Program Analysis and Verification

0368-4479

Noam Rinetzky

Lecture 11: Shape Analysis + Interprocedural Analysis

Slides credit: Roman Manevich, Mooly Sagiv, Eran Yahav
3-Value logic based shape analysis
Sequential Stack

void push (int v) {
    Node *x = malloc(sizeof(Node));
    x->d = v;
    x->n = Top;
    Top = x;
}

int pop() {
    if (Top == NULL) return EMPTY;
    Node *s = Top->n;
    int r = Top->d;
    Top = s;
    return r;
}

Want to Verify
No Null Dereference
Underlying list remains acyclic after each operation
Shape Analysis via 3-valued Logic

1) Abstraction
   – 3-valued logical structure
   – canonical abstraction

2) Transformers
   – via logical formulae
   – soundness by construction
     • embedding theorem, [SRW02]
Concrete State

• represent a concrete state as a two-valued logical structure
  – Individuals = heap allocated objects
  – Unary predicates = object properties
  – Binary predicates = relations

• parametric vocabulary

(storeless, no heap addresses)
Concrete State

- $S = <U, \iota>$ over a vocabulary $P$
- $U$ – universe
- $\iota$ - interpretation, mapping each predicate from $p$ to its truth value in $S$

- $U = \{ u_1, u_2, u_3 \}$
- $P = \{ \text{Top}, n \}$
- $\iota(n)(u_1,u_2) = 1$, $\iota(n)(u_1,u_3)=0$, $\iota(n)(u_2,u_1)=0$, ...
- $\iota(\text{Top})(u_1)=1$, $\iota(\text{Top})(u_2)=0$, $\iota(\text{Top})(u_3)=0$
void push (int v) {
    Node *x = malloc(sizeof(Node));
    \exists w: x(w)
x->d = v;   \exists w: x(w)
x->n = Top;
    \exists w: Top(w)
    Top = x;
}

\neg \exists v1,v2: n(v1, v2) \land n*(v2, v1)
\neg \exists v1,v2: n(v1, v2) \land Top(v2)
## Concrete Interpretation Rules

<table>
<thead>
<tr>
<th>Statement</th>
<th>Update formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = \text{NULL} )</td>
<td>( x'(v) = 0 )</td>
</tr>
<tr>
<td>( x = \text{malloc}() )</td>
<td>( x'(v) = \text{IsNew}(v) )</td>
</tr>
<tr>
<td>( x = y )</td>
<td>( x'(v) = y(v) )</td>
</tr>
<tr>
<td>( x = y \rightarrow \text{next} )</td>
<td>( x'(v) = \exists w: y(w) \land n(w, v) )</td>
</tr>
<tr>
<td>( x \rightarrow \text{next} = y )</td>
<td>( n'(v, w) = (\neg x(v) \land n(v, w)) \lor (x(v) \land y(w)) )</td>
</tr>
</tbody>
</table>
Example: \( s = \text{Top} \rightarrow n \)

\[ s'(v) = \exists v_1: \text{Top}(v_1) \land n(v_1, v) \]

<table>
<thead>
<tr>
<th>Top</th>
<th>( u_1 )</th>
<th>( u_2 )</th>
<th>( u_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>u1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>u2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>u3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>( u_1 )</th>
<th>( u_2 )</th>
<th>( u_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>u1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>u2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>u3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Top</th>
<th>( u_1 )</th>
<th>( u_2 )</th>
<th>( u_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>u1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>u2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>u3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>( u_1 )</th>
<th>( u_2 )</th>
<th>( u_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>u1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>u2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>u3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s</th>
<th>( u_1 )</th>
<th>( u_2 )</th>
<th>( u_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>u1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>u2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>u3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Collecting Semantics

\[ \text{CSS}[v] = \begin{cases} \{ <\emptyset,\emptyset> \} & \text{if } v = \text{entry} \\ \bigcup \{ \llbracket \text{st}(w) \rrbracket(S) \mid S \in \text{CSS}[w] \} \cup \{ (w,v) \in \text{E}(G), \newline \quad w \in \text{Assignments}(G) \} \\
\bigcup \{ S \mid S \in \text{CSS}[w] \} \cup \{ (w,v) \in \text{E}(G), \newline \quad w \in \text{Skip}(G) \} \\
\bigcup \{ S \mid S \in \text{CSS}[w] \text{ and } S \models \text{cond}(w) \} \cup \{ (w,v) \in \text{True-Branches}(G) \} \\
\bigcup \{ S \mid S \in \text{CSS}[w] \text{ and } S \models \neg\text{cond}(w) \} \cup \{ (w,v) \in \text{False-Branches}(G) \} & \text{otherwise} \end{cases} \]
Collecting Semantics

• At every program point – a potentially infinite set of two-valued logical structures
• Representing (at least) all possible heaps that can arise at the program point

• Next step: find a bounded abstract representation
3-Valued Logic

- $1 = \text{true}$
- $0 = \text{false}$
- $1/2 = \text{unknown}$

- A join semi-lattice, $0 \sqcup 1 = 1/2$
3-Valued Logical Structures

• A set of individuals (nodes) $U$

• Relation meaning
  – Interpretation of relation symbols in $P$
    $i(p^0)() \rightarrow \{0, 1, 1/2\}$
    $i(p^1)(v) \rightarrow \{0, 1, 1/2\}$
    $i(p^2)(u,v) \rightarrow \{0, 1, 1/2\}$

• A join semi-lattice: $0 \sqcup 1 = 1/2$
### Boolean Connectives [Kleene]

<table>
<thead>
<tr>
<th>$\land$</th>
<th>0</th>
<th>1/2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1/2</td>
<td>0</td>
<td>1/2</td>
<td>1/2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1/2</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\lor$</th>
<th>0</th>
<th>1/2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1/2</td>
<td>1</td>
</tr>
<tr>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Property Space

- $3\text{-struct}[P] = \text{the set of 3-valued logical structures over a vocabulary (set of predicates) } P$

- Abstract domain
  - $\emptyset (3\text{-Struct}[P])$
  - $\subseteq$ is $\subseteq$
Embedding Order

• Given two structures $S = <U, \iota >$, $S' = <U', \iota '>\text{ and an onto function } f : U \rightarrow U' \text{ mapping individuals in } U \text{ to individuals in } U'$

