join

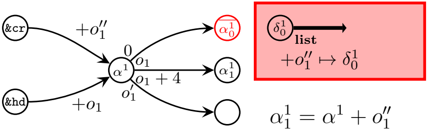
Hierarchical abstract join : extension

Inner loop second join :

- **First argument :**

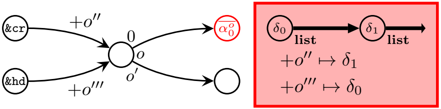


- **Second argument :**



$\alpha^1_1 = \alpha^1 + o''_1$

- **Output :**



Computation

1. **Make elements compatible**
   - **introduction** at node α^1_1, in the second argument
   - **fusion** of both second argument sub-memories

   similar to **array analyses**
2. **Main memory join :** shape abstract domain
3. **Sub-memory join :** also **classical** shape join

Outline

Implementation and results

Implementation inside the **MemCAD analyzer**

- **Two instances** of the **shape abstract domain**
- **Hierarchical abstraction** implemented as a **functor**

- Effectively, an integration of an **array analysis** to reason about **contiguous points-to edges**
- **Shape analysis algorithms** are reused as is, **unmodified**

Automatic verifcation (no runtime errors), analysis of code using **free-pool** compared to equivalent codes with **malloc** (more in the paper) :

Program	Allocation method	
	malloc	free-pool array
structure construction	0.195	**0.520**
structure traversal	0.056	**0.107**

Roughly, a 2X to 3X slowdown (but analysis of much **trickier code**)

Concluding remarks

Analysis of a trick code that is implementing their own malloc

- integration of **array abstract interpretation** techniques into a **separation logic based shape analyzer**
- **multi-level** view of the memory, matching the data-types

Achieved thanks to a **modular shape abstract domain**

(Mid term) future task :

remove the contiguousness assumption on sub-memories

- **Verification of memory managers**

(Long term) future work : integration into Astrée

- **Significant work** in abstract domain engineering
- **Verification of industrial code** user defined memory management