Shape Abstractions with Support for Sharing and Disjunctions

Huisong Li
Advised by: Xavier Rival

ENS, INRIA, CNRS, PSL*

March 8, 2018
Software is challenging

Software is extremely complex, huge and important

- military, medical, transportation, bank systems, ...
- hard to develop and maintain
- often buggy, e.g. a recent Mac os allows you to become a root user without a password
- testing and code review, useful, but cannot guarantee anything

We want to guarantee that:

- safe: absence of run time errors, especially for critical software
- secure: does not leak important information
- be functionally correct
Programs manipulating dynamic data structures are challenging

Dynamic data structures, e.g., linked list, binary search tree

- pointers as links
  - dereferencing of null, uninitialized, and dangling pointers
- dynamic memory allocation and deallocation
  - illegal free, memory leak
- structural properties have to be preserved
  - complex code
Formal verification

- prove a program satisfies certain properties using mathematics
- formal semantics + formal specification
describe programs and program properties in mathematical language

Automatic formal verification

- program algorithm

Sound

- the verification answers yes $\Rightarrow$ a program satisfies a specification

Complete

- a program satisfies a specification $\Rightarrow$ the verification answers yes

**undecidable problem: no complete, sound and automatic algorithm**
Conservative static analyses

Conservative static analyses aim at automatically verifying programs

- **sound + automatic + not complete**
- based on abstraction (over-approximation) approach

Abstract interpretation is a framework to design static analyses

- **abstract program properties**
  - e.g., intervals as abstraction of integers
- **abstract program operations**
  - e.g., \([m_1, m_2] + [n_1, n_2] = [m_1 + n_1, m_2 + n_2]\)
- **widening \(\triangledown\) for computing abstract loop invariants**

**abstract domain = abstractions + abstract operations + widening**

Existing analyses

- numeric analysis
- memory analysis
- ...
Points-to abstraction

Concrete memory:

![Concrete Memory Diagram]

Points-to abstraction:

- abstract concrete addresses with symbolic variables
  \[ v_0 \rightarrow \alpha_0 \quad v_1 \rightarrow \alpha_1 \]
  \[ v_2 \rightarrow \alpha_2 \quad v_3 \rightarrow \alpha_3 \]
- abstract memory cells with points-to predicates
  \[ \alpha_0 \leftarrow \alpha_1 \land \alpha_1 \leftarrow \alpha_2 \land \alpha_2 \leftarrow \alpha_3 \land \alpha_3 \rightarrow 0 \]

Limitation:

- hard to express disjointness of memory cells to support strong update
  \[ \alpha_0 \leftarrow \alpha_1 \text{ and } \alpha_1 \leftarrow \alpha_2 \] describe different memory cells
Separating conjunction

Concrete memory:

\[ V_0 \rightarrow V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow 0 \]

Points-to abstraction with separating conjunction (\( \ast \)) (John C. Reynolds’02):

- separating conjunction (\( \ast \)) allows us to express disjointness
  \[ \alpha_0 \mapsto \alpha_1 \ast \alpha_1 \mapsto \alpha_2 \ast \alpha_2 \mapsto \alpha_3 \ast \alpha_3 \mapsto 0 \]
  \[ \implies \forall 0 \leq i, j \leq 3, i \neq j \implies \alpha_i \neq \alpha_j \]
- separating conjunction (\( \ast \)) enables local reasoning

\[
\frac{[\phi] P [\phi']} {\left[ \phi \ast \psi \right] P \left[ \phi' \ast \psi \right]}
\]
Summarization of unbounded inductive data structures

Concrete memory:

\[ v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow 0 \]

Abstraction with summarization:

- **inductive definitions** to precisely describe dynamic data structures
  \[ \alpha \cdot \text{list} ::= \alpha = 0 \lor \alpha \neq 0 \land \exists \beta. \alpha \cdot n \mapsto \beta \ast \beta \cdot \text{list} \]

