Static Analysis Projects

- Implement static analysis for a toy set language inspired by Python
- Implement static analysis on top of LLVM
  - Flow sensitive points-to for programs w/o procedures
- Implement static analysis on top of SOOT
  - Flow sensitive points-to for programs w/o procedures
- Implement k-CNF based extension on points to analysis for a toy language
  - \( p = \& q \)
More Suggested Static Analysis Projects

◆ Information Flow
  – Partition the data structures into two classes
    » High and Low
  – Show that information from high data structure cannot leak to X

◆ Taint checking for Java
Principles of Shape Analysis

Mooly Sagiv
Thomas Reps
Reinhard Wilhelm
and also

- Universität des Saarlandes
  - J. Bauer, Kreiker,
  - R. Biber
  - S. Parduhn
  - R. Seidel
  - J. Reineke
- University of Wisconsin
  - F. DiMaio
  - D. Gopan
  - A. Loginov
- IBM Research
  - J. Field
  - H. Kolodner
  - M. Rodeh
- Inria
  - B. Jeannet
- Berkeley
  - B. McCloskey
- Tel-Aviv University
  - D. Amit
  - I. Bogudlov
  - G. Arnold
  - G. Erez
  - N. Dor
  - T. Lev-Ami
  - R. Manevich
  - R. Shaham
  - A. Rabinovich
  - N. Rinetzky
  - E. Yahav
  - G. Yorsh
  - A. Warshavsky
- Microsoft Research
  - J. Berdine
  - B. Cook
  - G. Ramalingam
Original Problem: **Shape Analysis**
(Jones and Muchnick 1981)

- Characterize dynamically allocated data
  - \(x\) points to an acyclic list, cyclic list, tree, dag, etc.
  - show that data-structure invariants hold
- Identify may-alias relationships
- Establish “disjointedness” properties
  - \(x\) and \(y\) point to structures that do not share cells
- Memory Safety
  - No null and dangling de-references
  - No memory leaks
- In OO programming
  - Everything is in the heap \(\Rightarrow\) requires shape analysis
int *p, *q;
q = (int *) malloc();
p = q;
l1: *p = 5;
    p = (int *) malloc();
l2: printf(*q);  /* printf(5) */
Example: Concrete Interpretation

```
x = NULL

F  T

\text{\texttt{t = malloc(\ldots);}}
\text{\texttt{t \rightarrow next = x;}}
\text{\texttt{x = t}}
\text{\texttt{return x}}
```

```
\text{empty}
\text{\texttt{t \rightarrow x}}
\text{\texttt{t \rightarrow n \rightarrow x}}
\text{\texttt{t \rightarrow n \rightarrow n \rightarrow x}}
\text{\ldots}
```

```
\text{\texttt{t \rightarrow o}}
\text{\texttt{t \rightarrow n \rightarrow o}}
\text{\texttt{t \rightarrow n \rightarrow n \rightarrow o}}
\text{\ldots}
```
Example: Abstract Interpretation

\[ x = \text{NULL} \]

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

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

\[ x = t \]

\[ \text{return } x \]

\[ \text{empty} \]

\[ t \rightarrow x \]

\[ t \rightarrow n \]

\[ t \rightarrow n \rightarrow n \]
List reverse(Element *head)
{
    List rev, ne;
    rev = NULL;
    while (head != NULL) {
        ne = head -> next;
        head -> next = rev;
        head = ne;
        head = ne;
    }
    return rev;
}
Memory Leakage

Element* reverse(Element *head)
{
    Element *rev, *ne;
    rev = NULL;
    while (head != NULL) {
        ne = head -> next;
        head -> next = rev;
        rev = head;
        head = ne;
    }
    return rev;
}
No memory leaks
Directed Reachability

• Directed reachability suffice to describe many properties of data structures
  – Absence of garbage
    • \( \forall x: r^*(\text{root}, x) \)
  – Acyclicity
    • \( \forall x: x \neq \text{root} \Rightarrow \neg r^*(\text{root}, x) \)
  – Data Structure Invariants
    • \( \forall x: f^*(\text{root}, x) \Leftrightarrow b^*(\text{root}, x) \)