• We say that $f$ embeds $S$ in $S'$ (denoted by $S \sqsubseteq S'$) if
  – for every predicate symbol $p \in P$ of arity $k$: $u_1, ..., u_k \in U$, $\iota (p)(u_1, ..., u_k) \sqsubseteq \iota '(p)(f(u_1), ..., f(u_k))$
  – and for all $u' \in U'$
    $\{ u \mid f(u) = u' \} > 1 \sqsubseteq \iota '(sm)(u')$

• We say that $S$ can be embedded in $S'$ (denoted by $S \sqsubseteq^f S'$) if there exists a function $f$ such that $S \sqsubseteq^f S'$
Tight Embedding

• $S' = \langle U', \iota' \rangle$ is a tight embedding of $S=\langle U, \iota \rangle$ with respect to a function $f$ if:
  – $S'$ does not lose unnecessary information

\[
\iota'(u'_1, \ldots, u'_k) = \bigsqcap \{ \iota(u_1, \ldots, u_k) \mid f(u_1) = u'_1, \ldots, f(u_k) = u'_k \}
\]

• One way to get tight embedding is canonical abstraction
Canonical Abstraction

[Top] $\rightarrow$ $u_1$ $\rightarrow$ $u_2$ $\rightarrow$ $u_3$

[Sagiv, Reps, Wilhelm, TOPLAS02]
Canonical Abstraction

[Top] → u1 → n → u2 → n → u3

[Top] → u2

[Sagiv, Reps, Wilhelm, TOPLAS02]
 Canonical Abstraction

Top

u1

n

u2

n

u3

Top


Canonical Abstraction
Canonical Abstraction
Canonical Abstraction
Canonical Abstraction ($\beta$)

- Merge all nodes with the same unary predicate values into a single summary node.
- Join predicate values:
  $$\iota'(u'_1, \ldots, u'_k) = \bigsqcup \{ \iota(u_1, \ldots, u_k) \mid f(u_1)=u'_1, \ldots, f(u_k)=u'_k \}$$
- Converts a state of arbitrary size into a 3-valued abstract state of bounded size.
- $$\alpha(C) = \bigsqcup \{ \beta(c) \mid c \in C \}$$
Information Loss
Instrumentation Predicates

- Record additional derived information via predicates

\[
\begin{align*}
    r_x(v) &= \exists v_1: x(v_1) \land n^*(v_1, v) \\
    c(v) &= \exists v_1: n(v_1, v) \land n^*(v, v_1)
\end{align*}
\]
Embedding Theorem: **Conservatively** Observing Properties

No Cycles

\[ \neg \exists v_1, v_2: n(v_1, v_2) \land n^*(v_2, v_1) \]

No cycles (derived)

\[ \forall v: \neg c(v) \]
void push (int v) {
    Node *x = malloc(sizeof(Node));
    x->d = v;
    x->n = Top;
    Top = x;
}

int pop() {
    if (Top == NULL) return EMPTY;
    Node *s = Top->n;
    int r = Top->d;
    Top = s;
    return r;
}
Abstract Semantics

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

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

\[ s'(v) = \exists v_1: \text{Top}(v_1) \land n(v_1,v) \]
Best Transformer ($s = \text{Top} \rightarrow n$)

Concrete Semantics

Abstract Semantics

Canonical Abstraction

$s'(v) = \exists v_1: \text{Top}(v_1) \land n(v_1, v)$
Semantic Reduction

- Improve the precision of the analysis by recovering properties of the program semantics
- A Galois connection $(C, \alpha, \gamma, A)$
- An operation $\text{op}: A \rightarrow A$ is a **semantic reduction** when
  - $\forall l \in L_2 \ \text{op}(l) \sqsubseteq l$ and
  - $\gamma(\text{op}(l)) = \gamma(l)$
The Focus Operation

• Focus: Formula → (φ(3-Struct) ⇔ φ(3-Struct))

• Generalizes materialization

• For every formula φ
  – Focus(φ)(X) yields structure in which φ evaluates to a definite values in all assignments
  – Only maximal in terms of embedding
  – Focus(φ) is a semantic reduction
  – But Focus(φ)(X) may be undefined for some X
Partial Concretization Based on Transformer ($s=\text{Top} \rightarrow n$)

Abstract Semantics

Canonical Abstraction

$S' = \exists v_1: \text{Top}(v_1) \land n(v_1, v)$

Focus $\tau_n$
Partial Concretization

- Locally refine the abstract domain per statement
- Soundness is immediate
- Employed in other shape analysis algorithms
  [Distefano et.al., TACAS’06, Evan et.al., SAS’07, POPL’08]
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
Sources of Constraints

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

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

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

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

\[ n^*(v_1, v_2) \leftrightarrow t[n](v1, v_2) \]
Abstract Transformers: Summary

• Kleene evaluation yields sound solution
• Focus is a statement-specific partial concretization
• Coerce applies global constraints
Abstract Semantics

\[
SS[v] = \begin{cases}
\{<\emptyset, \emptyset>\} & \text{if } v = \text{entry} \\
\bigcup \{ t\_embed(\text{coerce}(\llbracket st(w) \rrbracket_3(\text{focus}_{F(w)}(SS[w])))) \} & \text{otherwise} \\
\bigcup \{ S \mid S \in SS[w] \} \cup \\
\bigcup \{ t\_embed(S) \mid S \in \text{coerce}(\llbracket st(w) \rrbracket_3(\text{focus}_{F(w)}(SS[w]))) \text{ and } S \models 3 \text{ cond}(w) \} \cup \\
\bigcup \{ t\_embed(S) \mid S \in \text{coerce}(\llbracket st(w) \rrbracket_3(\text{focus}_{F(w)}(SS[w]))) \text{ and } S \models 3 \neg \text{cond}(w) \} \cup \\
\end{cases}
\]
Recap

• Abstraction
  – canonical abstraction
  – recording derived information

• Transformers
  – partial concretization (focus)
  – constraint solver (coerce)
  – sound information extraction
void push (int v) {
    Node *x = alloc(sizeof(Node));
    x->d = v;
    x->n = Top;
    Top = x;
}

∀v:¬c(v)
¬∃v1,v2: n(v1, v2) ∧ Top(v2)
What about procedures?
Procedural program

void main() {
    int x;
    x = p(7);
    x = p(9);
}

int p(int a) {
    return a + 1;
}
Effect of procedures

The effect of calling a procedure is the effect of executing its body.
goal: compute the abstract effect of calling a procedure
Reduction to intraprocedural analysis

