TVLA: A system for inferring Quantified Invariants

Tal Lev-Ami
Tom Reps
Mooly Sagiv
Reinhard Wilhelm
Greta Yorsh
Example: Mark and Sweep

void Mark(Node root) {
    if (root != NULL) {
        pending = ∅
        pending = pending ⋃ {root}
        marked = ∅
        while (pending ≠ ∅) {
            x = SelectAndRemove(pending)
            marked = marked ⋃ {x}
            t = x → left
            if (t ≠ NULL)
                if (t ∉ marked)
                    pending = pending ⋃ {t}
            t = x → right
            if (t ≠ NULL)
                if (t ∉ marked)
                    pending = pending ⋃ {t}
        }
    }
    assert(marked = Reachset(root))
}

∀v: marked(v)⇔∃r: root(r) ∧ reach(w, v)

void Sweep() {
    unexplored = Universe
    collected = ∅
    while (unexplored ≠ ∅) {
        x = SelectAndRemove(unexplored)
        if (x ∉ marked)
            collected = collected ⋃ {x}
    }
    assert(collected = Universe − Reachset(root))
}
Example: Mark

```c
void Mark(Node root) {
    if (root != NULL) {
        pending = ∅
        pending = pending ∪ {root}
        marked = ∅
        while (pending ≠ ∅) {
            x = SelectAndRemove(pending)
            marked = marked ∪ {x}
            t = x → left
            if (t ≠ NULL)
                if (t ∉ marked)
                    pending = pending ∪ {t}
            t = x → right
            if (t ≠ NULL)
                if (t ∉ marked)
                    pending = pending ∪ {t}
        }
    }
}
```

∀v: marked(v) ⇔ ∃r: root(r) ∧ reach(r, v)
Example: Mark

```c
void Mark(Node root) {
    if (root != NULL) {
        pending = \emptyset
        pending = pending \cup \{root\}
        marked = \emptyset
        while (pending \neq \emptyset) {
            x = SelectAndRemove(pending)
            marked = marked \cup \{x\}
            t = x \rightarrow left
            if (t \neq NULL)
                if (t \not\in marked)
                    pending = pending \cup \{t\}
            /* t = x \rightarrow right
            * if (t \neq NULL)
            * if (t \not\in marked)
            *    pending = pending \cup \{t\}
            */
        }
    }
    assert(marked = Reachset(root))
}
```
Example: Mark

```c
void Mark(Node root) {
    if (root != NULL) {
        pending = ∅
        pending = pending ∪ {root}
        marked = ∅
        while (pending ≠ ∅) {
            x = SelectAndRemove(pending)
            marked = marked ∪ {x}
            t = x → left
            if (t ≠ NULL)
                if (t ∉ marked)
                    pending = pending ∪ {t}
        }
    }
    assert(marked = Reachset(root))
}
```
A Singleton Buffer

Boolean empty = true;
Object b = null;

produce() {
1: Object p = new();
2: await (empty) then {
    b = p;
    empty = false;
}
3:
}

consume() {
Object c;
4: await (!empty) then {
    c = b;
    empty = true;
}
5: use(c);
6: dispose(c);
7:
}
∀t, e, v: t ≠ e ∧ c(t, v) ∧ c(e, w) → v ≠ w

Boolean empty = true;
Object b = null;

produce() {
    1: Object p = new();
    2: await (empty) then {
        b = p;
        empty = false;
    }
    3: 
}

consume() {
    Object c;
    4: await (!empty) then {
        c = b;
        empty = true;
    }
    5: use(c);
    6: dispose(c);
    7: 
}
Quantified Invariants are hard

- Corner cases
- Sizes
- Nested loops
- Code updates
- First order reasoning is hard
Our approach

• Automatically infer sound invariants
• Library (PL) designers can define families of interesting invariants
• Limited form of invariants
• Sound but incomplete reasoning
  – Abstract interpretation over quantified invariants
  – Parameterized by the
Applications