- **inductive predicates** as instances of inductive definitions:
  \[ \alpha_0 \cdot n \mapsto \alpha_1 \ast \alpha_1 \cdot \text{list} \]

Abstract state (formula)

Abstract state (graph)
Example: Forward analysis of a list traversal program

1. \texttt{list\* c = h;} \\
2. \texttt{while (c \neq \texttt{NULL})} \\
3. \texttt{c = c -> n;} \\

**Forward analysis:**
- start from a given abstract pre-condition
- automatically compute an abstract post-condition
Example: Forward analysis of a list traversal program

1. `list* c = h;`
2. `while(c != NULL)`
   
   ![Diagram](h, c) → `list` → $\alpha_0 \neq 0$

3. `c = c -> n;`

**abstract state: shape abstraction $\times$ numerical abstraction**
Example: Forward analysis of a list traversal program

```
1 list* c = h;
2 while (c != NULL)
   h, c
   list
   α₀
   α₀ ≠ 0
3 c = c -> n;
```

- Unfolding the inductive predicate
Example: Forward analysis of a list traversal program

1. \texttt{list* c = h;}
2. \texttt{while (c \neq NULL)}
   \begin{align*}
   \alpha_0 & \neq 0 \\
   \alpha_0 & \neq 0 \land \alpha_0 = 0
   \end{align*}
3. \texttt{c = c -> n;}

- unfolding generates case splits
- the second case is unsatisfiable
Example: Forward analysis of a list traversal program

1. list* c = h;

   ![Diagram of abstract states](image1)

2. while (c != NULL)

3. c = c -> n;

   ![Diagram of abstract states](image2)

- widening $\nabla$ abstract states to compute loop invariant
- widening folds back unfolded predicates
Example: Forward analysis of a list traversal program

1. list* c = h;
2. while(c != NULL)
   \[\alpha_0 \quad \text{list} \quad \alpha_1 \quad \text{list} \quad | \alpha_1 \neq 0\]
3. c = c -> n;

Abstract loop invariant
Limitation: sharing is hard to express

List:
- recursive data structure
- no sharing (a node can only be dereferenced by a pointer)

\[ \alpha \rightarrow \text{list} \iff \alpha = 0 \lor \alpha \rightarrow \text{list} \]

Adjacency lists representing directed graphs:
- a recursive data structure (list of lists)
- unbounded sharing
  a node can be dereferenced by many edge pointers
- inductive definition cannot capture unbounded sharing
Limitation: disjunctions are necessary but costly

Without merging, disjuncts number grows exponentially in disjunctive forward analysis

For scalability, disjuncts number should be kept small
Fewer disjuncts means lower analysis cost
But merging disjuncts may lose precision

Deciding how to merge disjuncts without losing too much precision is critical
We study abstractions to improve expressiveness and scalability:

For sharing problem:
- separation-logic based shape analysis for unstructured sharing

For disjunction control problem:
- semantic-directed clumping of disjunctive abstract states

Implemented and evaluated within the MemCAD static analyzer
Table of Contents

1 Introduction

2 Shape analysis for unstructured sharing
   • Abstract states
   • Analysis algorithm
   • Experimental evaluation

3 Semantic-directed clumping of disjunctive abstract states
   • Silhouettes
   • Silhouette guided clumping and joining
   • Experimental evaluation

4 Conclusion and future directions
Graph random path traversal

typedef struct node {
    struct node * next;
    int id;
    struct edge * edges;
} node;

typedef struct edge {
    struct node * dest;
    struct edge * next;
} edge;

0 node* c = h;
   // start at the first node
1 while(c != NULL){
2    edge* s = c -> edges;
3       ......
4    c = s -> dest;
5    n = c -> id;
6       // random visit a successor
7 }

Analysis goals:

- preservation of structural properties of adjacency lists
- absence of memory errors, e.g., dereferencing of null, uninitialized, and dangling pointers
Towards precise summarization of adjacency lists

Concrete adjacency list:

Inductive definition for adjacency lists following list of list structure:

- nodes can only be dereferenced from `next` field of a previous node
- information about edge pointers is missing
Towards precise summarization of adjacency lists