- Procedure inlining
- Naive solution: call-as-goto
Reminder: Constant Propagation

\[ \begin{align*} \top & \quad \text{Variable not a constant} \\ \bot & \quad \text{No information} \\ -\infty & \quad \ldots \\ -1 & \quad 0 \\ 1 & \quad \ldots \\ \infty & \end{align*} \]
Reminder: Constant Propagation

- \( L = (\text{Var} \rightarrow \mathbb{Z}, \sqsubseteq) \)
- \( \sigma_1 \sqsubseteq \sigma_2 \) iff \( \forall x: \sigma_1(x) \sqsubseteq' \sigma_2(x) \)
  - \( \sqsubseteq' \) ordering in the \( \mathbb{Z} \) lattice

- Examples:
  - \([x \mapsto \perp, y \mapsto 42, z \mapsto \perp] \sqsubseteq [x \mapsto \perp, y \mapsto 42, z \mapsto 73]\)
  - \([x \mapsto \perp, y \mapsto 42, z \mapsto 73] \sqsubseteq [x \mapsto \perp, y \mapsto 42, z \mapsto \top]\)
Reminder: Constant Propagation

• Conservative Solution
  – Every detected constant is indeed constant
    • But may fail to identify some constants
  – Every potential impact is identified
    • Superfluous impacts
Procedure Inlining

```c
void main() {
    int x;
    x = p(7);
    x = p(9);
}

int p(int a) {
    return a + 1;
}
```
Procedure Inlining

```c
int p(int a) {
    return a + 1;
}

void main() {
    int x;
    x = p(7);
    x = p(9);
}

void main() {
    int a, x, ret;
    a = 7; ret = a+1; x = ret;
    [a ↦ 7, x ↦ 8, ret ↦ 8]
    a = 9; ret = a+1; x = ret;
    [a ↦ 9, x ↦ 10, ret ↦ 10]
}
```

52
Procedure Inlining

• Pros
  – Simple

• Cons
  – Does not handle recursion
  – Exponential blow up
  – Reanalyzing the body of procedures

\[
\begin{align*}
\text{p1} \{ & \\
\text{call p2} & \\
\text{...} & \\
\text{call p2} & \\
\} \\
\text{p2} \{ & \\
\text{call p3} & \\
\text{...} & \\
\text{call p3} & \\
\} \\
\text{p3} \{ & \\
\} \\
\end{align*}
\]
A Naive Interprocedural solution

• Treat procedure calls as gotos
void main() {
    int x;
    x = p(7);
    x = p(9);
}
int p(int a) {
    return a + 1;
}
void main() {
    int x;
    x = p(7);
    x = p(9);
}

int p(int a) {
    [a \rightarrow 7]
    return a + 1;
}
void main() {
    int x ;
    x = p(7);
    x = p(9) ;
}

int p(int a) {
    [a ↦ 7]
    return a + 1;
    [a ↦ 7, $\$ \mapsto 8]
}

diagram of function calls and return values
void main() {
    int x;
    x = p(7);
    \[x \mapsto 8\]
    x = p(9);
    \[x \mapsto 8\]
}

int p(int a) {
    [a \mapsto 7]
    return a + 1;
    [a \mapsto 7, $$ \mapsto 8$$]
}

58
void main() {
    int x;
    x = p(7);
    [x \mapsto 8]
    x = p(9);
    [x \mapsto 8]
}

int p(int a) {
    [a \mapsto 7]
    return a + 1;
    [a \mapsto 7, $$ \mapsto 8]$
}

\textbf{Simple Example}
void main() {
    int x;
    x = p(7);
    [x \rightarrow 8]
    x = p(9);
    [x \rightarrow 8]
}

int p(int a) {
    [a \leftrightarrow 7, a \rightarrow 9]
    return a + 1;
    [a \leftrightarrow 7, $$ \leftrightarrow 8]
}
void main() {
    int x;
    x = p(7);
    [x \rightarrow 8]
    x = p(9);
    [x \rightarrow 8]
}

int p(int a) {
    [a \rightarrow 1]
    return a + 1;
    [a \rightarrow 7, $$ \rightarrow 8]
}

int p(7)
retc p(7)
call p(9)
retc p(9)

ret a+1
void main() {
    int x;
    x = p(7);
    [x ← 8]
    x = p(9);
    [x ← 8]
}

int p(int a) {
    [a ← 1]
    return a + 1;
    [a ← 1, $$ ← 1]
}

---

**Simple Example**

```c
int p(int a) {
    return a + 1;
}

void main() {
    int x;
    x = p(7);
    [x ← 8]
    x = p(9);
    [x ← 8]
}
```

---

Diagram:
- **main()** block
  - Call to **p(7)**
    - Return to **p(7)**
      - Call to **p(9)**
        - Return to **p(9)**
          - Return to **main()**
void main() {
    int x;
    x = p(7);
    [x \mapsto \tau]
    x = p(9);
    [x \mapsto \tau]
}

int p(int a) {
    [a \mapsto \tau]
    return a + 1;
    [a \mapsto \tau, \$\$ \mapsto \tau]
}

63
A Naive Interprocedural solution

• Treat procedure calls as gotos
• Pros:
  – Simple
  – Usually fast
• Cons:
  – Abstract call/return correlations
  – Obtain a conservative solution
## analysis by reduction

### Call-as-goto

```c
void main() {
    int x;
    x = p(7);
    x = p(9);
}
```

### Procedure inlining

```c
int p(int a) {
    return a + 1;
}
```

```c
void main() {
    int a, x, ret;
    a = 7; ret = a+1; x = ret;
    a = 9; ret = a+1; x = ret;
}
```

**why was the naive solution less precise?**
Stack regime

P() {
    ...
    R();
    ...
}

R() {
    ...
}

Q() {
    ...
    R();
    ...
}

R
P

66
Guiding light

• Exploit stack regime
  ➔ Precision
  ➔ Efficiency
Simplifying Assumptions

- Parameter passed by value
- No procedure nesting
- No concurrency

✓ Recursion is supported
Topics Covered

✓ Procedure Inlining
✓ The naive approach
  • Valid paths
  • The callstring approach
  • The Functional Approach
  • IFDS: Interprocedural Analysis via Graph Reachability
  • IDE: Beyond graph reachability