\( r^*(x, y) \) denotes a finite directed path of relation of \( r \) from \( x \) to \( y \)
rotate(List first, List last) {
    if (first != NULL) {
        last -> next = first;
        first = first -> next;
        last = last -> next;
        last -> next = NULL;
    }
    assert acyclic first;
}
Logical Structures (Labeled Graphs)

- Nullary relation symbols
- Unary relation symbols
- Binary relation symbols
- \( \mathsf{FO}^{\mathsf{TC}} \) over \( \mathsf{TC} \), \( \forall \exists \neg \land \lor \) express logical structure properties
- Logical Structures provide meaning for relations
  - A set of individuals (nodes) \( U \)
  - Interpretation of relation symbols in \( P \)
    - \( p^0() \rightarrow \{0,1\} \)
    - \( p^1(v) \rightarrow \{0,1\} \)
    - \( p^2(u,v) \rightarrow \{0,1\} \)
Representing Stores as Logical Structures

- Locations \(\approx\) Individuals
- Program variables \(\approx\) Unary relations
- Fields \(\approx\) Binary relations
- Example
  - \(U = \{u_1, u_2, u_3, u_4, u_5\}\)
  - \(x = \{u_1\}\), \(p = \{u_3\}\) \(n = \{<u_1, u_2>, <u_2, u_3>, <u_3, u_4>, <u_4, u_5>\}\)
Interesting Properties

rotate(List first, List last) {
    if (first != NULL) {
        last -> next = first;
        first = first -> next;
        last = last -> next;
        last -> next = NULL;
    }
}

✓ No null-de references
✓ No memory leaks
✓ Returns an acyclic linked list
✓ Partially correct
Reasoning about Directed Reachability is hard

- Not first order expressible
- Not recursively enumerable
- Not clear if can be updated in first order logic
- What about modularity
  - Do we need to reason about the calling context?
Incremental Reachability
Small (local) updates

\[ n \xrightarrow{x \mapsto n := \text{NULL}} n' \]

\[ n^* \xrightarrow{\text{Weak logic}} n'^* \]

\[ \text{FO}^{TC} \]

\[ \text{FO}^{TC} \]
Adding an edge
\[ c \rightarrow n = d \]

\[ n^*(\alpha, \beta) \leftrightarrow n^*(\alpha, \beta) \lor (n^*(\alpha, c) \land n^*(d, \beta)) \]
Updating Directed Reachability in General Graph is Hard.
Removing an edge (destructive update)

\[
c \rightarrow n = \text{NULL}
\]

\[
n^*(\alpha, \beta) \iff n^*(\alpha, \beta) \land \neg(n^*(\alpha, c) \land n^+(c, \beta))
\]
Canonical Abstraction

- Model the heap as a set of relations evolve over time
- Not just binary
- Every relation has three values $0 \sqcup 1 = \frac{1}{2}$ (don’t know)
- Partition the individuals into equivalence classes based on the values in the unary relations
  - Every individual is mapped to its equivalence class
- Combine relations via $\sqcap$
  
  $p^S(u'_1, ..., u'_k) = \sqcap \{ p^B(u_1, ..., u_k) \mid f(u_1) = u'_1, ..., f(u_k) = u'_k \}$
- At most $2^A$ abstract individuals
- The basis of Three-Valued-Analysis (TVLA)

[TOPLAS’02] S. Sagiv, T.W. Reps, R. Wilhelm: Parametric Shape Analysis via 3-Valued Logic
x = NULL;

while (...) do {
    t = malloc();
    t →next=x;
    x = t
}

Canonical Abstraction
x = NULL;
while (...) do {
    t = malloc();
    t -> next = x;
    x = t
}

∀V: (x = V ∧ t = V) ∨ (x ≠ V ∧ t ≠ V)
∀V, W: (x ≠ V ∧ t ≠ V) ∧ (x = W ∧ t = W) → ¬n(V, W)
Don’t go generic!
Domain Specialization

- Many programs manipulate specialized data structures
  - singly, doubly-linked (circular) lists, trees
- Design specialized abstract domains
- Similar to theories in decision procedures