• Memory safety & preservation of data structure invariants [Dor, SAS’00, Loginov, ISSTA’08]
• Compile-time garbage collection [Shaham, SAS’03]
• Correct API usage [Ramalingam, PLDI’02, Yahav. PLDI’04]
• Typestate verification [Yahav, ISSTA’06]
• Partial & total correctness
  – Sorting implementations [Lev-Ami, ISTTA’00, Rinetzky, SAS’05]
  – Deutsch-Shorr-Waite [Loginov, SAS’06]
• Thread modular shape analysis [Gotsman, PLDI’07]
• Linearizability [Amit, CAV’07, Berdine, CAV’08]
Example: Concrete Interpretation

\[ x = \text{NULL} \]

\[ \text{F} \rightarrow t = \text{malloc(..)}; \]
\[ \text{T} \rightarrow t \rightarrow \text{next}=x; \]
\[ x = t \]
\[ \text{return } x \]

\[ \text{empty} \]
\[ t \rightarrow x \]
\[ t \rightarrow n \rightarrow x \]
\[ t \rightarrow n \rightarrow n \rightarrow x \]

\[ t \rightarrow x \]
\[ t \rightarrow n \rightarrow x \]
\[ t \rightarrow n \rightarrow n \rightarrow x \]

\[ t \rightarrow x \]
\[ t \rightarrow n \rightarrow x \]
\[ t \rightarrow n \rightarrow n \rightarrow x \]
Shape Analysis

x = NULL

F

t = malloc(..);

T

t→next=x;

x = t

return x
Outline

• Canonical Abstraction [TOPLAS’02]
• Quantified Invariants
• Operating on Abstractions
Abstract Interpretation
[Cousot & Cousot]

- Checking interesting program properties is undecidable
- Use abstractions
- Every verified property holds (sound)
- But may fail to prove properties which always hold (incomplete)
  - false alarms
- Minimal false alarms
Simplified Abstract Interpretation

Abstract domain (bounded)

Concrete domain (unbounded)

Simplified Abstract Interpretation
Most Precise Abstract Transformer
[Cousot, Cousot POPL 1979]
An example predicate abstraction

\[ x = 3 \]

\[
\text{while (true) } \{ \\
\quad x = x + 1 ; \\
\}
\]

predicates

\[ \{ x > 0 \} \]
An example predicate abstraction

x = 3

while (true) {
    x = x + 1 ;
}

predicates

{x > 0}
An example predicate abstraction

predicates

\{ x > 0 \}

\{ 0, 1 \}

x = 3

\{ 1 \}

while (true) {
  \{ 1 \}
  \{ 1 \}
  x = x + 1 ;
  \{ 1 \}
}

\{ 1 \}
Shape Analysis as Abstract Interpretation

- Represent concrete stores as labeled directed graphs
  - 2-valued structures \{0, 1\}
  - Abstract away
    - Concrete locations
    - Primitive values
  - But unbounded
- Represent abstract stores as labeled directed graphs
  - 3-valued structures \{0, 1, \frac{1}{2}\}
  - Several concrete nodes are represented by a summary node
  - Abstract away field correlations
Representing Concrete Stores by Logical Structures

- Parametric vocabulary
- Heap
  - Locations ≈ Individuals
  - Program variables ≈ Unary relations
  - Fields ≈ Binary relations
Representing Stores as Logical Structures
(Example)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>u1</th>
<th>u2</th>
<th>u3</th>
<th>u4</th>
<th>u5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>p</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>u2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>u3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>u4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>u5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

x → u1 → u2 → u3 → u4 → u5
Representing Abstract Stores by 3-Valued Logical Structures

• A join semi-lattice: $0 \sqcup 1 = 1/2$
• $\{0, 1, \frac{1}{2}\}$ values for relations
Canonical Abstraction ($\beta$)

- Partition the individuals into **equivalence classes** based on the values of their unary relations
  - Every individual is mapped into its equivalence class
- Collapse binary relations via $\sqcap$
  - $p^S(u_1', u_2') = \sqcap \{ p^B(u_1, u_2) \mid f(u_1) = u_1', f(u_2) = u_2' \}$
- At most $2^A$ abstract individuals
x = NULL;
while (...) do {
    t = malloc();
    t \rightarrow next = x;
    x = t
}
x = NULL;
while (…) do {
    t = malloc();
    t \rightarrow \text{next} = x;
    x = t
}

Canonical Abstraction
Canonical Abstraction and Equality

• **Summary nodes** may represent more than one element
• (In)equality need not be preserved under abstraction
• Explicitly record equality
• Summary nodes are nodes with $\text{eq}(u, u)=1/2$
x = NULL;
while (…) do {
    t = malloc();
    t -> next = x;
    x = t
}
x = NULL;
while (…) do {
    t = malloc();
    t \rightarrow next = x;
    x = t
}

Canonical Abstraction
Canonical Abstraction

• Partition the individuals into \textit{equivalence classes} based on the values of their unary relations
  – Every individual is mapped into its equivalence class