• The trivial modular approach
Join-Over-All-Paths (JOP)

• Let paths(v) denote the potentially infinite set paths from start to v (written as sequences of edges)

• For a sequence of edges \([e_1, e_2, \ldots, e_n]\) define
  \(f[e_1, e_2, \ldots, e_n]: \mathbb{L} \rightarrow \mathbb{L}\) by composing the effects of basic blocks
  \(f[e_1, e_2, \ldots, e_n](l) = f(e_n)(\ldots(f(e_2)(f(e_1)(l))\ldots)\)

• \(JOP[v] = \bigcup\{f[e_1, e_2, \ldots,e_n](l) \mid [e_1, e_2, \ldots, e_n] \in \text{paths}(v)\}\)
Join-Over-All-Paths (JOP)

Paths transformers:
- $f[e_1,e_2,e_3,e_4]$
- $f[e_1,e_2,e_7,e_8]$
- $f[e_5,e_6,e_7,e_8]$
- $f[e_5,e_6,e_3,e_4]$
- $f[e_1,e_2,e_7,e_8,e_9, e_1,e_2,e_3,e_4,e_9,...]$ (initial)
- $f[e_5,e_6,e_7,e_8](initial)$ (initial)
- $f[e_5,e_6,e_3,e_4](initial)$ (initial)
- $f[e_5,e_6,e_3,e_4](initial)$ (initial) ...

Number of program paths is unbounded due to loops
The lfp computation approximates JOP

- \( JOP[v] = \bigcup \{ f[e_1, e_2, ..., e_n](v) \mid [e_1, e_2, ..., e_n] \in \text{paths}(v) \} \)

- \( LFP[v] = \bigcup \{ f[e](LFP[v']) \mid e = (v', v) \} \)
  \[ LFP[v_0] = \mathcal{I} \]

- \( JOP \subseteq LFP \) - for a monotone function
  - \( f(x \uplus y) \supseteq f(x) \uplus f(y) \)

- \( JOP = LFP \) - for a distributive function
  - \( f(x \uplus y) = f(x) \uplus f(y) \)

JOP may not be precise enough for interprocedural analysis!
Interprocedural analysis

Supergraph
• **paths(n)** the set of paths from s to n
  – ( (s,n₁), (n₁,n₃), (n₃,n₁) )
Interprocedural Valid Paths

IVP: all paths with matching calls and returns

And prefixes
Interprocedural Valid Paths

- **IVP** set of paths
  - Start at program entry
- Only considers matching calls and returns
  - aka, valid
- Can be defined via context free grammar
  - matched ::= matched (i matched )i | ε
  - valid ::= valid (i matched | matched
    - *paths* can be defined by a regular expression
Join Over All Paths (JOP)

\[ \text{JOP}[v] = \bigcup \{ [[e_1, e_2, \ldots, e_n]](v) \mid (e_1, \ldots, e_n) \in \text{paths}(v) \} \]

- \( \text{JOP} \subseteq \text{LFP} \)
  - Sometimes \( \text{JOP} = \text{LFP} \)
    - precise up to “symbolic execution”
    - Distributive problem

\[ \llbracket f_k \circ \ldots \circ f_1 \rrbracket \in L \to L \]
The Join-Over-Valid-Paths (JVP)

- \( vpaths(n) \) all valid paths from program start to \( n \)

- \( JVP[n] = \bigcup \{[[e_1, e_2, \ldots, e]](i) \mid (e_1, e_2, \ldots, e) \in vpaths(n) \} \)

- \( JVP \sqsubseteq JOP \)
  - In some cases the JVP can be computed
  - (Distributive problem)
The Call String Approach

• The data flow value is associated with sequences of calls (call string)

• Use Chaotic iterations over the supergraph
void main() {
    int x;
    c1: x = p(7);
    c2: x = p(9);
}

int p(int a) {
    return a + 1;
}
void main() {
    int x;
    c1: x = p(7);
    c2: x = p(9);
}

int p(int a) {
    c1: [a ↦ 7]
    return a + 1;
}
void main() {
    int x ;
    c1: x =  p(7);
    c2: x =  p(9) ;
}

int p(int a) {
    return a + 1;
}

// Example
// c1: [a → 7]
// c1: [a → 7, $$$ → 8]

// Inference
// x = p(7)
Simple Example

void main() {
    int x;
    c1: x = p(7);
    \[ x \mapsto 8 \]
    c2: x = p(9);
}

int p(int a) {
    c1: [a \mapsto 7]
    return a + 1;
    c1: [a \mapsto 7, $$ \mapsto 8]
}

Simple Example

```c
void main() {
    int x;
    c1: x = p(7);
    ε: [x ↦ 8]
    c2: x = p(9);
}
```

```c
int p(int a) {
    c1:[a ↦7]
    return a + 1;
    c1:[a ↦7, $$ ↦ 8]
}
```
Simple Example

```c
void main() {
    int x;
    c1: x = p(7);
    ε: [x ↦ 8]
    c2: x = p(9);
}

int p(int a) {
    c1:[a ↦7]
    c2:[a ↦9]
    return a + 1;
    c1:[a ↦7, $$ ↦8]
}
```
Simple Example

```c
void main() {
    int x ;
    c1: x = p(7);
    ε : [x ↦ 8]
    c2: x = p(9) ;
}

int p(int a) {
    c1:[a ↦7]
    c2:[a ↦9]
    return a + 1;
    c1:[a ↦7, $$ ↦8]
    c2:[a ↦9, $$ ↦10]
}
```
void main() {
    int x;
    c1: x = p(7);
    ε: [x ↦ 8]
    c2: x = p(9);
    ε: [x ↦ 10]
}

int p(int a) {
    c1:[a ↦ 7]
    c2:[a ↦ 9]
    return a + 1;
    c1:[a ↦ 7, $$ ↦ 8]
    c2:[a ↦ 9, $$ ↦ 10]
}

The Call String Approach

• The data flow value is associated with sequences of calls (call string)
• Use Chaotic iterations over the supergraph

• To guarantee termination limit the size of call string (typically 1 or 2)
  – Represents tails of calls

