Shape Analysis via 3-Valued Logic

Mooly Sagiv
Tel Aviv University

http://www.cs.tau.ac.il/~msagiv/toplas02.pdf

www.cs.tau.ac.il/~tvla

Plan

- "Realistic" applications
- Techniques for scaling
- Interprocedural Analysis
- Some research problems

Heap & Concurrency [Yahav POPL'01]

- Concurrency with the heap is evil…
- Java threads are just heap allocated objects
- Data and control are strongly related
  - Thread-scheduling info may require understanding of heap structure (e.g., scheduling queue)
  - Heap analysis requires information about thread scheduling

```
Thread t1 = new Thread();
Thread t2 = new Thread();
…
t = t1;
…
t.start();
```

[Michael&Scott PODC96]

```
public void enqueue(Object value) {
  node = new QueueItem()                  // allocate queue node
  node.val = value                 // copy enqueued value into node
   node.next.ref = NULL
     while(true) {                        // Keep trying until done
       tail = this.Tail             // get Tail.ptr and Tail.count
      next = tail.ref.next                // get next ptr and count
       if (tail == this.Tail) {           // are tails consistent?
         if (next.ref == NULL) {          // was tail pointing to last
   node?
           if CAS(tail.ref.next,
               next, <node, next.count+1>) { // try connect
            break                         // Enqueue is done. Exit loop
           }
        } else {                        // tail wasn't pointing to last
   node
          CAS(this.Tail, tail,<next.ref, tail.count+1>) // try advance
   tail
        }
     }
   }
 CAS(this.Tail, tail, <node, tail.count+1>) //enqueue done. try swing
   tail
}
```

Correctness

- P1 The linked list is always connected
- P2 Nodes are only inserted after the last node of the linked list
- P3 Nodes are only deleted from the beginning of the linked list
- P4 Head always points to the first node in the linked list
- P5 Tail always points to a node in the linked list

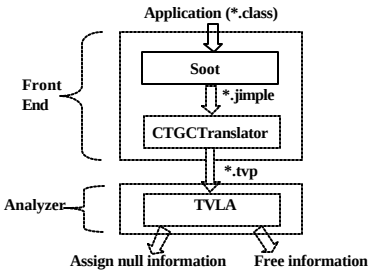
Examples Verified

Program	**Property**
twoLock Q	No interference No memory leaks Partial correctness
Producer/consumer	No interference No memory leaks
Apprentice Challenge	Counter increasing
Dining philosophers with resource ordering	Absence of deadlock
Mutex	Mutual exclusion
Web Server	No interference

Compile-Time GC for Java (Ran Shaham, SAS'03, SCP)

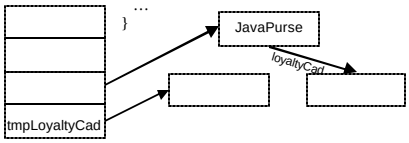
- The compiler can issue free when objects are no longer needed
- Analysis of Java/JavaCard programs
- Requires forward information
- Maintained via history automata
  - Provides instrumentation predicates
- More automatic analysis (G. Arnold)

CTGC architecture



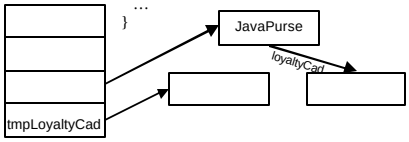
Usage of CTGC output (1)

```
private void expandLoyaltyProgramIfNeeded() {
   currLoyatyCount++;
   if (currLoyaltyCount > loyaltyCount.length) {
     tmpLoyaltyCad = new short[loyaltyCount.length * 2];
     // The array is currently copied using a for loop
     Util.arrayCopyNonAtomic(loyaltyCad, 0, tmpLoyatyCad, …);
     // loyaltyCad could be freed here
     loyaltyCard = tmpLoyatyCad
   }
  // similar code for expanding loyaltySIO array
  …
}
```



