Announcements

- “Last day”
- Presentations next week
  - Send me your title
  - Upload your slides at least 30 min before class
- Papers due May 5
- Extra credit opportunity: peer review of papers (due May 6)

---

**End-User Program Analysis for Data Structures**

Bor-Yuh Evan Chang  
University of Colorado

Collaborators: Xavier Rival (INRIA), George C. Necula (UC Berkeley)

---

Software errors cost a lot

~$60 billion annually (~0.5% of US GDP)
- 2002 National Institute of Standards and Technology report

> total annual revenue of Microsoft
> 10x annual budget of NSF

---

But there’s hope in program analysis

**Microsoft** uses and distributes the **Static Driver Verifier**

**Airbus** applies the **Astrée Static Analyzer**

Companies, such as Coverity and Fortify, market static source code analysis tools

---

Because program analysis can eliminate entire classes of bugs

For example,
- Reading from a closed file: `read( );`  
- Reacquiring a locked lock: `acquire( );`

How?
- Systematically examine the program
- Simulate running program on “all inputs”
- “Automated code review”
Program analysis by example: Checking for double acquires

Simulate running program on “all inputs”

```c
// x now points to an unlocked lock
acquire(x);
```

Program analysis by example: Checking for double acquires

Simulate running program on “all inputs”

```c
// x now points to an unlocked lock in a linked list
acquire(x);
```

Must abstract

```c
// x now points to an unlocked lock in a linked list
acquire(x);
```

To address the precision challenge

Traditional program analysis mentality:

"Why can’t developers write more specifications for our analysis? Then, we could verify so much more."

"Since developers won’t write specifications, we will use default abstractions (perhaps coarse) that work hopefully most of the time."

End-user approach:

"Can we design program analyses around the user? Developers write testing code. Can we adapt the analysis to use those as specifications?"

Overview of contributions

Extensible Inductive Shape Analysis

Precise inference of data structure properties
Able to check, for instance, the locking example
Targeted to software developers
Uses data structure checking code for guidance
- Turns testing code into a specification for static analysis
Efficient
- 10-100x speed-up over generic approaches
- Builds abstraction out of developer-supplied checking code

Summary of overview

Challenge in analysis: Finding a good abstraction
precise enough but not more than necessary

Powerful, generic abstractions
expensive, hard to use and understand
Built-in, default abstractions
often not precise enough (e.g., data structures)

End-user approach:

Must involve the user in abstraction
without expecting the user to be a program analysis expert
Extensible Inductive Shape Analysis

Precise inference of data structure properties

End-user approach

Shape analysis is a fundamental analysis

Data structures are at the core of
- Traditional languages (C, C++, Java)
- Emerging web scripting languages

Improves verifiers that try to
- Eliminate resource usage bugs (locks, file handles)
- Eliminate memory errors (leaks, dangling pointers)
- Eliminate concurrency errors (data races)
- Validate developer assertions

Enables program transformations
- Compile-time garbage collection
- Data structure refactorings

Shape analysis is not yet practical

Choosing the heap abstraction difficult for precision

Traditional approaches:
- TVLA [Sagiv et al.]
- Parametric in low-level, analyzer-oriented predicates
  - Very general and expressive
  - Hard for non-expert
- Space Invader [Distefano et al.]
- Built-in high-level predicates
  - Hard to extend
  - No additional user effort (if precise enough)

End-user approach:
- Parametric in high-level, developer-oriented predicates
  - Extensible
  - Targeted to developers

Shape analysis by example:
Removing duplicates

Code Review / Static Analysis

```plaintext
// l is a sorted doubly-linked list
for each node cur in list l {
  remove cur if duplicate;
}
assert l is sorted, doubly-linked with no duplicates;
```

```
Example / Testing

assert l is sorted, doubly-linked with no duplicates;
```

```plaintext
// l is a sorted doubly-linked list
for each node cur in list l {
  remove cur if duplicate;
}
assert l is sorted, doubly-linked with no duplicates;
```

```
Example / Testing

assert l is sorted, doubly-linked with no duplicates;
```

```
assert l is sorted, doubly-linked with no duplicates;
```

```
assert l is sorted, doubly-linked with no duplicates;
```

Key insight

for being developer-friendly and efficient

Utilize “run-time checking code” as specification for static analysis.

```
dll(h, p) =
if (h = null)
  then true
else
  h \rightarrow prev = p and dll(h \rightarrow next, h)

assert(sorted_dll((..., p, (...));
```

```
checkers

• p specifies where prev should point
```

Contribution:

Build the abstraction for analysis out of developer-supplied checking code

Contribution:

Automatically generalize checkers for complicated intermediate states

Our framework is ...

An automated shape analysis with a precise memory abstraction based around invariant checkers.

- Extensible and targeted for developers
  - Parametric in developer-supplied checkers
- Precise yet compact abstraction for efficiency
  - Data structure-specific based on properties of interest to the developer
Shape analysis is an abstract interpretation on abstract memory descriptions with ...