[POPL’96,TOPLAS’98] Shmuel Sagiv, Thomas W. Reps, Reinhard Wilhelm: Solving Shape-Analysis Problems in Languages with Destructive Updating
[TACAS’06] D. Distefano, P.W. O’Hearn, H. Yang: A Local Shape Analysis Based on Separation Logic
The Instrumentation Principle

- Users define extra derived relations
- Jargon for expressing inductive invariants
- Refines the abstraction
- Refines concretization
- TVLA generates update-formulas

[TOPLAS’10] T.W. Reps, M. Sagiv, A. Loginov:
Finite differencing of logical formulas for static analysis
Heap-Sharing Relation

\[ is(V) = \exists V_1, V_2: n(V_1, V) \land n(V_2, V) \land V_1 \neq V_2 \]

\[ \forall V: (x = V \land t = V \land \neg is(V)) \lor (x \neq V \land y \neq V \land \neg is(V)) \]
Heap-Sharing Relation

\[ \text{is}(V) = \exists V_1, V_2: n(V_1, V) \land n(V_2, V) \land V_1 \neq V_2 \]

\[ \forall V: (x = V \land t = V \land \neg \text{is}(V)) \lor (x \neq V \land y \neq V \land \neg \text{is}(V)) \]
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

$t[n](v_1, v_2) = n^*(v_1,v_2)$
List Segments

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

\[ x \rightarrow n \rightarrow y \]

\[ x \rightarrow n \rightarrow u_{2,3,4,6,7,8} \rightarrow n \rightarrow y \]
Reachability from a variable

- $r_y(v) = n^*(y, v)$
Sortedness
Example: Sortedness

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

\[
\text{inOrder} = 1 \\
\text{dle} \\
u_1 \\
x \\
t \\
\text{dle} \\
u_2 \\
\text{dle} \\
\ldots \\
\text{dle} \\
u_n \\
\text{dle}
\]

\[
\text{inOrder} = 1 \\
\text{dle} \\
u_{1..n} \\
x \\
t \\
\text{dle} \\
u_2..n \\
\text{dle}
\]

\[
\text{inOrder} = 1 \\
\text{dle} \\
\]

\[
\text{inOrder} = 1 \\
\text{dle} \\
\]
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;
}