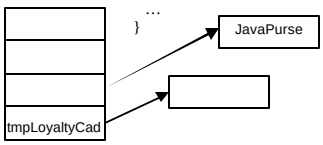
Usage of CTGC output (1)

```
private void expandLoyaltyProgramIfNeeded() {
   currLoyatyCount++;
   if (currLoyaltyCount > loyaltyCount.length) {
     tmpLoyaltyCad = new short[loyaltyCount.length * 2];
     // The array is currently copied using a for loop
     Util.arrayCopyNonAtomic(loyaltyCad, 0, tmpLoyatyCad, …);
   // loyaltyCad could be freed here
     loyaltyCard = tmpLoyatyCad
   }
  // similar code for expanding loyaltySIO array
  …
}
```



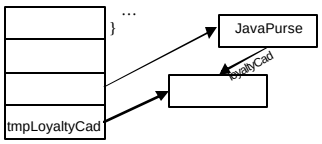
Usage of CTGC output (1)

```
private void expandLoyaltyProgramIfNeeded() {
   currLoyatyCount++;
   if (currLoyaltyCount > loyaltyCount.length) {
     tmpLoyaltyCad = new short[loyaltyCount.length * 2];
     // The array is currently copied using a for loop
     Util.arrayCopyNonAtomic(loyaltyCad, 0, tmpLoyatyCad, …)
   // loyaltyCad could be freed here
     loyaltyCad = tmpLoyatyCad
   }
  …
}
```



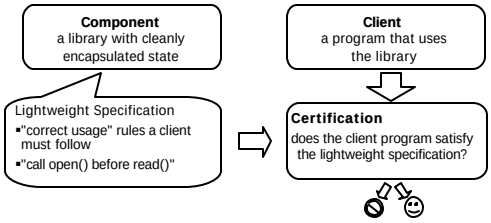
Usage of CTGC output (1)

```
private void expandLoyaltyProgramIfNeeded() {
   currLoyatyCount++;
   if (currLoyaltyCount > loyaltyCount.length) {
     tmpLoyaltyCad = new short[loyaltyCount.length * 2];
     // The array is currently copied using a for loop
     Util.arrayCopyNonAtomic(loyaltyCad, 0, tmpLoyatyCad, …);
   // loyaltyCad could be freed here
     loyaltyCad = tmpLoyatyCad
   }
  …
}
```



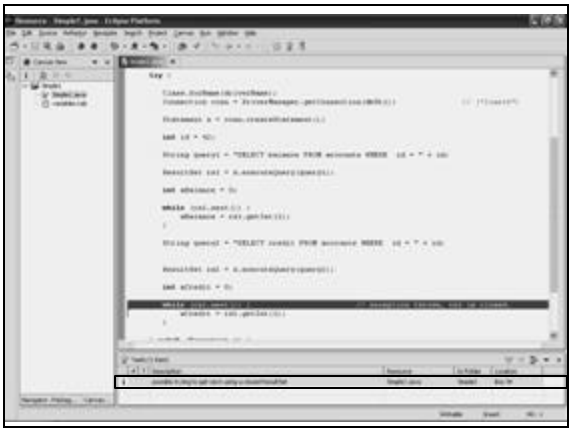
Verification of Safety Properties (PLDI'02, 04)

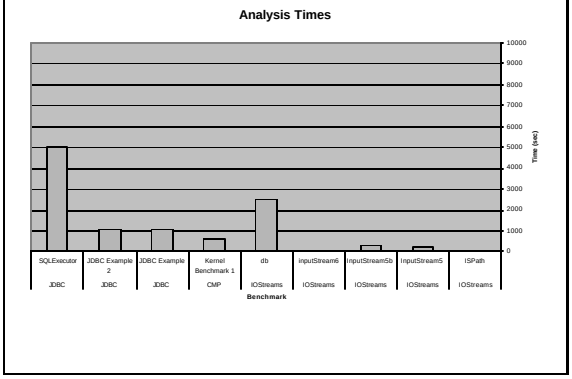
The *Canvas* Project (with IBM Watson)
(***C***omponent ***A***nnotation, ***V***erification ***a***nd ***S***tuff)

