Abstracting Calling Context with Shapes

Xavier Rival
INRIA Rocquencourt & ENS Paris

Bor-Yuh Evan Chang
University of Colorado at Boulder
Interprocedural analysis is important

- Static analysis
  
  sound and automatic inference of program properties
  
  - shape analysis: heap properties
  - value analysis: numerical, boolean invariants

- Procedures are central to programming
  
  ⇒ thus, interprocedural analysis is key to program reasoning

- Challenges:
  
  - choice in abstraction: of functions or of calls
  - recursion: unbounded heap and stack
### Two main approaches to interprocedural analysis

<table>
<thead>
<tr>
<th>“tabulation” approach</th>
<th>“inlining” approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>analyze each definition</td>
<td>analyze each call</td>
</tr>
<tr>
<td>abstracts $\mathcal{P}(\mathcal{S} \rightarrow \mathcal{S})$</td>
<td>abstracts $\mathcal{P}(\mathcal{S})$</td>
</tr>
<tr>
<td>+ modularity</td>
<td>- not modular</td>
</tr>
<tr>
<td>+ reuse of invariants</td>
<td>- re-analysis in $\neq$ contexts</td>
</tr>
<tr>
<td>- deals with state relations</td>
<td>+ deals with states</td>
</tr>
<tr>
<td>- complex tabulation strategy</td>
<td>+ straightforward iteration</td>
</tr>
<tr>
<td>[Reps et al.'95 ]</td>
<td>[Rinetzky et al.'01 ]</td>
</tr>
<tr>
<td>challenge: frame problem</td>
<td>challenge: unbounded calls</td>
</tr>
</tbody>
</table>

- deals with state relations
- complex tabulation strategy

[Reps et al.'95 ]

[Rinetzky et al.'01 ]
Challenges of the tabulation approach

• Modular approach: requires an abstraction for the effect of functions
  ♦ denotational semantics $\llbracket f \rrbracket : S_{pre} \rightarrow S_{post}$
  ♦ we need to abstract $P(S_{pre} \rightarrow S_{post})$

relations are harder to abstract than states

• Call graph analysis: computes tables of pre and post-conditions

• Table entry reuse: frame inference problem

Abstracting Calling Context with Shapes – p.4/24
Challenges of the inlining method

- Analyze each procedure in its context: very context sensitive analysis
  - no procedure summaries
  - analysis in the calling context

- Unbounded calls

  \[ f^0, x \rightarrow f^1, x \rightarrow f^2, x \rightarrow \ldots \]

  → how to guarantee finite abstractions and termination?

- Shape analysis to the rescue!
  - [Rinetzky, CC’01]: TVLA based abstraction of call stacks
  - Outstanding difficulties: requires heavy instrumentation

Abstracting Calling Context with Shapes – p.5/24
Our proposal

Apply *inductive shape* analysis to summarize unbounded calling contexts in a whole program, *state-based interprocedural analysis*

Implications:

- **Parametric interprocedural analysis**
  - supports the choice of base domain
  - extends existing shape analyzer Xisa
- **Very context sensitive** treatment of *procedure calls*
- **Stack summarization** together with the heap
Outline

- Approaches to interprocedural analysis
  √ Principles of shape analysis with inductive definitions
    - Abstracting calling contexts
    - Inferring calling context summaries
    - Conclusion
Memory abstraction using separating shape graphs

- Memory splitting into regions

- Graph abstraction: \( \{ \text{values, addresses} \rightarrow \text{nodes}, \text{cells} \rightarrow \text{edges} \) \)

- Region summarization:

  - Abstraction parameterized by a set of inductive definitions
  - Defines a concretization relation
Inductive shape analysis algorithms

- **Unfolding**: splitting of summaries

\[
\begin{align*}
\text{x} & \quad \text{y} \\
\text{list} & \quad \text{list} \\
\Rightarrow & \\
\text{x} & \quad \text{y} \\
\text{list} & \quad \text{list}
\end{align*}
\]

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

\[
\begin{align*}
\text{x} & \quad \text{y} \\
\text{next} & \quad \text{data} \\
\text{list} & \quad \text{list} \\
\Rightarrow & \\
\text{x} & \quad \text{y} \\
\text{next} & \quad \text{data} \\
\text{list} & \quad \text{list}
\end{align*}
\]

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

\[
\begin{align*}
\text{x} & \quad \text{y} \\
\text{next} & \quad \text{data} \\
\text{list} & \quad \text{list} \\
\Rightarrow & \\
\text{x} & \quad \text{y}
\end{align*}
\]
Inductive definitions

- Concrete structure, e.g., a doubly linked list

- Inductive structure described by a logical formula

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

- Corresponds to graph unfolding rules:

Parameters capture non trivial pointer relations

- Parametric abstract domain:
  - generic algorithms for unfolding, folding, abstract post-conditions...
  - inductive definitions are parameters, depending on the data-structures
Widening operator

- **Widening** is needed to enforce termination of the analysis

\[
\text{list} \xrightarrow{\text{y}} \text{list} \quad \nabla
\]

- Region matching

\[
\text{list} \xrightarrow{\text{y}} \text{list} \quad \nabla
\]

- Semantic rules for per region weakening

\[
\text{list} \xrightarrow{\text{y}} \quad \nabla
\]

- Widening:
Challenges

- Model and summarize the stack and the heap together
- Infer stack inductive definitions
  - for heap structures: user-supplied definitions
  - for the call stack: unreasonable to expect the user to supply them...
    ⇒ call stack inference problem
- Ensure original shape analysis algorithms work
Outline

- Approaches to interprocedural analysis
- Principles of shape analysis with inductive definitions