inOrder = 1
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;
}

inOrder = 1/2  inOrder = 1
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)⇔ reach[root](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}
        */
    }
    assert(marked = Reachset(root))
}
```
Bug Found

• There may exist an individual that is reachable from the root, but not marked

\[ \exists r: \text{root}(r) \land r[\text{root}](r) \land \neg p(r) \land m(r) \land r[\text{root}](e) \land \neg m(e) \land \neg \text{root}(e) \land \neg p(e) \]

\[ \forall r, e: \text{root}(r) \land r[\text{root}](r) \land \neg p(r) \land m(r) \land r[\text{root}](e) \land \neg m(e) \land \neg \text{root}(e) \land \neg p(e) \]

\[ \rightarrow \neg \text{left}(r, e) \]
Materialization
• Materialize elements from summary nodes
• Exploit locality of the program
  – Small changes in the invariant

[LISP’93] E. Wang, P.N. Hilfinger: Analysis of Recursive Types in Lisp-Like Languages


Best Transformer [CC79]

Concrete Semantics

Abstract Semantics

\[ s = \text{Top} \rightarrow n \]

\[ s = \text{Top} \rightarrow n \]

Abstraction

\[ s = \text{Top} \rightarrow n \]
Then a Miracle Occurs

“I think you should be more explicit here in step two.”
Local Concretization Based Transformer

Materialization (Local Concretization)

Abstract Semantics

Abstraction

$$\llbracket s=\text{Top} \rightarrow n \rrbracket$$

$$\llbracket s=\text{Top} \rightarrow n \rrbracket^\#$$
Semantic Reduction

- Improve the precision of the analysis by recovering properties of the program semantics

- A Galois connection \((L_1, \alpha, \gamma, L_2)\)

- An operation \(\text{op}: L_2 \rightarrow L_2\) is a **semantic reduction**
  
  \[ \forall l \in L_2 \quad \text{op}(l) \sqsubseteq l \]
  
  \[ \gamma(\text{op}(l)) = \gamma(l) \]

- Can be applied before and after basic operations
“Focus”-Based Transformer \((x = x \rightarrow n)\)

Kleene Evaluation

Focus\((x \rightarrow n)\) \Rightarrow “Partial \(\gamma\)”

canonical
The Focus Operation

- Focus: Formula $\rightarrow (P(3\text{-Struct}) \leftarrow P(3\text{-Struct}))$
- Generalizes materialization
- For every formula $\varphi$
  - Focus($\varphi$)(X) yields structure in which $\varphi$ evaluates to definite values in all assignments
  - Only maximal in terms of embedding
  - Focus($\varphi$) is a semantic reduction
  - But Focus($\varphi$)(X) may be undefined for some X
“Focus”-Based Transformer \((x = x \rightarrow n)\)

\[\exists w: x(w) \land n(w, v)\]

Kleene Evaluation

Focus\((x \rightarrow n)\)

“Partial \(\gamma\)”

canonical
The Coercion Principle

- Another Semantic Reduction
- Can be applied after Focus or after Update or both
- Increase precision by exploiting some structural properties possessed by all stores (Global invariants)
- Structural properties captured by constraints
- Apply a constraint solver
Apply Constraint Solver

\[ r[n,y](v) = 1 \]

\[ \text{is}(v) = 0 \]

\[ \text{is}(v) = 0 \]
Sources of Constraints

- Properties of the operational semantics
- Domain specific knowledge
  - Instrumentation predicates
- User supplied
Example Constraints

\[ x(v_1) \land x(v_2) \rightarrow eq(v_1, v_2) \]

\[ n(v, v_1) \land n(v, v_2) \rightarrow eq(v_1, v_2) \]

\[ n(v_1, v) \land n(v_2, v) \land \neg eq(v_1, v_2) \leftrightarrow is(v) \]

\[ n^*(v_3, v_4) \leftrightarrow t[n](v_1, v_2) \]
Apply Constraint Solver

\[ y = \max(v) = 0 \]

\[ x(v_1) \land x(v_2) \rightarrow \text{eq}(v_1, v_2) \]

\[ y = \max(v) = 0 \]
Apply Constraint Solver

\[ y\text{is}(v) = 0 \]

\[ n(v_1, v) \land n(v_2, v) \land \neg eq(v_1, v_2) \iff is(v) \]

\[ n(v_1, v) \land \neg is(v) \land \neg eq(v_1, v_2) \rightarrow \neg n(v_2, v) \]
Summary Transformers

- Kleene evaluation yields sound solution
- Focus is statement specific implements partial concretization
- Coerce applies global constraints
How to tabulate procedures?
N. Rinetzky

- Procedure $\equiv$ input/output relation
  - Not reachable $\Rightarrow$ Not affected
  - proc: local ($\equiv$reachable) heap $\Rightarrow$ local heap
How to handle sharing?

• External sharing may break the functional view
Cutpoints

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