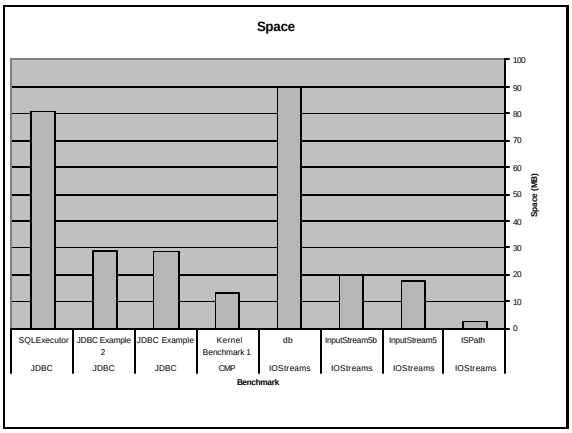


Prototype Implementation

- Applied to several example programs
  - Up to 5000 lines of Java
- Used to verify
  - Absence of concurrent modification exception
  - JDBC API conformance
  - IOStreams API conformance







Scaling

- Staged analysis
- Represent 3-valued structures with BDDs [Manevich SAS'02]
- Reduce static costs
- Controlled complexity
  - More coarse abstractions [Manevich SAS'04]
  - Counter example based refinement
- Assume/Guarantee Reasoning
  - Use procedure specifications [Yorsh, TACAS'04]
  - Decision procedures for linked data structures [Immerman, CAV'04, Lev-Ami, CADE '05]
- Exploit "good" program properties
  - Encapsulation & Data abstraction
- Handle procedures

Why is Heap Analysis Difficult?

- Destructive updating through pointers
  - `p® next = q`
  - Produces complicated aliasing relationships
  - Track aliasing on 3-valued structures
- Dynamic storage allocation
  - No bound on the size of run-time data structures
  - Canonical abstraction ⇒ finite-sized 3-valued structures
- Data-structure invariants typically only hold at the beginning and end of operations
  - Need to verify that data-structure invariants are re-established
  - Query the 3-valued structures that arise at the exit

Summary

- Canonical abstraction is powerful
  - Intuitive
  - Adapts to the property of interest
- Used to verify interesting program properties
  - Very few false alarms
- But scaling is an issue

Summary

- Effective Abstract Interpretation
  - Always terminates
  - Precise enough
  - But still expensive
- Can model
  - Heap
  - Unbounded arrays
  - Concurrency
- More instrumentation can mean more efficient
- But canonical abstraction is limited
  - Correlation between list lengths
  - Arithmetic
  - Partial heaps

Interprocedural Analysis

Noam Rinetzky

www.cs.tau.ac.il/~maon

How to handle procedures?

- Pure functions
  - Procedure ^{o} input/output relation
  - No side-effects

```
main() {
  int w=0,x=0,y=0,z=0;
  w = inc(y);
  x = inc(z);
  assert: w+x is even
}
```

p	ret
0	1
1	2
2	3
..	…

```
int inc(int p) {
  return 2 + p - 1;
}
```

How to handle procedures?

- Pure functions
  - Procedure ^{o} input/output relation
  - No side-effects

```
main() {
  int w=0,x=0,y=0,z=0;
  w = inc(y);
  x = inc(z);
  assert: w+x is even
}
```

w	x	y	z
E	E	E	E
O	E	E	E
O	O	E	E

p	ret
Even	Odd
Odd	Even

```
int inc(int p) {
  return 2 + p - 1;
}
```

What about global variables?