Concrete adjacency list:

Inductive definition for adjacency lists following list of list structure:

- extend the inductive definition with set parameters
- rely on set predicates to precisely capture properties of edge pointers
Towards precise summarization of adjacency lists

Concrete adjacency list:

Summarize the set of node addresses by a set variable $\mathcal{F}$:
Towards precise summarization of adjacency lists

Concrete adjacency list:

Enforce the property that each edge of a node in a set $\mathcal{E}$:

$$\alpha \in \text{edges}(\mathcal{E}) \iff \alpha = 0 \lor \exists \beta \in \mathcal{E}$$
Towards precise summarization of adjacency lists

Concrete adjacency list:

Enforce all the edges of any node in the same set:
Inductive definitions of adjacency lists

Node list definition:

\[
\text{nodes}(E, \mathcal{F}) \iff \alpha = 0 \land \mathcal{F} = \emptyset
\]

\[
\text{edges}(E) \iff \alpha = 0 \land \beta \in E
\]

Edge list definition:

\[
\text{nodes}(E, \mathcal{F}') \land \mathcal{F} = \{\alpha\} \cup \mathcal{F}'
\]

\[
\text{edges}(E) \land \beta \in E
\]
Abstract pre-condition of graph random path traversal

**Precondition:** $h$ points-to a valid adjacency list

$h$ nodes$(E, F)$

\[ F : \text{set of node addresses} \]
\[ E : \text{set of edge pointers} \]

```
0    node* c = h;
     // start at the first node
1   while(c != NULL){
2    edge* s = c -> edges;
       ......
3    c = s -> d;
4    n = c -> id;
     // random visit a successor
}
```
Abstract pre-condition of graph random path traversal

**Precondition**: $h$ points-to a valid adjacency list

$\alpha_0$ nodes($E$, $F$) \[ E \subseteq F \]

\[ \{ F : \text{set of node addresses} \]  
\[ E : \text{set of edge pointers} \]

0 node* c = h;  
    // start at the first node  
1 while(c != NULL){  
2    edge* s = c -> edges;  
    ......  
3    c = s -> d;  
4    n = c -> id;  
    // random visit a successor  
}
Abstract states

Combined abstract states: \( \overline{m} ::= (\overline{g}, \overline{n}, \overline{s}) \)

\( \overline{g} \): shape abstraction
- separating conjunction of points-to, inductive and segment predicates
- parameterized by inductive definitions with set parameters

\( \overline{n} \): numerical abstraction
- abstracts numerical constraints, e.g., \( \alpha \neq 0, \alpha = \beta \)

\( \overline{s} \): set abstraction
- abstracts set constraints, e.g., \( E = F_1 \cup F_2 \)
Set domains

Set domains automate set predicates reasoning

- **a common interface for set abstract domains**
  used by program analysis

- **linear based set domain**
  focuses on reasoning about linear partitions of sets
  
  \[ E_0 = \{\alpha_0, \ldots, \alpha_k\} \uplus E_1 \uplus \ldots \uplus E_m \]
  
  \[ E \subseteq \mathcal{F}, \alpha \in E, E = \mathcal{F} \]

- **BDD-based set domain**
  encode set predicates into boolean algebraic forms
  represent boolean algebraic forms as binary decision diagrams

**Arlen Cox, Bor-Yuh Evan Chang, Huisong Li, Xavier Rival**

Abstract Domains and Solvers for Sets Reasoning (LPAR’15)
Concretization

Concretization $\gamma(\bar{g}, \bar{n}, \bar{s})$:

- defines the meaning for abstract states in concrete states
- allows us to prove the soundness of the analysis

Example:

- concretization of symbolic variables and set variables
  
  $\alpha_4 \mapsto a_4 \quad \alpha_5 \mapsto a_5 \quad \alpha' \mapsto a_1 \quad \mathcal{E} \mapsto \{a_0, a_1, a_2, a_3\}$
Concretization \( \gamma(\bar{g}, \bar{n}, \bar{s}) \):