• Abstract inline
void main() {
    int x;
    c1: x = p(7);
    ε : [x ↦ 16]
    c2: x = p(9);
    ε :: [x ↦ 20]
}

int p(int a) {
    c1:[a ↦7]
    c2:[a ↦9]
    return c3: p1(a + 1);
    c1:[a ↦7, $$ ↦16]
    c2:[a ↦9, $$ ↦20]
}

int p1(int b) {
    c1.c3:[b ↦8]
    c2.c3:[b ↦10]
    return 2 * b;
    c1.c3:[b ↦8, $$ ↦16]
    c2.c3:[b ↦10, $$ ↦20]
}

Another Example (|cs|=2)
Another Example ($|cs|=1$)

void main() {
    int x ;
    c1: x = p(7);
    ε: [x ⇝ T]
    c2: x = p(9) ;
    ε: [x ⇝ T]
}

int p(int a) {
    c1:[a ⇝ 7]
    c2:[a ⇝ 9]
    return c3: p1(a + 1);
    c1:[a ⇝ 7, $\$\$ ⇝ T]
    c2:[a ⇝ 9, $\$\$ ⇝ T]
}

int p1(int b) {
    (c1|c2)c3:[b ⇝ T]
    return 2 * b;
    (c1|c2)c3:[b ⇝ T, $\$\$ ⇝ T]
}

Handling Recursion

```c
int p(int a) {
    c1: [a ↦ 7]  c1.c2+: [a ↦ τ]
    if (...) {
        c1: [a ↦ 7]  c1.c2+: [a ↦ τ]
        a = a - 1;
        c1: [a ↦ 6]  c1.c2+: [a ↦ τ]
        c2: p(a);
        c1.c2*: [a ↦ τ]
        a = a + 1;
        c1.c2*: [a ↦ τ]
    }
    c1.c2*: [a ↦ τ]
    x = -2*a + 5;
    c1.c2*: [a ↦ τ, x↦τ]
}
```
Summary Call String

• Easy to implement
• Efficient for very small call strings
• Limited precision
  – Often loses precision for recursive programs
  – For finite domains can be precise even with recursion (with a bounded callstring)

• Order of calls can be abstracted
• Related method: procedure cloning
The Functional Approach

- The meaning of a procedure is mapping from states into states
- The abstract meaning of a procedure is function from an abstract state to abstract states
- Relation between input and output
- In certain cases can compute JVP
The Functional Approach

• Two phase algorithm
  – Compute the dataflow solution at the exit of a procedure as a function of the initial values at the procedure entry (functional values)
  – Compute the dataflow values at every point using the functional values
Phase 1

void main() {
    p(7);
}

int p(int a) {
    [a ↦ a₀, x ↦ x₀]
    if (...) {
        [a ↦ a₀, x ↦ x₀]
        a = a - 1;
        [a ↦ a₀-1, x ↦ x₀]
        p (a);
        [a ↦ a₀-1, x ↦ -2a₀+7]
        a = a + 1;
        [a ↦ a₀, x ↦ -2a₀+7]
    }
    [a ↦ a₀, x ↦ x₀]  [a ↦ a₀, x ↦ τ]
    x = -2*a + 5;
    [a ↦ a₀, x ↦ -2*a₀+5]
}

p(a₀,x₀) = [a ↦ a₀, x ↦ -2a₀ + 5]
void main() {
    p(7);
    [x ↔ -9]
}

int p(int a) {
    if (…) {
        a = a - 1;
        p (a);
    }
    x = -2*a + 5;
    [a ↔ 7, x ↔ -9]  [a ↔ t, x ↔ t]
}

p(a₀, x₀) = [a ↔ a₀, x ↔ -2a₀ + 5]
Summary Functional approach

- Computes procedure abstraction
- Sharing between different contexts
- Rather precise
- Recursive procedures may be more precise/efficient than loops
- But requires more from the implementation
  - Representing (input/output) relations
  - Composing relations
Issues in Functional Approach

• How to guarantee that finite height for functional lattice?
  – It may happen that L has finite height and yet the lattice of monotonic function from L to L do not

• Efficiently represent functions
  – Functional join
  – Functional composition
  – Testing equality
Tabulation

• Special case: L is finite
• Data facts: d ∈ L × L
• Initialization:
  — \( f_{\text{start},\text{start}} = (T,T) \); otherwise \((\bot,\bot)\)
  — \( S[\text{start}, T] = T \)

• Propagation of \((x,y)\) over edge \(e = (n,n')\)
  - Maintain summary: \( S[n',x] = S[n',x] \sqcup [n] (y) \)
  - n intra-node: \( \Rightarrow n' : (x, [n] (y)) \)
  - n call-node:
    \( \Rightarrow n' : (y,y) \) if \( S[n',y] = \bot \) and \( n' = \) entry node
    \( \Rightarrow n' : (x,z) \) if \( S[\text{exit(call(n)},y] = z \) and \( n' = \) ret-site-of \( n \)
  - n return-node: \( \Rightarrow n' : (u,y) \); \( n_c = \) call-site-of \( n' \), \( S[n_c,u]=x \)
CFL-Graph reachability

• Special cases of functional analysis
• Finite distributive lattices
• Provides more efficient analysis algorithms
• Reduce the interprocedural analysis problem to finding context free reachability
IDFS / IDE

• **IDFS** Interprocedural Distributive Finite Subset
  Precise interprocedural dataflow analysis via graph reachability. *Reps, Horowitz, and Sagiv, POPL’ 95*

• **IDE** Interprocedural Distributive Environment
  Precise interprocedural dataflow analysis with applications to constant propagation. *Reps, Horowitz, and Sagiv, FASE’ 95, TCS’ 96*
  – *More general solutions exist*
Possibly Uninitialized Variables

```
possibly uninitialized variables
Start
x = 3
if . . .
y = x
w = 8
```

```
printf(y)
```
IFDS Problems

• Finite subset distributive
  – Lattice $L = \mathcal{P}(D)$
  – $\subseteq$ is $\subseteq$
  – $\sqcup$ is $\cup$
  – Transfer functions are distributive

• Efficient solution through formulation as CFL reachability
Encoding Transfer Functions

• Enumerate all input space and output space
• Represent functions as graphs with $2(D+1)$ nodes
• Special symbol “0” denotes empty sets (sometimes denoted $\Lambda$)
• Example: $D = \{ a, b, c \}$
  \[ f(S) = (S - \{a\}) \cup \{b\} \]
Efficiently Representing Functions