```plaintext
append(y,z)
```

```
append(y,z)
```
Cutpoints

[POPL’05] N. Rinetzky, J. Bauer, T. W. Reps, S.l Sagiv, R. Wilhelm A semantics for procedure local heaps and its abstractions
[SAS’05] N. Rinetzky, M. Sagiv, E. Yahav: Interprocedural Shape Analysis for Cutpoint-Free Programs
[SAS’06] A. Gotsman, J. Berdine, B. Cook: Interprocedural Shape Analysis with Separated Heap Abstractions
[LMCS11,TACAS09] M. Faouzi Atig, A. Bouajjani, S.z Qadeer: Context-Bounded Analysis For Concurrent Programs With Dynamic Creation of Threads
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]
Partially Disjunctive Heap Abstraction (Manevich, SAS’04)

- Use a heap-similarity criterion
  - We defined similarity by universe congruence
- Merge similar heaps
- Avoid merging dissimilar heaps
- The same concrete state can belong to more than one abstract value
Partially Disjunctive Abstraction
Running times
Compositionality

• Apply shape analysis to one procedure at a time
• Calculate the effect of the procedure on the reachable heap
• Propagate the effect into the clients
• Useful in incremental settings
• Adopted by Facebook

[PLDI’11] Isil Dillig, Thomas Dillig, Alex Aiken, Mooly Sagiv: Precise and compact modular procedure summaries for heap manipulating programs.
Shape Analysis Principles

- Reason about directed reachability
- Specialized data structures
- Explore locality of updates
  - Materialization
- Explore locality of procedure and type safety for updating reachability [POPL ’05]
- Bottom-up shape analysis [POPL ’09, JACM ’11]
- Partially disjunctive analysis [SAS ’04, CAV ’07]

[POPL’09, JACM’11] C. Calcagno, Dino Distefano, Peter W. O'Hearn, Hongseok Yang: Compositional Shape Analysis by Means of Bi-Abduction
Bug Found

- There may exist an individual that is reachable from the root, but not marked

\[
\begin{align*}
\forall r, e: & \ (\text{root}(r) \land \text{r[root]}(r) \land \neg \text{p}(r) \land \text{m}(r) \land \text{r[root]}(e) \land \neg \text{m}(e) \land \neg \text{root}(e) \land \neg \text{p}(e)) \\
& \rightarrow \neg \text{left}(r,e)
\end{align*}
\]
## Properties Proved

<table>
<thead>
<tr>
<th>Program</th>
<th>Properties</th>
<th>#Graphs</th>
<th>Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>LindstromScan</td>
<td>CL, DI</td>
<td>1285</td>
<td>8.2</td>
</tr>
<tr>
<td>LindstromScan</td>
<td>CL, DI, IS, TE</td>
<td>183564</td>
<td>2185</td>
</tr>
<tr>
<td>SetRemove</td>
<td>CL, DI, SO</td>
<td>13180</td>
<td>106</td>
</tr>
<tr>
<td>SetInsert</td>
<td>CL, DI, SO</td>
<td>299</td>
<td>1.75</td>
</tr>
<tr>
<td>DeleteSortedTree</td>
<td>CL, DI</td>
<td>2429</td>
<td>6.24</td>
</tr>
<tr>
<td>DeleteSortedTree</td>
<td>CL, DI, SO</td>
<td>30754</td>
<td>104</td>
</tr>
<tr>
<td>InsertSortedTree</td>
<td>CL, DI</td>
<td>177</td>
<td>0.85</td>
</tr>
<tr>
<td>InsertSortedTree</td>
<td>CL, DI, SO</td>
<td>1103</td>
<td>2.5</td>
</tr>
<tr>
<td>InsertAVLttree</td>
<td>CL, DI, SO</td>
<td>1855</td>
<td>27.4</td>
</tr>
<tr>
<td>RecQuickSot</td>
<td>CL, DI, SO</td>
<td>5585</td>
<td>9.2</td>
</tr>
</tbody>
</table>

CL=memory safety    DI=data structure invariant    TE=termination    SO=sorted
Applying Shape Analysis to Real Code

Lukás Holík, Ondrej Lengál, Adam Rogalewicz, Jirí Simácek, Tomás Vojnar: Fully Automated Shape Analysis Based on Forest Automata. CAV 2013: 740-755


Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn: Scalable Shape Analysis for Systems Code. CAV 2008: 385-398

Alexey Loginov, Eran Yahav, Satish Chandra, Stephen Fink, Noam Rinetzky, Mangala Gowri Nanda: Verifying dereference safety via expanding-scope analysis. ISSTA 2008: 213-224

Eran Yahav, G. Ramalingam: Verifying safety properties using separation and heterogeneous abstractions. PLDI 2004: 25-3
Limitations of Shape Analysis

- Complex data structures
- Concurrency
- Modularity & libraries
- False alarms
thttpd: Web Server

Representation Invariants:
1. ∀n: Map. ∀v: Z.
   table[v] = n ⇒ n->index = v

2. ∀n: Map.
   n->rc = |{n' : Conn . n’->file_data = n}|
static void add_map(Map *m)
{
    int i = hash(m);
    ...  
    table[i] = m ;
    ...  
    m->index= i ;
    ...  
    m->rc++; 
}

Representation Invariants:
1. \( \forall n: \text{Map.} \ \forall v: \mathbb{Z}. \ \text{table}[v] = n \Rightarrow \text{index}[n] = v \)
2. \( \forall n: \text{Map.} \ \text{rc}[n] = |\{n': \text{Conn.} . \ \text{file_data}[n'] = n\}| \)