- defines the meaning for abstract states in concrete states
- allows us to prove the soundness of the analysis

Example:

- points-to edges are concretized into concrete memory cells
Concretization $\gamma(g, n, s)$:
- defines the meaning for abstract states in concrete states
- allows us to prove the soundness of the analysis

Example:

- inductive edges are concretized with unfolding
# Table of Contents

1. **Introduction**

2. **Shape analysis for unstructured sharing**
   - Abstract states
   - Analysis algorithm
   - Experimental evaluation

3. **Semantic-directed clumping of disjunctive abstract states**
   - Silhouettes
   - Silhouette guided clumping and joining
   - Experimental evaluation

4. **Conclusion and future directions**
Analysis principle

Extend existing abstract operations:
- abstract read, write
- unfolding

Support non-local unfolding
- enables dereferencing through unbounded pointers

Extend folding operations to synthesize set parameters of summary predicates:
- joining $\sqcup$
- widening $\triangledown$
- entailment checking $\sqsubseteq$
Unfolding

0  node* c = h;
    // start at the first node
1  while(c != NULL){
  h, c
  nodes(E, F)
  \alpha_0 \neq 0
  \alpha_0 \neq 0

  E \subseteq F

2  edge* s = c -> edges;
    ......
3  c = s -> d;
4  n = c -> id;
    // random visit a successor
}

Read a field in a summarized region
Unfolding

0 node* c = h;
   // start at the first node
1 while(c != NULL){
   nodes(E, F_1)
   h, c
   id
   α₀
   edges
   s
   edges(E)
   α₀ ≠ 0
   E ⊆ F
   F = {α₀} ∪ F_1

2 edge* s = c -> edges; ....
3 c = s -> d;
4 n = c -> id;
   // random visit a successor
}
Non-local unfolding

0  node* c = h;
1  while (c != NULL) {
2    edge* s = c -> edges;
3    c = s -> d;
4    n = c -> id;
}

- dereferencing graph nodes through edge pointers
- does not follow the inductive structure
Non-local unfolding

Refine shape abstraction according to set predicate of the form

$$\alpha \in \{\alpha_0, \ldots, \alpha_k\} \cup E_0 \cup \ldots \cup E_l$$

by localizing $\alpha$ in the shape abstraction:

$$\alpha = \alpha_0$$
$$\lor \ldots$$
$$\lor \alpha = \alpha_k$$
$$\lor \alpha \in E_0$$
$$\lor \ldots$$
$$\lor \alpha \in E_l$$
Non-local unfolding

Abstract state:

Perform non-local unfolding at $\alpha_3$ according to the set abstraction constraint $\alpha_3 \in \{\alpha_0\} \cup \mathcal{F}_1$:

- $\alpha_3 = \alpha_0$
- $\alpha_3 \in \mathcal{F}_1$
Non-local unfolding

Localize $\alpha_3$ to $\alpha_0$:

$$\text{read}(\alpha_3 \cdot \text{id}) = \text{read}(\alpha_0 \cdot \text{id}) = \beta$$
Localize $\alpha_3$ to $\mathcal{F}_1$ according to properties of the set parameter:

- $\mathcal{F}_1$ denotes the set of the addresses of the nodes described by the inductive edge.
- $\alpha_3$ is a node address summarized by the inductive edge.

$$\mathcal{F}_1 \text{ denotes the set of the addresses of the nodes described by the inductive edge}$$

$$\alpha_3 \text{ is a node address summarized by the inductive edge}$$
Non-local unfolding

Localize $\alpha_3$ to $\mathcal{F}_1$ according to properties of the set parameter:

- $\mathcal{F}_1$ denotes the set of the addresses of the nodes described by the inductive edge
- $\alpha_3$ is a node address summarized by the inductive edge

\[
\begin{align*}
\mathcal{F}_1 & \text{ denotes the set of the addresses of the nodes described by the inductive edge} \\
\alpha_3 & \text{ is a node address summarized by the inductive edge}
\end{align*}
\]