• Let $f: 2^D \rightarrow 2^D$ be a distributive function

• Then:
  - $f(X) = f(\emptyset) \cup (\bigcup \{ f(\{z\}) \mid z \in X \})$
  - $f(X) = f(\emptyset) \cup (\bigcup \{ f(\{z\}) \setminus f(\emptyset) \mid z \in X \})$
Representing Dataflow Functions

Identity Function
\[ f = \lambda V.V \]
\[ f(\{a, b\}) = \{a, b\} \]

Constant Function
\[ f = \lambda V.\{b\} \]
\[ f(\{a, b\}) = \{b\} \]
Representing Dataflow Functions

“Gen/Kill” Function

\[ f = \lambda V. (V - \{b\}) \cup \{c\} \]

\[ f(\{a, b\}) = \{a, c\} \]

Non-“Gen/Kill” Function

\[ f = \lambda V. \text{if } a \in V \]
\[ \text{then } V \cup \{b\} \]
\[ \text{else } V - \{b\} \]

\[ f(\{a, b\}) = \{a, b\} \]
\( x = 3 \)

\( p(x,y) \)

return from \( p \)

printf(y)

exit main

\( \Lambda^x \ y \)

\( \text{start main} \)

\( a \)

\( b \)

\( \Lambda \)

\( \text{start } p(a,b) \)

if \( \ldots \)

\( b = a \)

\( p(a,b) \)

return from \( p \)

printf(b)

exit p

exit main
Composing Dataflow Functions

\[ f_1 = \lambda V. \text{if } a \in V \]
\[ \text{then } V \cup \{b\} \]
\[ \text{else } V - \{b\} \]

\[ f_2 = \lambda V. \text{if } b \in V \]
\[ \text{then } \{c\} \]
\[ \text{else } \emptyset \]

\[ f_2 \circ f_1(\{a, c\}) = \{c\} \]
start main

\[ x = 3 \]

\[ p(x,y) \]

return from \( p \)

start \( p(a,b) \)

if \( \ldots \)

\[ b = a \]

\[ p(a,b) \]

return from \( p \)

printf(b)

\( \Lambda a \quad \Lambda b \)

Might \( y \) be uninitialized here?

YES!

Might \( y \) be uninitialized here?

NO!
The Tabulation Algorithm

• Worklist algorithm, start from entry of “main”
• Keep track of
  – Path edges: matched paren paths from procedure entry
  – Summary edges: matched paren call-return paths
• At each instruction
  – Propagate facts using transfer functions; extend path edges
• At each call
  – Propagate to procedure entry, start with an empty path
  – If a summary for that entry exits, use it
• At each exit
  – Store paths from corresponding call points as summary paths
  – When a new summary is added, propagate to the return node
Interprocedural Dataflow Analysis via CFL-Reachability

• Graph: Exploded control-flow graph

• $L$: $L(\text{unbalLeft})$
  – unbalLeft = valid

• Fact $d$ holds at $n$ iff there is an $L(\text{unbalLeft})$-path from $\langle \text{start}_{\text{main}}, \Lambda \rangle$ to $\langle n, d \rangle$
Asymptotic Running Time

• CFL-reachability
  – Exploded control-flow graph: $ND$ nodes
  – Running time: $O(N^3D^3)$
• Exploded control-flow graph → Special structure

Running time: $O(ED^3)$

Typically: $E \approx N$, hence $O(ED^3) \approx O(ND^3)$

“Gen/kill” problems: $O(ED)$
IDE

• Goes beyond IFDS problems
  – Can handle unbounded domains
• Requires special form of the domain
• Can be much more efficient than IFDS
Example Linear Constant Propagation

• Consider the constant propagation lattice

• The value of every variable $y$ at the program exit can be represented by:

$$y = \sqcup \{(a_x x + b_x) \mid x \in \text{Var}_* \} \sqcup c$$

$$a_x, c \in \mathbb{Z} \cup \{-, \top\} \quad b_x \in \mathbb{Z}$$

• Supports efficient composition and “functional” join
  – $[z := a \times y + b]$
  – What about $[z := x + y]$?
Linear constant propagation

Point-wise representation of environment transformers
IDE Analysis

• Point-wise representation closed under composition
• CFL-Reachability on the exploded graph
• Compose functions
declare x: integer
program main
begin
    call P(7)
    print (x) /* x is a constant here */
end

procedure P (value a: integer)
begin /* a is not a constant here */
    if a > 0 then
        a := a - 2
        call P (a)
        a := a + 2
    fi
    x := -2 * a + 5 /* x is not a constant here */
end
Costs

- $O(ED^3)$
- Class of value transformers $F \subseteq L \rightarrow L$
  - $id \in F$
  - Finite height
- Representation scheme with (efficient)
  - Application
  - Composition
  - Join
  - Equality
  - Storage
Conclusion

• Handling functions is crucial for abstract interpretation
• Virtual functions and exceptions complicate things
• But scalability is an issue
  – Small call strings
  – Small functional domains
  – Demand analysis
Challenges in Interprocedural Analysis

- Respect call-return mechanism
- Handling recursion
- Local variables
- Parameter passing mechanisms
- The called procedure is not always known
- The source code of the called procedure is not always available
A trivial treatment of procedure

• Analyze a single procedure
• After every call continue with conservative information
  – Global variables and local variables which “may be modified by the call” have unknown values
• Can be easily implemented
• Procedures can be written in different languages
• Procedure inline can help
Disadvantages of the trivial solution

• Modular (object oriented and functional) programming encourages small frequently called procedures
• Almost all information is lost
Bibliography

• **Textbook 2.5**


• **Two Approaches to interprocedural analysis by Micha Sharir and Amir Pnueli**

• **IDFS** Interprocedural Distributive Finite Subset Precise interprocedural dataflow analysis via graph reachability. *Reps, Horowitz, and Sagiv, POPL’ 95*

• **IDE** Interprocedural Distributive Environment Precise interprocedural dataflow analysis with applications to constant propagation. *Sagiv, Reps, Horowitz, and TCS’ 96*
A Semantics for Procedure Local Heaps and its Abstractions

Noam Rinetzky  Tel Aviv University
Jörg Bauer Universität des Saarlandes
Thomas Reps University of Wisconsin
Mooly Sagiv  Tel Aviv University
Reinhard Wilhelm Universität des Saarlandes
Motivation