- Procedures have side-effects
- Easy fix

p	g	ret	g'
0	0	1	0
…	…	…	…

p	g	ret	g'
Even	E/O	Odd	Even
Odd	E/O	Even	Odd

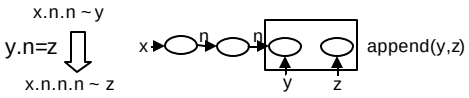
```
int g = 0;
main() {
  int w=0,x=0,y=0,z=0;
  w = inc(y);
  x = inc(z);
  assert: w+x+g is even
}
```

```
int inc(int p) {
  g = p;
  return 2 + p - 1;
}
```

But what about pointers and heap?

Pointers
- Aliasing
- Destructive update

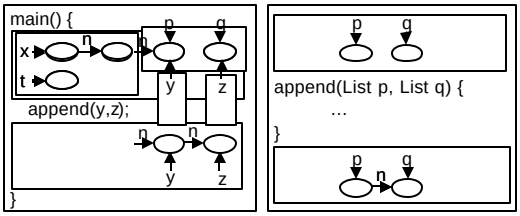
Heap
- Global resource
- Anonymous objects



How to tabulate append?

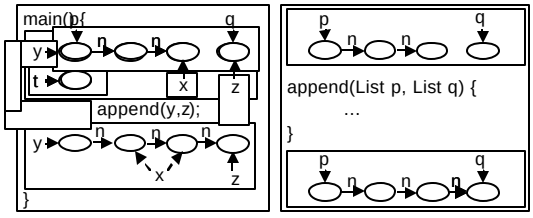
How to tabulate procedures?

- Procedure ^{o} input/output relation
  - Not reachable ➔ Not effected
  - proc: local (≡reachable) heap ➔ local heap



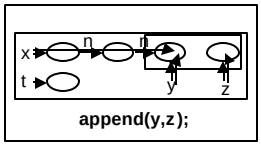
How to handle sharing?

- External sharing may break the functional view

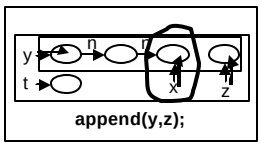


What's the difference?

1st Example



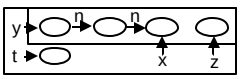
2nd Example



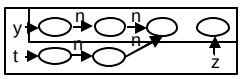
Cutpoints

- An object is a **cutpoint** for an invocation
  - Reachable from actual parameters
  - Not pointed to by an actual parameter
  - Reachable without going through a parameter

append(y,z)

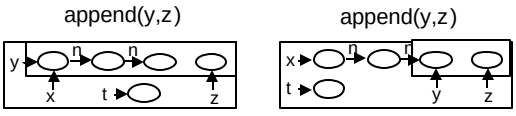


append(y,z)



Cutpoint freedom

- **Cutpoint-free**
  - Invocation: has no cutpoints
  - Execution: every invocation is cutpoint-free
  - Program: every execution is cutpoint-free



Main Results(POPL'05)

- Concrete operational semantics
  - Sequential programs
  - Local heap
  - Track cutpoints
  - Storeless
    - good for shape abstractions
  - Observational equivalent with "standard" global store-based heap semantics
    - Java and "clean" C
- Abstractions
  - Shape Analysis
    - Example: singly-linked lists
  - May-alias [Deutsch, PLDI 04]

Main results(SAS '05)

- Cutpoint freedom
- Non-standard concrete semantics
  - Verifies that an execution is cutpoint-free
  - Local heaps
- Interprocedural shape analysis
  - Conservatively verifies
    - program is cutpoint free
    - Desired properties
      - Partial correctness of quicksort
  - Procedure summaries
- Prototype implementation

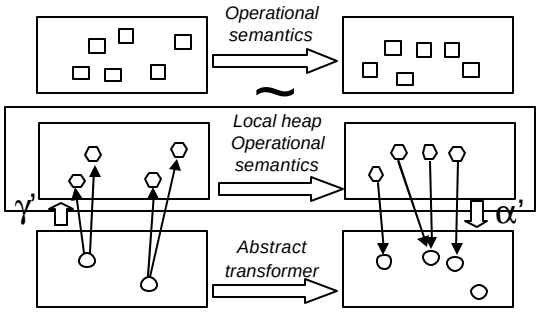
Plan

- ✓Cutpoint freedom
- Non-standard concrete semantics
- Interprocedural shape analysis
- Prototype implementation