Splitting the inductive edge into a segment and an inductive edge
Joining

Joining of two abstract states: \((\overline{g}_l, \overline{s}_l) \sqcup (\overline{g}_r, \overline{s}_r)\)

- compute a sound and common weaker abstraction \((\overline{g}_o, \overline{s}_o)\)
  \[
  \gamma(\overline{g}_l, \overline{s}_l) \subseteq \gamma(\overline{g}_o, \overline{s}_o) \\
  \gamma(\overline{g}_r, \overline{s}_r) \subseteq \gamma(\overline{g}_o, \overline{s}_o)
  \]

- joining of abstract shapes based on graph rewriting rules
Graph rewriting rules for shape joining

Weakening identical predicates:

\[(\overline{g}, \overline{s}_l) \sqcup_{\overline{\mathcal{G}}} (\overline{g}, \overline{s}_r) = \overline{g}\]

Weakening guided by existing summary predicates:

\[
\begin{align*}
\overline{s}_l' & \text{ instantiates } \mathcal{E} \text{ in } \overline{s}_l \\
\overline{s}_l' & \vdash \overline{g}_l \sqsubseteq \alpha \cdot \text{ind}(\mathcal{E}) \\
(\overline{g}_l, \overline{s}_l) & \sqcup_{\overline{\mathcal{G}}} (\alpha \cdot \text{ind}(\mathcal{E}), \overline{s}_r) = \alpha \cdot \text{ind}(\mathcal{E})
\end{align*}
\]

- instantiate \(\mathcal{E}\): resolve the meaning of set parameter \(\mathcal{E}\) in the left side abstraction
- currently done by using constraints from the inclusion check
- has to be sound and precise
- may have non unique solutions

Weakening both sides into new summary predicates:

- instantiation of set parameters
A joining example

\[ \overline{g_l}: \]
\[ \alpha_0 \xrightarrow{h} \text{nodes}(\mathcal{E}, \mathcal{F}) \]

\[ \overline{g_r}: \]
\[ \alpha_0 \xrightarrow{h} \text{id} \]
\[ \text{next} \xrightarrow{h} \text{nodes}(\mathcal{E}, \mathcal{F}_1) \]
\[ \text{edges} \xrightarrow{h} \text{edges}(\mathcal{E}) \]

\[ \overline{g_o}: \]
A joining example

- \( g_l: \)  
  - \( h \)  
  - \( \alpha_0 \)  
  - \( \text{nodes}(E, \mathcal{F}) \)

- \( g_r: \)  
  - \( h \)  
  - \( \alpha_0 \)  
  - \text{id}  
  - \( \text{edges}(E) \)

- \( g_O: \)  
  - \( h \)  
  - \( \alpha_0 \)  
  - \( \text{nodes}(E, \mathcal{F}) \)

\[ \mathcal{F} = \{ \alpha_0 \} \cup \mathcal{F}_1 \]

- list of nodes and their outgoing edges
- list of nodes and their outgoing edges

- list of edges of node \( \alpha_0 \)

- \( \&h \)  

- weaken the right side abstraction into the inductive edge
- instantiate set parameter \( \mathcal{F} \) according to the weakening process
Graph random path traversal

0 \textbf{node* c = h;}
   // start at the first node
1 \textbf{while}(c != NULL){

2 \textbf{edge* s = c -> edges;}
   .......
3 \textbf{c = s -> dest;}
4 \textbf{n = c -> id;}
   // random visit a successor
}

\textbf{h nodes}(E, F) \quad \textbf{c nodes}(E, X_0) \quad \textbf{c nodes}(E, X_1) \quad E \subseteq F \land F = X_0 \cup X_1
Experiment method and goals

Extend the MemCAD static analyzer

- extend inductive definitions with set parameters
- extend the memory abstract domain to take a set abstract domain as a parameter

Assess goals

- structure preservation of data structures with unbounded sharing can be proved
- the efficiency of memory abstract domain is preserved
**Experiment results and conclusion**

