Shape Analysis Applied to C Code

Xavier Rival;
Joint work with Vincent Laviron and Bor-Yuh Evan Chang

INRIA Rocquencourt and École Normale Supérieure
eXtensible Inductive Shape Analyzer

- **Inductive data-structure definitions** user-supplied parameters

  **Examples:**
  
  ♦ **List definition:**

  \[
  \alpha \cdot \text{list}() := \alpha = 0 \land \text{emp} \\
  \lor \alpha \neq 0 \land \alpha@nxt \mapsto \beta \land \alpha@dat \mapsto \beta' \lor \beta \cdot \text{list}()
  \]

  ♦ **Doubly linked list definition:**

  \[
  \alpha \cdot \text{dll}(\beta) := \alpha = 0 \land \text{emp} \\
  \lor \alpha \neq 0 \land \alpha@nxt \mapsto \gamma \land \alpha@prev \mapsto \beta \lor \gamma \cdot \text{dll}(\alpha)
  \]

- **Analysis process**, based on **abstract interpretation**:
  1. the user supplies **inductive definitions** for relevant structures
  2. **pre-analyses** of the inductive definitions to make the analysis **more accurate**
  3. **automatic, sound analysis** of the C code
The Abstract Domain

- A graph representation for separation logic formulae
  - **symbolic nodes** ($\alpha, \beta, \gamma \ldots$): integer values
e.g., addresses, data...
  - **edges**: constraints about distinct memory regions
- Example: **points-to edges** (more edges on next slides)
  - denoted $\alpha@f \mapsto \beta$, means "at address $\alpha$ plus offset $f$, we read $\beta$"
  - graph representation:

  ![Graph Representation](image)

  - denotes stores like:

  ![Store Representation](image)
Field Separation Model

- **Concretization** maps distinct edges into disjoint regions.
- One graph stands for a set of concrete stores:

  e.g., \( \alpha \) stands for stores like:

  \[ @g \rightarrow \text{to addr} [\alpha] \]

  \[ @f \rightarrow \text{to addr} [\alpha] \]

  \[ @g \rightarrow \text{to addr} [\beta] \]

  \[ @f \rightarrow \text{to addr} [\beta] \]

  \[ @g \rightarrow \text{to addr} [\gamma] \]

  \[ @f \rightarrow \text{to addr} [\gamma] \]

  \[ @g \rightarrow \text{to addr} [\gamma] \]

  \[ @f \rightarrow \text{to addr} [\gamma] \]

  \[ @g \rightarrow \text{to addr} [\beta] = \text{to addr} [\gamma] \]

  \[ @f \rightarrow \text{to addr} [\beta] = \text{to addr} [\gamma] \]
Summarization

• **User-specified inductive definition** for **lists**:

\[
\alpha \cdot \text{list}() := \begin{align*}
\alpha &= 0 \land \text{emp} \\
\lor\quad \alpha &\neq 0 \land \alpha@\text{nxt} \leftrightarrow \beta \land \alpha@\text{dat} \leftrightarrow \beta' \land \beta \cdot \text{list}()
\end{align*}
\]

• **Abstraction of structures** defined by induction:

♦ **abstract form**: \( \alpha \cdot \text{list} \)

♦ it **abstracts** stores like:


\[
\begin{array}{c}
\text{to_addr}[\alpha] \\
@\text{nxt} \\
@\text{dat}
\end{array} \quad \text{to_addr}[\beta] \\
\begin{array}{c}
@\text{nxt} \\
@\text{dat}
\end{array}
\]

• Similar abstraction for **structure segments**:

\( \alpha \cdot \text{list} \rightarrow \beta \cdot \text{list} \) abstracts stores like

\[
\begin{array}{c}
\text{to_addr}[\alpha] \\
@\text{nxt} \\
@\text{dat}
\end{array} \quad \text{to_addr}[\alpha] \\
\begin{array}{c}
@\text{nxt} \\
@\text{dat}
\end{array}
\]
A Combined Abstraction

- Not all properties can be represented in this shape domain $\mathbb{D}_{\text{shape}}$:
  - equalities / inequalities of addresses
    some are expressed, but not all
  - numerical constraints

- We need a numerical abstraction $\mathbb{D}_{\text{num}}$:
  - $\mathbb{D}_{\text{num}}$ also composed of a product of abstractions
  - reduced product: $\mathbb{D}_p = \mathbb{D}_{\text{shape}} \times \mathbb{D}_{\text{num}}$
    actually, a bit more complicated than reduced product
• List reversal:

```
assume(l points to a list)
x = l; y = NULL;
while (x != NULL){
    t = x -> nxt;  x -> nxt = y;
    y = x;  x = t;
}
```

• At point ①, we need unfolding: a form of partial concretization
  ♦ consists in expanding the inductive definition list
  ♦ generates symbolic disjunctions of graphs

• For termination, we need to apply widening at point ②:
  ♦ over-approximates disjuncts
  ♦ a set of local weakening rules

• What about the lower-level aspects of the language?
  ⇒ the focus of this talk
Physical Memory Mapping

- Points-to-edges:
  \[ \gamma(\alpha@f \mapsto \beta) = \{ \text{to}_\text{addr}[\alpha] + \text{offset}[f] \mapsto \text{to}_\text{addr}[\beta] \} \]

- field offsets are taken into account already
- physical representation read in the ABI