• Interprocedural shape analysis
  – Conservative static pointer analysis
  – Heap intensive programs
    • Imperative programs with procedures
    • Recursive data structures

• Challenge
  – Destructive update
  – Localized effect of procedures
Main idea

• Local heaps

call p(x);
Main idea

- Local heaps
- Cutpoints

```
call p(x);
```
Main Results

• Concrete operational semantics
  – Large step
    • Functional analysis
  – Storeless
    • Shape abstractions
  – Local heap
  – Observationally equivalent to “standard” semantics
    • Java and “clean” C

• Abstractions
  – Shape analysis [Sagiv, Reps, Wilhelm, TOPLAS ’02]
  – May-alias [Deutsch, PLDI ‘94]
  – ...
Outline

• Motivating example
  – Local heaps
  – Cutpoints
• Why semantics
• Local heap storeless semantics
• Shape abstraction
static void main() {
    List x = reverse(p);
    List y = reverse(q);
    List z = reverse(x);
}

static List reverse(List t) {
    return r;
}
static void main() {
    List x = reverse(p);
    List y = reverse(q);
    List z = reverse(x);
}

static List reverse(List t) {
    return r;
}

Example
Example

```java
static void main() {
    List x = reverse(p);
    List y = reverse(q);
    List z = reverse(x);
}

static List reverse(List t) {
    return r;
}
```

```java
List x = reverse(p);
List y = reverse(q);
List z = reverse(x);
return r;
```
Cutpoints

- **Separating** objects
  - Not pointed-to by a parameter
Cutpoints

- **Separating** objects
  - Not pointed-to by a parameter

\[ \text{proc}(x) \]
Cutpoints

- **Separating** objects
  - Not pointed-to by a parameter

proc(x)  
Stack sharing  

proc(x)  
Heap sharing
Cutpoints

- **Separating** objects
  - Not pointed-to by a parameter
- **Capture external** sharing patterns

proc(x)

Stack sharing

Heap sharing
Example

```java
static List reverse(List t) {
    return r;
}

static void main() {
    List x = reverse(p);
    List y = reverse(q);
    List z = reverse(x);
}
```

```java
static List reverse(List t) {
    return r;
}
```

```java
r
```
Outline

✓ Motivating example
  • Why semantics
  • Local heap storeless semantics
  • Shape abstraction
Abstract Interpretation
[Cousot and Cousot, POPL ’77]
Introducing local heap semantics

Operational semantics

Local heap Operational semantics

Abstract transformer
Outline

✓ Motivating example
✓ Why semantics
  • Local heap storeless semantics
  • Shape abstraction
Programming model

• Single threaded
• Procedures
  ✓ Value parameters
  ✓ Recursion
• Heap
  ✓ Recursive data structures
  ✓ Destructive update
  ✗ No explicit addressing (&)
  ✗ No pointer arithmetic
Simplifying assumptions

• No primitive values (only references)
• No globals
• Formals not modified
Storeless semantics

• No addresses
• Memory state:
  – Object: $2^\text{Access paths}$
  – Heap: $2^\text{Object}$
• Alias analysis
Example

code:
```java
static void main() {
    List x = reverse(p);
    List y = reverse(q);
    List z = reverse(x);
}
```

diagram:
```
representing the function calls and list reversals
```
static void main() {
    List x = reverse(p);
    List y = reverse(q);
    List z = reverse(x);
}

static List reverse(List t) {
    return r;
}
Cutpoint labels

• Relate pre-state with post-state
• Additional roots
• Mark cutpoints at and throughout an invocation
• **Cutpoint label**: the set of access paths that point to a cutpoint
  - when the invoked procedure starts

\[ L \equiv \{t.n.n.n\} \]
Sharing patterns

• Cutpoint labels encode sharing patterns

L ≡ \{t.n.n.n.n\}
Observational equivalence

- $\sigma_L \in \Sigma_L$ (Local-heap Storeless Semantics)
- $\sigma_G \in \Sigma_G$ (Global-heap Store-based Semantics)

$\sigma_L$ and $\sigma_G$ observationally equivalent when for every access paths $AP_1, AP_2$

$$\llbracket AP_1 = AP_2 \rrbracket(\sigma_L) \iff \llbracket AP_1 = AP_2 \rrbracket(\sigma_G)$$
Main theorem: semantic equivalence

• $\sigma_L \in \Sigma_L$ (Local-heap Storeless Semantics)
• $\sigma_G \in \Sigma_G$ (Global-heap Store-based Semantics)
• $\sigma_L$ and $\sigma_G$ observationally equivalent

$$\langle st, \sigma_L \rangle \xrightarrow{\text{LSL}} \sigma'_L \Leftrightarrow \langle st, \sigma_G \rangle \xrightarrow{\text{GSB}} \sigma'_G$$

$\sigma'_L$ and $\sigma'_G$ are observationally equivalent
Corollaries

• Preservation of invariants
  – Assertions: $\text{AP}_1 = \text{AP}_2$

• Detection of memory leaks
Applications

• Develop new static analyses
  – Shape analysis
• Justify soundness of existing analyses
Related work

• **Storeless semantics**
  – Jonkers, Algorithmic Languages ‘81
  – Deutsch, ICCL ‘92
Shape abstraction

• Shape descriptors represent unbounded memory states
  – Conservatively
  – In a bounded way

• Two dimensions
  – Local heap (objects)
  – Sharing pattern (cutpoint labels)
A Shape abstraction

$L = \{t.n.n.n\}$
A Shape abstraction
A Shape abstraction
A Shape abstraction
A Shape abstraction

$L = \{ t, r.n.n.n \}$
A Shape abstraction

$L_1 = \{t.n.n.n\}$

$L_2 = \{g.n.n.n\}$

$L = *$
Cutpoint-Freedom
How to tabulate procedures?

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

main() {
  append(y,z);
}

append(List p, List q) {
  ...
}

Diagram:
How to handle sharing?