**Splitting** of summaries

![Diagram showing the splitting process]

To reflect updates precisely

And **summarizing** for termination

---

**Outline**

- Learn information about the checker to use it as an abstraction
- Compare and contrast manual code review and our automated shape analysis

---

**Overview: Split summaries to interpret updates precisely**

Want abstract update to be "exact", that is, to update one "concrete memory cell".

The example at a high-level: iterate using `cur` changing the doubly-linked list from purple to red.

**Challenge:**

How does the analysis "split" summaries and know where to "split"?

---

**"Split forward"** by unfolding inductive definition

![Diagram showing the split forward process]

Analysis doesn’t forget the empty case

---

**"Split backward"** also possible and necessary

![Diagram showing the split backward process]

Technical Details:

- How does the analysis do this unfolding?
- Why is this unfolding allowed?
- (Key: Segments are also inductively defined) [POPL'08]

How does the analysis know to do this unfolding?

---

**Outline**

Derives additional information to guide unfolding

How do we decide where to unfold?

---

**Contribution:**

Turns testing code into specification for static analysis
Abstract memory as graphs
Make endpoints and segments explicit, yet high-level

Types for deciding where to unfold
Summary

Summary of checker parameter types
Tell where to unfold for which fields
Make analysis robust with respect to how checkers are written
Learn where in summaries unfolding won’t help
Can be inferred automatically with a fixed-point computation on the checker definitions

Summary of interpreting updates
Splitting of summaries needed for precision
Unfolding checkers is a natural way to do splitting
When checker traversal matches code traversal
Checker parameter types
Enable, for example, “back pointer” traversal without blindly guessing where to unfold

Outline
1. splitting and interpreting update
2. type inference on checker definitions
3. summarizing
abstract interpretation
shape analyzer
Summarize by folding into inductive predicates

```c
last = l; cur = l->next;
while (cur != null) {
  // ... cur, last ...
  if (...) last = cur;
  cur = cur->next;
}
```

Previous approaches guess where to false for each graph.

**Contribution:**
Determine where by comparing graphs across history

**Challenge:** Precision (e.g., last, cur separated by at least one step)

---

Summary: Given checkers, everything is automatic

- **Type inference on checker definitions**
- **Abstract interpretation**
- **Splitting and interpreting update**
- **Summarizing**
- **Shape analyzer**

---

Results: Performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Max. Num. Graphs at a Program Pt</th>
<th>Analysis Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>singly-linked list reverse</td>
<td>1</td>
<td>TVLA: 290 ms</td>
</tr>
<tr>
<td>doubly-linked list reverse</td>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>doubly-linked list copy</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td>doubly-linked list remove</td>
<td>5</td>
<td>5.4</td>
</tr>
<tr>
<td>search tree with parent insert</td>
<td>3</td>
<td>17.9</td>
</tr>
<tr>
<td>search tree with parent insert and back</td>
<td>5</td>
<td>TVLA: 850 ms</td>
</tr>
<tr>
<td>two-level skip list rebalance</td>
<td>1</td>
<td>11.7</td>
</tr>
<tr>
<td>Linux scull driver</td>
<td>4</td>
<td>2969.6</td>
</tr>
</tbody>
</table>

Times negligible for data structure operations (often in sec or 1/10 sec)

---

Demo: Doubly-linked list reversal

- **Body of loop over the elements:**
  - Swaps the `next` and `prev` fields of `curr`.

- **Already reversed segment**
- **Node whose `next` and `prev` fields were swapped**
- **Not yet reversed list**

http://xisa.cs.colorado.edu

---

Experience with the tool

- **Checkers are easy to write** and try out
  - Enlightening (e.g., red-black tree checker in 6 lines)
  - Harder to "reverse engineer" for someone else’s code
  - Default checkers based on types useful

- **Future expressiveness and usability improvements**
  - **Pointer arithmetic and arrays**
  - More generic checkers:
    - polymorphic
    - higher-order parameterized by other predicates

- **Future evaluation:** user study

---

Summary of Extensible Inductive Shape Analysis

**Key Insight:** Checkers as specifications

- Developer View: Global, Expressed in a familiar style
- Analysis View: Capture developer intent, not arbitrary inductive definitions

**Constructing the program analysis**

- Intermediate states: **Generalized segment** predicates
- Splitting: Checker parameter **types with levels**
  - `h : [next(0), prev(0)]`  `p : [next(1), prev(1)]`
- Summarizing: **History-guided approach**
Conclusion

Extensible Inductive Shape Analysis
precision demanding program analysis
improved by novel user interaction

Developer: Gets results corresponding to
intuition
Analysis: Focused on what’s important to
the developer

Practical precise tools for better software
with an end-user approach!

What can inductive
shape analysis do for you?

http://xisa.cs.colorado.edu