• Collapse relations via $\sqsubseteq$

  $p^S(u'_1, ..., u'_k) = \sqsubseteq \{ p^B(u_1, ..., u_k) \mid f(u_1)=u'_1, ..., f(u'_k)=u_k \}$

• At most $2^A$ abstract individuals
x = NULL;
while (…) do {
    t = malloc();
    t →next=x;
    x = t
}
Limitations

• Information on summary nodes is lost
Increasing Precision

• Global invariants
  – User-supplied, or consequence of the semantics of the programming language

• Record extra information in the concrete interpretation
  – Tunes the abstraction
  – Refines the concretization

• Naturally expressed in FO$^{TC}$
Heap Sharing relation

\( is(v) = \exists v_1, v_2 : n(v_1, v) \land n(v_2, v) \land v_1 \neq v_2 \)
Heap Sharing relation

\[ is(v) = \exists v_1, v_2: n(v_1, v) \land n(v_2, v) \land v_1 \neq v_2 \]
Reachability relation

\[ t[n](v1, v2) = n^*(v1,v2) \]
List Segments

\[ u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow u_4 \rightarrow u_5 \rightarrow u_6 \rightarrow u_7 \rightarrow u_8 \]

\[ u_1 \quad u_2,3,4,6,7,8 \quad u_5 \]

\[ x \rightarrow y \]
Reachability from a variable

- $r[n,y](v) = \exists w: y(w) \land n^*(w, v)$
Additional Instrumentation relations

- $\text{inOrder}(v) = \forall w: n(v, w) \rightarrow \text{data}(v) \leq \text{data}(w)$
- $\text{c}_{fb}(v) = \forall w: f(v, w) \rightarrow b(w, v)$
- $\text{tree}(v)$
- $\text{dag}(v)$
- Weakest Precondition
  [Ramalingam, PLDI’02]
- Learned via Inductive Logic Programming
  [Loginov, CAV’05]
- Counterexample guided refinement
Most Precise Abstract Transformer
[Cousot, Cousot POPL 1979]
Partial Concretization
Partial Concretization
Best Transformer \((x = x \rightarrow n)\)
Partial Concretization based Transformer \((x = x \rightarrow n)\)

Abstract

Semantics

canonical abstraction
Partial Concretization

- Employed in other shape analysis algorithms [Distefano, TACAS’06, Evan, SAS’07, POPL ’08]
- Soundness is immediate
- Can even guarantee precision under certain conditions [Lev-Ami, VMCAI’07]
- Locally refine the abstract domain per statement
Non-Fixed-Partition

$\begin{align*}
x & = x \rightarrow n \\
y & \rightarrow y \rightarrow x \rightarrow y
\end{align*}$
Shape Analysis

\[ x = \text{NULL} \]

\[ t = \text{malloc}(..); \]

\[ t \rightarrow \text{next}=x; \]

\[ x = t \]

\[ \text{return } x \]
[TOPLAS’02, Lev-Ami, SAS’00]

• Concrete transformers using first order formulas
• Effective algorithms for computing transformers
  – Partial concretization
  – 3-valued logic Kleene evaluation
  – Finite differencing & incremental algorithms
    [Reps, ESOP’03]
• A parametric yacc like system[TVLA]
  – http://www.cs.tau.ac.il/~tvla
Applications
Proving Correctness of Sorting Implementations (Lev-Ami, Reps, S, Wilhelm ISSTA 2000)

• Partial correctness
  – The elements are sorted
  – The list is a permutation of the original list

• Termination
  – At every loop iterations the set of elements reachable from the head is decreased
Sortedness
Example: Sortedness

\[ \text{inOrder}(v) = \forall v_1: n(v, v_1) \rightarrow \text{dle}(v, v_1) \]
Example: InsertSort

typedef struct list_cell {
    int data;
    struct list_cell *n;
} *List;

List InsertSort(List x) {
    List r, pr, rn, l, pl; r = x; pr = NULL;
    while (r != NULL) {
        l = x; rn = r->n; pl = NULL;
        while (l != r) {
            if (l->data > r->data) {
                pr->n = rn; r->n = l;
                if (pl == NULL) x = r;
                else pl->n = r;
                r = pr;
                break;
            }
            pl = l; l = l->n;
        }
        pr = r; r = rn;
    }
    return x;
}
Example: InsertSort

typedef struct list_cell {
    int data;
    struct list_cell *n;
} *List;

