

Announcements

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

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 2

End-User Program Analysis for Data Structures

Bor-Yuh Evan Chang
University of Colorado


April 22, 2008

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 

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 4

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

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 5

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”

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 6

Program analysis by example: Checking for double acquires

Simulate running program on “all inputs”

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

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 7

Program analysis by example: Checking for double acquires

Simulate running program on “all inputs”

```
... code ...
// x now points to an unlocked lock in a linked list
```

acquire(x);
... code ...

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 8

Must abstract

Abstraction too coarse or **not precise** enough (e.g., lost x is always unlocked)

```
... code ...
// x now points to an unlocked lock in a linked list
```

acquire(x); ✘
... code ...

mislabeled good code as buggy

For decidability, must **abstract**—“model all inputs” (e.g., merge objects)

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 9

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?”

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 10

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

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 11

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

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 12

End-user approach

Extensible Inductive Shape Analysis

Precise inference of data structure properties

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

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 14

Shape analysis by example: Removing duplicates

Example/Testing

```
// 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;
```

Code Review/Static Analysis

program-specific intermediate state more complicated

"sorted dl list"

"segment with no duplicates"

"no duplicates"

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 15

Shape analysis is not yet practical

Choosing the heap abstraction difficult for precision

Traditional approaches:

TVLA [Sagiv et al.]

- + 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:

Xfsa

- + Parametric in high-level, developer-oriented predicates
- + Extensible
- + Targeted to developers

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 16

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->prev == p and
    dll(h->next, h)
  checker
  • p specifies where prev should point
```

Contribution: Build the abstraction for analysis out of developer-specified checking code

Contribution: Automatically generalize checkers for complicated intermediate states

assert(sorted_dll_nodup(l,...));

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 17

Our framework is ...

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

```
dll(h, p) =
  if (h == null) then
    true
  else
    h->prev == p and
    dll(h->next, h)
  checkers
```

→

shape analyzer

- 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

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 18

Shape analysis is an abstract interpretation on abstract memory descriptions with ...

Splitting of summaries

To reflect updates precisely

And **summarizing** for termination

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 19

Outline

Learn information about the checker to use it as an abstraction

checkers

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

2 type inference on checker definitions

1 splitting and interpreting update

Compare and contrast manual code review and our automated shape analysis

shape analyzer

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 20

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”?

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 21

“Split forward” by unfolding inductive definition

Analysis doesn't forget the empty case

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

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 22

“Split backward” also possible and necessary

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?

```

true
else
  h->prev == p and
  dll(h->next, h)

```

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 23

Outline

Derives additional information to guide unfolding

checkers

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

2 type inference on checker definitions

1 splitting and interpreting update

3 summarizing

abstract interpretation

shape analyzer

Contribution: Turns testing code into specification for static analysis

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 24

Abstract memory as graphs

Make endpoints and segments explicit, yet high-level

Contribution: Generalization of checker (Intuitively, $dll(\alpha, null)$ up to $dll(\gamma, \beta)$.)

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 25

Types for deciding where to unfold

Summary

Checker "Run" (call tree/derivation)

-2	$dll(\alpha, null)$
-1	$dll(\beta, \alpha)$
0	$dll(\gamma, \beta)$
1	$dll(\delta, \gamma)$
	$dll(null, \delta)$

Checker Definition

$h: \{next(0), prev(0)\}$
 $p: \{next(-1), prev(-1)\}$

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

Says:
 For $h \rightarrow next / h \rightarrow prev$, unfold from h
 For $p \rightarrow next / p \rightarrow prev$, unfold before h

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 26

Types make the analysis robust with respect to how checkers are written

Doubly-linked list checker (as before)

Summary

Alternative doubly-linked list checker

Summary

Different types for different unfolding

$h: \{next(0), prev(0)\}$
 $p: \{next(-1), prev(-1)\}$
 $dll(h, p) =$
 if $(h = null)$ then true
 else $h \rightarrow prev = p$ and $dll(h \rightarrow next, h)$

$h: \{next(0), prev(-1)\}$
 $dll(h) =$
 if $(h \rightarrow next = null)$ then true
 else $h \rightarrow next \rightarrow prev = h$ and $dll(h \rightarrow next)$

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 27

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

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 28

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

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 29

Outline

Bor-Yuh Evan Chang · End-User Program Analysis for Data Structures 30

Summarize by folding into inductive predicates

```

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

```

Previous approaches guess where to fold for each graph.

Contribution: Determine where by comparing graphs across history

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

Bor-Yuh Evan Chang - End-User Program Analysis for Data Structures 31

Summary: Given checkers, everything is automatic

```

diff(h, p) =
  if (h == null) then
    true
  else
    h->prev == prev and
    diff(h->next, h)

```

Bor-Yuh Evan Chang - End-User Program Analysis for Data Structures 32

Results: Performance

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

Benchmark	Max. Num. Graphs at a Program Pt	Analysis Time (ms)
singly-linked list reverse	1	TVLA: 290 ms } 1.0
doubly-linked list reverse	Space Invader only analyzes lists (built-in)	1.5
doubly-linked list copy		5.4
doubly-linked list remove		17.9
doubly-linked list remove and back		18.1
search tree with parent insert	3	TVLA: 850 ms } 16.6
search tree with parent insert and back	5	64.7
two-level skip list rebalance	1	11.7
Linux scull driver (894 loc) (char arrays ignored, functions inlined)	4	3969.6

Verified shape invariant as given by the checker is preserved across the operation.

Bor-Yuh Evan Chang - End-User Program Analysis for Data Structures 33

Demo: Doubly-linked list reversal

```

prev = curr->prev;
next = curr->next;
curr->prev = next;
curr->next = prev;
prev = curr;
curr = next;

```

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

http://xisa.cs.berkeley.edu

Bor-Yuh Evan Chang - End-User Program Analysis for Data Structures 34

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 "element kind unspecified"
 - higher-order parameterized by other predicates

Future evaluation: user study

Bor-Yuh Evan Chang - End-User Program Analysis for Data Structures 35

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

$h : \{next(0), prev(0)\} \quad p : \{next(-1), prev(-1)\}$

Summarizing: History-guided approach

Bor-Yuh Evan Chang - End-User Program Analysis for Data Structures 36

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!

Bor-Yuh Evan Chang - End-User Program Analysis for Data Structures

37



*What can inductive
shape analysis do for you?*

<http://xisa.cs.berkeley.edu>