Programming model

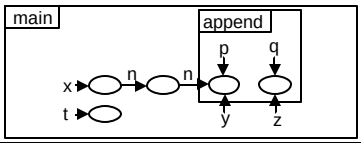
- Single threaded
- Procedures
  - ✓Value parameters
    - Formal parameters not modified
  - ✓Recursion
- Heap
  - ✓Recursive data structures
  - ✓Destructive update
  - × No explicit addressing (&)
  - × No pointer arithmetic

Introducing local heap semantics



Memory states

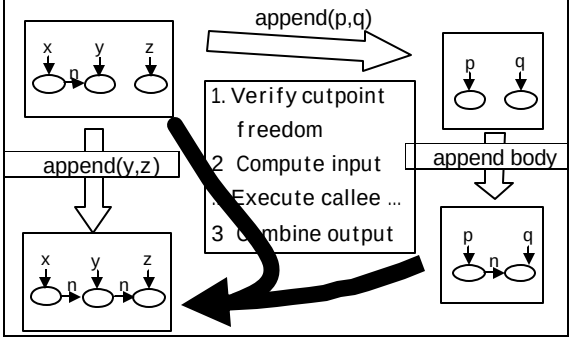
- A memory state encodes a **local heap**
  - Local variables of the **current procedure invocation**
  - Relevant part of the heap
    - Relevant ≡ Reachable



Abstract semantics

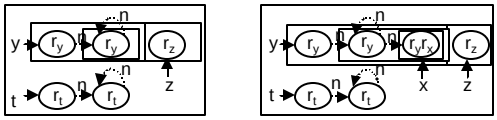
- Conservatively apply statements using 3-valued logic (with the non-standard semantics)
  - Use canonical abstraction
  - Reinterpret FO formulas using Kleene value

Procedure calls



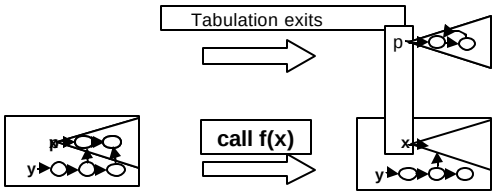
Conservative verification of cutpoint-freedom

- Invoking append(y,z) in main
  - $R_{\{y,z\}}(v) \Leftrightarrow \exists v_1 : y(v_1) \wedge n^*(v_1,v) \vee \exists v_1 : z(v_1) \wedge n^*(v_1,v)$
  - $isCP_{main,\{y,z\}}(v) = R_{\{y,z\}}(v) \wedge (\neg y(v) \wedge \neg z(v)) \wedge (x(v) \vee t(v) \vee \exists v_1 : \neg R_{\{y,z\}}(v_1) \wedge n(v_1,v))$