<table>
<thead>
<tr>
<th>Description</th>
<th>LOCs</th>
<th>“BDD” time (ms)</th>
<th>“LIN” time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Shape</td>
<td>Set</td>
</tr>
<tr>
<td>Node: add</td>
<td>27</td>
<td>44</td>
<td>0.3</td>
</tr>
<tr>
<td>Edge: add</td>
<td>26</td>
<td>31</td>
<td>0.2</td>
</tr>
<tr>
<td>Edge: delete</td>
<td>22</td>
<td>45</td>
<td>0.4</td>
</tr>
<tr>
<td>Node list traversal</td>
<td>25</td>
<td>117</td>
<td>1.5</td>
</tr>
<tr>
<td>Edge list iteration +</td>
<td>34</td>
<td>332</td>
<td>2.7</td>
</tr>
<tr>
<td>dest. read</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graph path: deterministic</td>
<td>31</td>
<td>360</td>
<td>2.7</td>
</tr>
<tr>
<td>Graph path: random</td>
<td>43</td>
<td>765</td>
<td>7.1</td>
</tr>
</tbody>
</table>
Experiment results and conclusion

- successfully establishes memory safety and structural preservation
- analysis time spent in the shape domain in line with that usually observed in MemCAD
- BDD-based set domain is less efficient than linear set domain

Huisong Li, Xavier Rival, Bor-Yuh Evan Chang

Shape Analysis for Unstructured Sharing (SAS’15)
Table of Contents

1 Introduction

2 Shape analysis for unstructured sharing
   - Abstract states
   - Analysis algorithm
   - Experimental evaluation

3 Semantic-directed clumping of disjunctive abstract states
   - Silhouettes
   - Silhouette guided clumping and joining
   - Experimental evaluation

4 Conclusion and future directions
Disjunction is necessary

Concrete memory:

```
| h | . . | . . | . . | 0 |
```

Abstract state:

```
| h | list |
```

```
search_min_max()

min = max = c = h;
while(c! = NULL){
    if(c -> d < min -> d) min = c;
    if(c -> d > max -> d) max = c;
    c = c -> n;
}
```

Concrete memories:

```
| h | min | max | . . | 0 |
| h | max | min | . . | 0 |
```

Disjunctive abstract post state:

```
| h | list | min | list | max | list |
```

| h | list | max | list | min | list |

Huisong Li
Sharing & Disjunctions
March 8, 2018
32 / 51
Existing techniques to deal with disjunctions

**Disjunctive completion** (Cousot & Cousot ’79):
effectively use $\mathcal{P}(A)$, i.e. allow all disjunctive abstractions

- only works for finite abstract domain, less expressive
- often very expensive

$A$: analysis domain
Existing techniques to deal with disjunctions

**Disjunctive completion** (Cousot & Cousot ’79):

- effectively use \( \mathcal{P}(\mathcal{A}) \), i.e. allow all disjunctive abstractions

**Canonicalization**: (Sagiv & Reps & Wilhelm ’02, Distefano & O’Hearn & Yang ’06)

- first, use a sound normalization \( \Phi : \mathcal{A} \rightarrow \mathcal{A}' \), where \( \mathcal{A}' \) is finite
- then, use \( \mathcal{P}(\mathcal{A}') \) at widening to compute loop invariants

- \( \mathcal{A} \) may be infinite, more expressive
- expressiveness is restricted by \( \mathcal{A}' \)
- can be very expensive

\( \mathcal{A} \): analysis domain
\( \mathcal{A}' \subset \mathcal{A} \): finite domain
Existing techniques to deal with disjunctions

**Disjunctive completion** (Cousot & Cousot’79):
- effectively use $\mathcal{P}(A)$, i.e. allow all disjunctive abstractions

**Canonicalization**: (Sagiv & Reps & Wilhelm’02, Distefano & O’Hearn & Yang’06)
- first, use a sound normalization $\Phi : A \rightarrow A'$, where $A'$ is finite
- then, use $\mathcal{P}(A')$ at widening to compute loop invariants