List InsertSort(List x) {
    if (x == NULL) return NULL
    pr = x; r = x->n;
    while (r != NULL) {
        pl = x; rn = r->n; l = x->n;
        while (l != r) {
            pr->n = rn;
            r->n = l;
            pl->n = r;
            r = pr;
            break;
        }
        pl = l;
        l = l->n;
    }
    pr = r;
    r = rn;
}
Verification of Safety Properties (PLDI’02, 04)
The Canvas Project (with IBM Watson)
(Component Annotation, Verification and Stuff)

**Component**
a library with cleanly encapsulated state

**Client**
a program that uses the library

**Lightweight Specification**
- "correct usage" rules a client must follow
- "call open() before read()"

**Certification**
does the client program satisfy the lightweight specification?
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
Concurrency

- Models threads as ordinary objects [Yahav, POPL’01]
- Thread-modular shape analysis [Gotsman, PLDI’07]
- Heap decomposition [Manevich, SAS’08]
- Thread quantification [Berdine, CAV’08]
- Enforcing a locking regime [Rinetzkey]
Correctness of concurrent ADT implementations

[Amit CAV’07, Berdine, CAV’08]

• Small pointer manipulation programs
• Fine grained concurrency
• Benign data races
• Error prone
• Interesting properties
  – Memory safety
  – Partial correctness
  – Linearizability
  – Liveness
Treiber's Non-blocking Stack

[1] void push(Stack *S, data_type v) {
[2]     Node *x = alloc(sizeof(Node));
[3]     x->d = v;
[4]     do {
[5]         Node *t = S->Top;
[6]         x->n = t;
[7]     } while (!CAS(&S->Top,t,x));
[8] }

[9] data_type pop(Stack *S){
[10] do {
[12]     if (t == NULL)
[13]         return EMPTY;
[14]     Node *s = t->n;
[15]     data_type r = s->d;
[16]     } while (!CAS(&S->Top,t,s));
[17]     return r;
[18] }
Experimental results

<table>
<thead>
<tr>
<th>Verified Programs</th>
<th>#states</th>
<th>time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Treiber’s stack [1986]</td>
<td>764</td>
<td>7</td>
</tr>
<tr>
<td>Two-lock queue [Michael &amp; Scott, PODC’96]</td>
<td>3,415</td>
<td>17</td>
</tr>
<tr>
<td>Non-blocking queue</td>
<td>10,333</td>
<td>252</td>
</tr>
</tbody>
</table>

- First automatic verification of linearizability for unbounded number of threads
Handling Larger Programs

- Staged analysis
- Handling procedures
- Specialized abstractions
  - Counterexample guided refinement [McMillan, POPL’08]
- Coercer abstractions
  - Weaker summary nodes [Arnold, SAS’06]
  - Special join operator [Manevich, SAS’04, TACAS’07, SAS’08, Yang’08]
  - Heterogeneous abstractions [Yahav, PLDI’04]
- Implementation techniques
  - Optimizing transformers [Bogodlov, CAV’07]
  - Optimizing GC
  - Reducing static size
  - Partial evaluation
  - Persistent data structures [Manevich, SAS’04]
  - …
Handling Procedures

• Complicated sharing patterns [Rinetzky, CC’01]
• Relational shape analysis [Jeannet, SAS’04]
• New semantics for procedures (Cutpoints) [Rinetzky, POPL’05]
• Tabulation for cutpoint free programs [Rinetzky, SAS’05]
• Handling cutpoints [Gotsman, SAS’06]
• Modularity [Rinezky, ESOP’07]
How to tabulate procedures?

- Procedure $\equiv$ input/output relation
  - Not reachable $\Rightarrow$ Not effected
  - proc: local ($\equiv$reachable) heap $\Rightarrow$ local heap

```c
main() {
    append(y,z);
    append(List p, List q) {
        ...
    }
}
```
How to handle sharing?

- External sharing may break the functional view
What’s the difference?

1\textsuperscript{st} Example

```
append(y,z);
```

2\textsuperscript{nd} Example

```
append(y,z);
```

*x* → \( n \) → \( n \) → \( y \) → \( z \)

*y* → \( n \) → \( n \) → \( x \) → \( z \)
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
Introducing local heap semantics

Operational semantics

Local heap Operational semantics

Abstract transformer

\[ \gamma' \sim \alpha' \]
Interprocedural shape analysis

• Procedure ≡ input/output relation
Interprocedural shape analysis

• Reusable procedure summaries
  – Heap modularity
Summary

• Canonical abstraction is powerful
  – Intuitive
  – Adapts to the property of interest
  – More instrumentation may mean more efficient

• Used to verify interesting program properties
  – Very few false alarms

• But scaling is an issue