- Abstraction of nested structures: (one contiguous region)

```c
struct foo{
    int f0;
    struct{
        int f1;
        int f2;
    } bar;
} x;
```

\[ \begin{array}{c|c}
@f0 & n_0 \\
@b.f1 & n_1 \\
@b.f2 & n_2 \\
\end{array} \]

\[ \blacktriangleleft \]

\[ \begin{array}{c}
\alpha \\
\text{bar} \cdot f1 \\
\text{bar} \cdot f2 \\
\end{array} \]

\[ \begin{array}{c}
\beta_0 \\
\beta_1 \\
\beta_2 \\
\end{array} \]
Pointers and Nested Structures

- Pointers into nested structures, e.g. $Y = \& (X \cdot \text{bar})$; results in

- Abstraction:
  - for cell $Y$: $\{\text{to_addr}[\alpha_0] \leftrightarrow \text{to_addr}[\alpha_1] + \text{offset}[	ext{bar}]\}$
  - we need another offset on the destination side:

- General case: edges representing

\[
\{\text{to_addr}[\alpha_0] + \text{offset}[f0] \leftrightarrow \text{to_addr}[\alpha_1] + \text{offset}[f1]\}
\]
Pointer Arithmetic (Field-Level)

- Abstraction of a pointer expression:
  - sets of pairs “node + offset” \((\alpha, \text{off})\)
  - computation over \text{off} should be performed in \(\mathbb{D}\)\_\text{num}
  - at field level, pointer arithmetic reduces to offset computations
- Abstraction of an l-value: results in a points-to edge

\[
\begin{array}{c}
\alpha_0 \\
\downarrow f_0 \\
\alpha_1 \\
\end{array}
\]

- read content, dereference: edge target, i.e., \((\alpha_0, f_0)\)
  - in the case of data, such pairs should be evaluated in \(\mathbb{D}\)\_\text{num}
- address of \((\&):\) edge source, i.e., \((\alpha_1, f_1)\)

Note that unfolding inductive edges may turn out necessary
Sizes of Data-Types

• Another example structure:

```c
struct foo{
    short f0;
    int f1;
}
x;
```

• Offsets are defined by the ABI (existence of padding bytes)
  same for field sizes

• Size of memory cells needs be represented in the abstract level
  ♦ new abstraction (with a node for padding bytes):

  ![Graphical representation of memory cells](image)

  In a 32-bits machine, \(f0 = 0; f1 = 4\)

  ♦ consistency: edges should not overlap (... for now)
Union Types and Casts

- Let us consider the type:

```c
union{
    int u0;
    struct{
        char f0;
        char f1;
    } u1;
}
```

- Several ways to access the same cell:

- Casts typically result in similar issues
Using the Logical Conjunction Connector

- We need to maintain **several views** of **union** contents
  
  In our example:

  ![Diagram](image)

- Can be expressed using **logical conjunction**

- **Consequences on analysis algorithms:**
  - **Inductive definitions** account for distinct views
    thus **unfolding** generates conjunctions
  - **Translations** between views, e.g., from `int` to 4 `char` and back
  - **Join:**
    - similar views can be maintained
    - other views can safely be **dropped**

- **Logical conjunctions are local** (cost in join)
Memory Allocation

- **Semantics of malloc**(\textit{size}):  
  - either returns a valid pointer to a new region of size \textit{size}  
    the content of this region is undefined  
  - or fails, and returns 0

- **Analysis of the successful case:**  
  1. create a new node  
  2. add no outgoing edge  
    i.e., an empty conjunction of edges, with size \textit{size}

- **Failed case** should also be treated  
  programmers often use a wrapper to exit cleanly
Memory Free

- Not all pointers can be freed

![Diagram showing pointers and malloced regions]

- We can safely free $X$ ($Y$ becomes dangling)
- We cannot free $Y$!

- Semantics:
  - Only pointers to a malloced region can be disposed
  - A table of malloced regions is maintained

- Analysis:
  - For sound analysis of `free`, we need to abstract that table
  - Property to express in $\mathbb{D}_{\text{num}}$: $\alpha$ refers to a malloced region of size $s$
Summary

- Several extensions to points-to edges allow coping with
  - nested structures, union types
  - field level pointer arithmetic
  - size of memory cells
  - memory management
- These elements are reflected in
  - user supplied data-structure inductive definitions
  - folding / unfolding algorithms
- Implementation in the analyzer is mostly done
Abstraction of Array

- Arrays are contiguous memory regions
  - similar techniques should apply
  - abstractions of an integer array of length 3:

- Advantage: existing array abstractions can be used
  e.g., smashing, expansion...

- Additional issues:
  - length is not always static
    edge size defined by a symbolic name
  - shape content: e.g., an array of lists
    array base element abstraction might refer to shapes: not in $\mathbb{D}_{\text{num}}$
Unfolding Array Cells

- **Edge splitting**, e.g., in order to read a cell:
  
  if $\mathbb{D}_{\text{num}}$ proves $0 < \delta \leq \epsilon - s$, we can unfold

  $\alpha$ 0 $\epsilon$ bytes $\beta$  

  into

  $\alpha$ $\delta$ bytes $\beta_0$ 
  $\beta_1$ $s$ bytes 
  $\beta_2$ $\epsilon - \delta - s$ bytes 

  materialization of a cell (to write / read)

- Requires good support from $\mathbb{D}_{\text{num}}$
  especially for folding (widening)
Abstraction of the Calling Stack