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

1st Example

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

2nd Example

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

The difference is highlighted in the 2nd example by the red circle around the variable $x$. In the 1st example, $x$ is not connected to the list, while in the 2nd example, $x$ is connected, indicating a different behavior in the context of list manipulation.
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
Cutpoint freedom

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

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

```
append(y,z)
```
Interprocedural shape analysis for cutpoint-free programs using 3-Valued Shape Analysis
Memory states: 2-Valued Logical Structure

• A memory state encodes a local heap
  – Local variables of the current procedure invocation
  – Relevant part of the heap
    • Relevant \( \equiv \) Reachable
Memory states

- Represented by first-order logical structures

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x(v)$</td>
<td>Variable $x$ points to $v$</td>
</tr>
<tr>
<td>$n(v_1, v_2)$</td>
<td>Field $n$ of object $v_1$ points to $v_2$</td>
</tr>
</tbody>
</table>
Memory states

- Represented by first-order logical structures

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x(v)$</td>
<td>Variable $x$ points to $v$</td>
</tr>
<tr>
<td>$n(v_1, v_2)$</td>
<td>Field $n$ of object $v_1$ points to $v_2$</td>
</tr>
</tbody>
</table>
Operational semantics

• Statements modify values of predicates
• Specified by predicate-update formulae
  – Formulae in FO-TC
1. Verify cutpoint freedom
2. Compute input
3. Execute callee
4. Combine output
Procedure call:
1. Verifying cutpoint-freedom

- 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
Procedure call:
1. Verifying cutpoint-freedom

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

(main’s locals: x,y,z,t)
Procedure call:
2. Computing the input local heap

- Retain only reachable objects
- Bind formal parameters
Procedure body: append(p,q)
Procedure call:
3. Combine output

Call state

Output state
Procedure call:
3. Combine output

Call state
Output state

Auxiliary predicates
- \( \text{inUc}(v) \)
- \( \text{inUx}(v) \)
Observational equivalence

• $\sigma_{CPF} \in \Sigma_{CPF}$ (Cutpoint free semantics)
• $\sigma_{GSB} \in \Sigma_{GSB}$ (Standard semantics)

$\sigma_{CPF}$ and $\sigma_{GSB}$ observationally equivalent when for every access paths $AP_1, AP_2$

$$\left[ AP_1 = AP_2 \right](\sigma_{CPF}) \Leftrightarrow \left[ AP_1 = AP_2 \right](\sigma_{GSB})$$
Observational equivalence

- For cutpoint free programs:
  - $\sigma_{\text{CPF}} \in \Sigma_{\text{CPF}}$ (Cutpoint free semantics)
  - $\sigma_{\text{GSB}} \in \Sigma_{\text{GSB}}$ (Standard semantics)
  - $\sigma_{\text{CPF}}$ and $\sigma_{\text{GSB}}$ observationally equivalent

- It holds that
  - $\langle st, \sigma_{\text{CPF}} \rangle \sim \sigma'_{\text{CPF}} \iff \langle st, \sigma_{\text{GSB}} \rangle \sim \sigma'_{\text{GSB}}$
  - $\sigma'_{\text{CPF}}$ and $\sigma'_{\text{GSB}}$ are observationally equivalent
Introducing local heap semantics

Operational semantics

Local heap Operational semantics

Abstract transformer
Shape abstraction

- Abstract memory states represent unbounded concrete memory states
  - Conservatively
  - In a bounded way
  - Using 3-valued logical structures
3-Valued logic

- $1 = \text{true}$
- $0 = \text{false}$
- $1/2 = \text{unknown}$
- A join semi-lattice, $0 \sqcup 1 = 1/2$
Canonical abstraction
Instrumentation predicates

- Record derived properties
- Refine the abstraction
  - Instrumentation principle [SRW, TOPLAS’02]
- Reachability is central!

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_x(v) )</td>
<td>( v ) is reachable from variable ( x )</td>
</tr>
<tr>
<td>( r_{obj}(v_1, v_2) )</td>
<td>( v_2 ) is reachable from ( v_1 )</td>
</tr>
<tr>
<td>ils(( v ))</td>
<td>( v ) is heap-shared</td>
</tr>
<tr>
<td>c(( v ))</td>
<td>( v ) resides on a cycle</td>
</tr>
</tbody>
</table>
Abstract memory states (with reachability)
The importance of reachability:
Call append(y,z)
Abstract semantics

- Conservatively apply statements on abstract memory states
  - Same formulae as in concrete semantics
  - Soundness guaranteed [SRW, TOPLAS’02]
Procedure calls

1. Verify cutpoint freedom
2. Compute input
3. Execute callee
4. Combine output

append(y,z)

append(p,q)

append body
Conservative verification of cutpoint-freedom

- Invoking append(y,z) in main
  - \( R_{\{y,z\}}(v) = \exists v_1: y(v_1) \land n^*(v_1, v) \lor \exists v_1: z(v_1) \land n^*(v_1, v) \)
  - \( \text{isCP}_{\text{main},\{y,z\}}(v) = R_{\{y,z\}}(v) \land (\neg y(v) \land \neg z(v_1)) \land (x(v) \lor t(v) \lor \exists v_1: \neg R_{\{y,z\}}(v_1) \land n(v_1, v)) \)
Interprocedural shape analysis

Tabulation exits
Interprocedural shape analysis

Analyze f

Tabulation exits

call f(x)
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

<table>
<thead>
<tr>
<th>Data structure</th>
<th>Verified properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Singly linked list</td>
<td>Cleanness, acyclicility</td>
</tr>
<tr>
<td>Sorting (of SLL)</td>
<td>+ Sortedness</td>
</tr>
<tr>
<td>Unshared binary trees</td>
<td>Cleaness, tree-ness</td>
</tr>
</tbody>
</table>
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]
Summary

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

- Properties proved
  - Absence of null dereferences
  - Listness preservation
  - API conformance
- Recursive $\approx$ Iterative
- Procedural abstraction
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
  – ...
Summary

• Operational semantics
  – Storeless
  – Local heap
  – Cutpoints
  – Equivalence theorem

• Applications
  – Shape analysis
  – May-alias analysis
Project

• 1-2 Students in a group
  – 3-4: Bigger projects
• Theoretical + Practical
• Your choice of topic
  – Contact me in 2 weeks
• Submission – 15/Sep
  – Code + Examples
  – Document
  – 20 minutes presentation
Past projects

- JavaScript Dominator Analysis
- Attribute Analysis for JavaScript
- Simple Pointer Analysis for C
- Adding program counters to Past Abstraction
- Verification of Asynchronous programs
- Verifying SDNs using TVLA
- Verifying independent accesses to arrays in GO
Past projects

• Detecting index out of bound errors in C programs
• Lattice-Based Semantics for Combinatorial Models Evolution