**Partitioning approach**: (Cousot & Cousot’92, Handjieva & Tzolovski’98, Rival & Mauborgne’07)
- effectively use a lattice of the form $B \rightarrow A$, with finite $B$
  - partition criteria include control flow, context, etc
  - such criteria tend to not work effectively for shape analysis

$A$: analysis domain
$B$: partition criterion
Existing techniques to deal with disjunctions

**Disjunctive completion** (Cousot&Cousot’79):
effectively use $\mathcal{P}(\mathcal{A})$, i.e. allow all disjunctive abstractions

**Canonicalization**: (Sagiv&Reps&Wilhelm’02, Distefano&O’Hearn&Yang’06)
first, use a sound normalization $\Phi : \mathcal{A} \rightarrow \mathcal{A}'$, where $\mathcal{A}'$ is finite
then, use $\mathcal{P}(\mathcal{A}')$ at widening to compute loop invariants

**Partitioning approach** (Cousot&Cousot’92, Handjieva&Tzolovski’98, Rival&Mauborgne’07)
effectively use a lattice of the form $\mathcal{B} \rightarrow \mathcal{A}$, with finite $\mathcal{B}$

We need a semantic technique to improve disjunction handling
Precision loss in join

Abstract states:

\( m_0 \)

\[
\begin{array}{c}
\text{h} \\
\text{list} \\
\text{min list} \\
\text{max list}
\end{array}
\]

\( m_1 \)

\[
\begin{array}{c}
\text{h} \\
\text{list} \\
\text{max list} \\
\text{min list}
\end{array}
\]

Imprecise upper bounds:

\[
\begin{array}{c}
\text{h} \\
\text{list} \\
\text{min list}
\end{array}
\]

\[
\begin{array}{c}
\text{h} \\
\text{list} \\
\text{max list}
\end{array}
\]

pointer \( \text{max} \) is lost

pointer \( \text{min} \) is lost

Abstract states \( m_0 \) and \( m_1 \) have several incomparable, imprecise upper bounds, but no precise least upper bound

Joining them will lose precision
Precision loss in join

Abstract states:

$\overline{m}_0$

$\overline{m}_1$

Imprecise upper bounds:

Observation

Pointers’ order in data structures has a big impact on join precision. When pointers are in different orders, join tends to be very imprecise as no abstract state can preserve different pointers’ orders.
Precision loss in join

Abstract states:

\[ \overline{m}_0 \]

\[ \begin{array}{c}
  h \\
  \text{list} \\
  \text{min} \\
  \text{list} \\
  \text{max} \\
  \text{list}
\end{array} \rightarrow

\[ \overline{m}_1 \]

\[ \begin{array}{c}
  h \\
  \text{list} \\
  \text{max} \\
  \text{list} \\
  \text{min} \\
  \text{list}
\end{array} \rightarrow

Imprecise upper bounds:

\[ \begin{array}{c}
  h \\
  \text{list} \\
  \text{min} \\
  \text{list}
\end{array} \rightarrow

pointer max is lost

\[ \begin{array}{c}
  h \\
  \text{list} \\
  \text{max} \\
  \text{list}
\end{array} \rightarrow

pointer min is lost

To quickly identify imprecise joins, we need:

- a coarse abstraction that can capture pointers’ orders
- a relation of the coarse abstraction that can characterize imprecise joins
Our contribution: silhouette abstraction

- silhouette abstraction (abstraction of abstract states)
- silhouette guided clumping
  rely on silhouette to quickly decide whether disjuncts can be joined precisely
- silhouette guided joining
  rely on silhouette to compute a precise upper bound
Silhouette

**Silhouette graph**

- **Nodes:** pointer values
- **Edges:** access path strings over fields (reachability)

\[ e ::= \epsilon \mid f \mid (f_0 + \ldots + f_n)^* \mid e \cdot e \]

**Silhouette is an abstraction of abstract states.**

Concrete memory:

<table>
<thead>
<tr>
<th>h</th>
<th>min</th>
<th>max</th>
</tr>
</thead>
</table>

Abstract state:

- red edges are abstracted away.
- green edges are abstracted by access paths.
- the `list` predicate is abstracted by access path `n^*`.

