Calling Context Abstraction with Shapes

Xavier Rival
INRIA/ENS Paris

Bor-Yuh Evan Chang 張博聿
U of Colorado, Boulder

National Taiwan University - December 17, 2010
Work to Appear in POPL 2011
Programming Languages Research at the University of Colorado, Boulder
Interprocedural analysis is important

Procedures **Central** to Programming

```java
int f(int x) { ... }
let f x = ... 
function f(x) { ... }
```

Interprocedural Analysis **Key** to Program Reasoning
Two approaches to interprocedural analysis

**Each procedure separately**

“build procedure summaries”

- modular
- reuse
- pre/post invariants
- abstracts state
- complex tabulation strategy

**Whole program simulation**

“simulate inlined procedures”

- non-modular
- reanalyze procedure at each call site
- abstracts states
- simple iteration

analyze each definition vs. each call

infer abstractions of pairs vs. individual states
Two approaches to interprocedural analysis

State of Practice:
Almost all interprocedural analyses today

This Talk:
Using shape analysis techniques

Challenge: Frame Inference
Challenge: Unbounded Calls

“infinite inlining”
Our approach is to ...

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

- “Very” context-sensitive
  - Simultaneous summarization of the stack and heap
- Use simpler base domains with precision
  - Need only abstract sets of states not relations
Shape analysis is an abstract interpretation on abstract memory descriptions with ...

Splitting of summaries

To reflect updates precisely

And summarizing for termination

Shape analysis is an abstract interpretation on abstract memory descriptions with “shapes”

Here, summarize the call stack of activation records with “shapes”

Bor-Yuh Evan Chang, University of Colorado at Boulder
Challenge: Obtain stack inductives

Xisa is a **shape analysis** with an precise abstraction based around user-supplied **invariant checkers**.

```
fun h.dill(p) = 
  if (h = null) then true
  else h\rightarrow prev = prev and h\rightarrow next.dill(h)
```

- Reasonable to expect user-supplied inductive definitions for user-defined heap structures
- Unreasonable to expect inductive definitions describing possible call stacks.
  - **Contribution**: derived automatically

Bor-Yuh Evan Chang 張博聿, University of Colorado at Boulder
Roadmap

- Background: Memory as graphs
- Abstracting calling contexts
- Deriving inductive cases for calling context summarization
Memory as separating shape graphs

Analogous to separation logic formulas

\[ h.\text{dll}(p) = \begin{cases} \text{true} & \text{if } (h = \text{null}) \\ \text{else} & h \rightarrow \text{prev} = p \text{ and } h \rightarrow \text{next. dll}(h) \end{cases} \]
Unfolding inductive summaries

Possible unfoldings give an inductive definition
Roadmap

- Background: Memory as graphs
- Abstracting calling contexts
- Deriving inductive cases for calling context summarization
Concrete view of a recursive example

```c
void main() {
    ...
    l = fix(l, NULL);
}
// c is a singly-linked list
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 c;
}
```

Bor-Yuh Evan Chang 張博聿, University of Colorado at Boulder
Putting calling contexts into shape graphs

- **activation record base**
- **“frame pointer”**
- **Need summarization**
- **local variables are fields of activation records**
- **heap summary**
- **eliding data field**

Bor-Yuh Evan Chang, University of Colorado at Boulder
Calling context is a list

Inductive structure always a “list”

Summarize calling context using an inductive predicate stack whose definition is derived “on-the-fly”

“Node” kinds program-specific
Calling context summarization

Example instance (with all fields)

A call stack summary

Any number of recursive calls

Top activation

Call to \texttt{fix} from \texttt{main}
Roadmap

- Background: Memory as graphs
- Abstracting calling contexts
- Deriving inductive cases for calling context summarization
Intuition

- At a call, new **activation record added**
- Need to **widen** to obtain summaries with **stack instances** (but need the definition of **stack**)
- Compare a few iterations to augment the **definition of stack**, then apply widening.
  - **Subtraction**
void main() {
    ... 
    walk(l);
}
void walk(list* x) {
    if (x != NULL)
        walk(x->next);
}
## Preliminary Experience

### Benchmark

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Recursive (ms)</th>
<th>Iterative (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>list traversal</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>list get nth</td>
<td>22</td>
<td>4</td>
</tr>
<tr>
<td>list insert nth</td>
<td>48</td>
<td>16</td>
</tr>
<tr>
<td>list remove nth</td>
<td>27</td>
<td>11</td>
</tr>
<tr>
<td>list deletion</td>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>list append</td>
<td>20</td>
<td>13</td>
</tr>
<tr>
<td>list reverse</td>
<td>29</td>
<td>5</td>
</tr>
</tbody>
</table>

### Case Study:

- Discussing precision with this approach versus the modular approach.

  [see paper]

  points per recursion (call site and return site)
Conclusion

• Xisa applied straightforwardly
  - folding at call sites
  - unfolding at return sites
  - widening applied on recursion
  - core analysis algorithms remain
  - evidence for flexibility of the framework

• New option for interprocedural analysis
  - “very” context-sensitive
  - no need to abstract relations ⇒ simpler base domains
http://www.cs.colorado.edu/~bec/
Programming Languages Research at the University of Colorado, Boulder

Amer Diwan  Jeremy Siek  Bor-Yuh Evan Chang  Sriram Sankaranarayanan
How do we effectively express computation?
language design, type systems, logic

How do we assist reasoning about programs?
program analysis, development tools

How do we make programs run efficiently?
performance analysis, compilation

How do we get reliable, secure software?
verification, model checking

PL research at CU has **breadth**!
PL researchers at CU collaborate!

- Language design
- Performance analysis
- Program analysis
- Verification

Gradual Programming
Dynamic Algorithmic Complexity
Preventing Resource Exhaustion Attacks

Bor-Yuh Evan Chang
University of Colorado at Boulder
Formal methods *connections*

Prof. **Aaron Bradley** (ECEE)
Prof. **Fabio Somenzi** (ECEE)
The PL group has fun together!

Group meetings at the Boulder Tea House once/twice a month

Successes: 2 papers at each of POPL’11, PLDI’10, and POPL’10

Travel to conferences (Todd at OOPSLA’09)
Our group

Devin
Weiyu
PhD
Sam
Jonathan
BS
Aleks

Huck
MS

Amer
Jeremy
Faculty

James
Evan
Sriram

Bor-Yuh Evan Chang 張博聿, University of Colorado at Boulder
Programming Languages Research at the University of Colorado, Boulder
Applying to Colorado

- Computer Science Department information
  http://www.cs.colorado.edu/grad/admission/

- Deadlines
  Jan 2 for Fall (Oct 1 for Spring)

- Graduate Advisor: Jackie DeBoard
  jacqueline.deboard@colorado.edu

- Talk to me about application fee waiver
  http://www.cs.colorado.edu/~bec/