√ Abstracting calling contexts
  - Inferring calling context summaries
  - Conclusion
**Challenges with recursion**

```c
void main()
{
    dll ∗ l; // assume l points to a sll
    l = fix(l, NULL);
}
dll ∗ fix(dll ∗ c, dll ∗ p){
dll ∗ ret;
    if(c!=NULL){
        c->prev = p;
        ▶ c->next = fix(c->next, c);
        if(check(c->data)){
            ret = c->next;
            remove(c);
        } else ret = c;
    }
    return ret;
}
```

- Heap is unbounded, needs abstraction (shape analysis)
- But stack may also grow unbounded, needs abstraction
- Complex relations between both stack and heap

{ turns a singly linked list into a doubly linked list
removes some elements

```
main 1
|fix| c 0 |
|   | ret ? |

main 1
|fix| c 11 |
|   | ret ? |
```

Abstracting Calling Context with Shapes – p.14/24
Calling contexts as shape graphs

- Concrete assembly call stack modelled in a separating shape graph
  - one node per activation record address
**Calling contexts as shape graphs**

- Concrete assembly call stack modelled in a separating shape graph
  - one node per activation record address
Calling contexts as shape graphs

- Concrete assembly call stack modelled in a separating shape graph
  - one node per activation record address
  - explicit edges for frame pointers
Calling contexts as shape graphs

- Concrete assembly call stack modelled in a separating shape graph
  - one node per activation record address
  - explicit edges for frame pointers
  - local variables turn into activation record fields
Calling contexts as shape graphs

- Concrete assembly call stack modelled in a separating shape graph
  - one node per activation record address
  - explicit edges for frame pointers
  - local variables turn into activation record fields
Calling contexts as shape graphs

- Concrete assembly call stack modelled in a separating shape graph
  - one node per activation record address
  - explicit edges for frame pointers
  - local variables turn into activation record fields
Inductive structure of the calling context

- Repeating pattern: evidence of an inductive structure

The inductive based summarization applies...

- Inductive structure of the call stack captured by the rule

\[
\text{fix}::\text{ctx} \rightarrow \text{stack}(\beta_1, \beta_2) \xrightarrow{\sim} \text{unfold}
\]

- Unfolds one activation record

- Challenge: discover this rule automatically (in a few slides...)

Abstracting Calling Context with Shapes – p.16/24
Summarizing calling contexts

Third iteration with no summarization

Shape graph with summarization

- Top activation record still exposed
- Summarized contexts of the form $\text{fix} \cdot \text{fix}^+ \cdot \text{main}$
Outline

- Approaches to interprocedural analysis
- Principles of shape analysis with inductive definitions
- Abstracting calling contexts

✓ Inferring calling context summaries
- Conclusion
Graph subtraction for stack rule inference

- **Subtraction of two consecutive iterates:**

  ![Graph subtraction diagram]

  - Top-most activation records match **should remain exposed**
Graph subtraction for stack rule inference

- **Subtraction of two consecutive iterates:**

- **Top-most activation records match** should remain exposed
Graph subtraction for stack rule inference

- Subtraction of two consecutive iterates:

  - Top-most activation records match should remain exposed
  - Bottom of the call stack match
  - Difference: one activation record + heap “footprint”
    \[ \implies \text{generates an inductive rule} \]
  - Additional work: gathering of node relations, numerical predicates

Abstracting Calling Context with Shapes – p.19/24
Graph subtraction for stack rule inference

- **Subtraction of two consecutive iterates:**

![Diagram showing graph subtraction for stack rule inference]

**Inferred rule:**

\[
\text{fix}::\text{ctx} \rightarrow \text{unfold}
\]

Abstracting Calling Context with Shapes – p.19/24
Fixpoint computation

- **Iteration 1, Iteration 2 and fixpoint iterates:**

  ![Diagrams showing fixpoint computation]

  - **Insight:**
    - the sequence of calls builds the stack
    - subtraction “recognizes” the structure being built
    - graph widening proves its preservation

- **Upon return:**
  unfolding of stack summaries and widening over return sequences
Experience

- **Micro-benchmarks**
  - iterative programs: one fixpoint computation per loop
  - recursive programs: stack rule inference
    - + two fixpoint computations (calls and returns)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Recursive (ms)</th>
<th>Iterative (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>list insertion nth element</td>
<td>48</td>
<td>16</td>
</tr>
<tr>
<td>list remove nth element</td>
<td>27</td>
<td>11</td>
</tr>
<tr>
<td>list deletion (memory free)</td>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>list append</td>
<td>20</td>
<td>13</td>
</tr>
</tbody>
</table>

- Roughly 3x slowdown: performance in line with the iterative code analysis

- **Case study** discussing precision wrt tabulation algorithm (paper):
  - stack inductive definition can capture data invariants too
  - high context sensitivity (with or without heap shape)
Outline

- Approaches to interprocedural analysis
- Principles of shape analysis with inductive definitions
- Abstracting calling contexts
- Inferring calling context summaries

✓ Conclusion
Conclusion

• Extension of widening with subtraction to synthesize rules
  ♦ previously: generic widening for user supplied templates
  ♦ subtraction: generic widening + inferred templates from knowing when the structure is built

• Alternative view on interprocedural analysis
  ♦ very context sensitive abstraction
  ♦ straightforward extension of Xisa
    ▶ abstraction of a concrete semantic model for procedures
    ▶ core analysis algorithms remain unchanged

• Perspective: hybrid approach to interprocedural analysis, e.g.:
  ♦ tables for abstracting a library interface
  ♦ summarization for analyzing complex private procedures