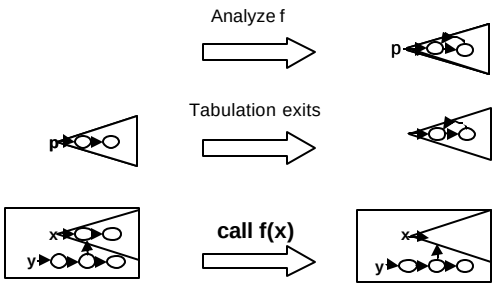


Cutpoint free **Not Cutpoint free**

Interprocedural shape analysis

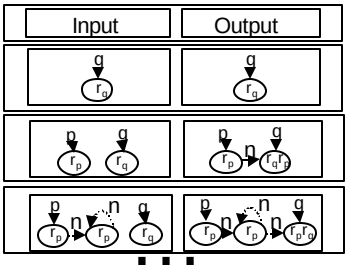


Interprocedural shape analysis



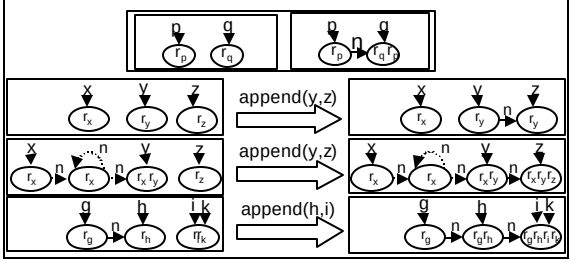
Interprocedural shape analysis

- Procedure ≡ input/output relation



Interprocedural shape analysis

- Reusable procedure summaries
  - Heap modularity



Plan

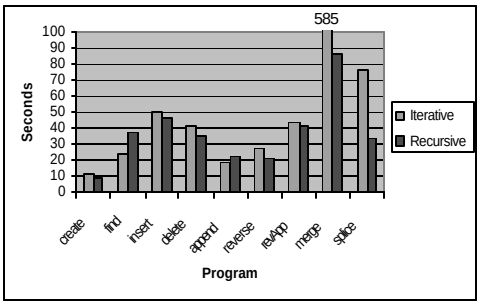
- ✓ Cutpoint freedom
- ✓ Non-standard concrete semantics
- ✓ Interprocedural shape analysis
- Prototype implementation

Prototype implementation

- TVLA based analyzer
- Soot-based Java front-end
- Parametric abstraction

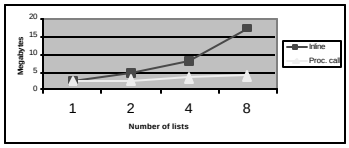
Data structure	Verified properties
Singly linked list	Cleanness, acyclicity
Sorting (of SLL)	+ Sortedness
Unshared binary trees	Cleaness, tree-ness

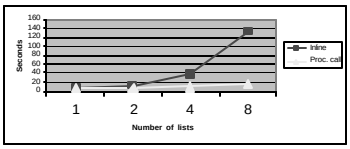
Iterative vs. Recursive (SLL)



Inline vs. Procedural abstraction

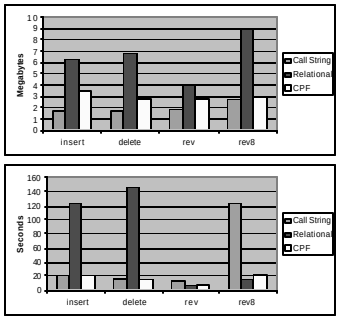
```
// Allocates a list of
// length 3
List create3(){
   …
}

main() {
  List x1 = create3();
  List x2 = create3();
  List x3 = create3();
  List x4 = create3();
  …
}
```





Call string vs. Relational vs. CPF

[Rinetzky and Sagiv, CC'01] [Jeannet et al., SAS'04]



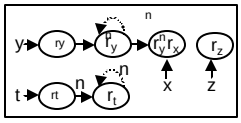
Related Work

- **Interprocedural shape analysis**
  - Rinetzky and Sagiv, CC '01
  - Chong and Rugina, SAS '03
  - Jeannet et al., SAS '04
  - Hackett and Rugina, POPL'05
  - Rinetzky et al., POPL'05
- **Local Reasoning**
  - Ishtiaq and O'Hearn, POPL'01
  - Reynolds, LICS '02
- **Encapsulation**
  - Noble et al. IWACO '03
  - ...

Future work

- Bounded number of cutpoints
- False cutpoints
  - Liveness analysis

```
append(y,z);
x = null;
```



Summary

- Cutpoint freedom
- Non-standard operational semantics
- Interprocedural shape analysis
  - Partial correctness of quicksort
- Prototype implementation

(Some) Research Problems

- A model checker for heap allocated data structures
- Different heap abstractions
- Random interpretation for heap
- Bounded model checking for heap
- Hoare style verification for heap

Mange Tak