Silhouette:

<table>
<thead>
<tr>
<th>h</th>
<th>n</th>
<th>min</th>
<th>n</th>
<th>d</th>
<th>list</th>
<th>max</th>
</tr>
</thead>
</table>

- All the red edges are abstracted away.
- All the green edges are abstracted by access paths.
- The `list` predicate is abstracted by access path `n^*`. 
Silhouette-based weak abstract entailment check

Entailment check algorithms

\[ \sqsubseteq_{\mathcal{M}_\omega} : \text{decide inclusion of abstract states} \] (complex rewriting rules)

\[ \sqsubseteq_{\mathcal{G}} : \text{decide inclusion of silhouettes} \] (classical inclusion of constraints)

- Silhouette entailment check \( \sqsubseteq_{\mathcal{G}} \) offers a weak characterization for abstract states entailment check \( \sqsubseteq_{\mathcal{M}_\omega} \)

\[
\begin{align*}
\sqsubseteq_{\mathcal{M}_\omega} (\overline{m}_0, \overline{m}_1) &= \text{true} \implies \sqsubseteq_{\mathcal{G}} (\text{sil}(\overline{m}_0), \text{sil}(\overline{m}_1)) = \text{true} \\
\sqsubseteq_{\mathcal{G}} (\text{sil}(\overline{m}_0), \text{sil}(\overline{m}_1)) &= \text{false} \implies \sqsubseteq_{\mathcal{M}_\omega} (\overline{m}_0, \overline{m}_1) = \text{false}
\end{align*}
\]

- Silhouette entailment check \( \sqsubseteq_{\mathcal{G}} \) is much cheaper

Use silhouette entailment check to decide quickly when abstract states entailment does not hold
Silhouette-based weak abstract entailment check

Entailment check algorithms

\(\sqsubseteq_{\mathcal{M}_\omega}\): decide inclusion of abstract states (complex rewriting rules)

\(\sqsubseteq_{\mathcal{G}}\): decide inclusion of silhouettes (classical inclusion of constraints)

Use silhouette entailment check to decide quickly when abstract states entailment does not hold
Silhouette join

- Offers a weak, but precise characterization for abstract states join
  - set up the basis for clumping
- Much cheaper than abstract states join
- Simply replace access paths with an approximating reg-exp

Abstract states and their silhouettes:

\[ \overline{m}_0: \quad \text{sil}(\overline{m}_0): \]

\[ \overline{m}_1: \quad \text{sil}(\overline{m}_1): \]

The silhouettes of \( \overline{m}_0 \) and \( \overline{m}_1 \) cannot be joined precisely.
Abstract states \( \overline{m}_0 \) and \( \overline{m}_1 \) cannot also be joined precisely.
Silhouette join

- Offers a weak, but precise characterization for abstract states join
  - set up the basis for clumping
- Much cheaper than abstract states join
- Simply replace access paths with an approximating reg-exp

Abstract states and their silhouettes:

\[
\begin{align*}
  \text{\text{m}_2}: & \quad h, \min \quad \text{list} \quad \max \quad \text{list} \\
  \text{\text{sil} } (\text{\text{m}_2}): & \quad h, \min \quad n^* \quad \max \\
  \text{\text{m}_3}: & \quad h \quad \text{list} \quad \min, \max \quad \text{list} \\
  \text{\text{sil} } (\text{\text{m}_3}): & \quad h \quad n^* \quad \min, \max
\end{align*}
\]

The silhouettes of \text{\text{m}_2} and \text{\text{m}_3} can be joined precisely.

Abstract states \text{\text{m}_2} and \text{\text{m}_3} can also be joined precisely.
Table of Contents

1 Introduction

2 Shape analysis for unstructured sharing
   - Abstract states
   - Analysis algorithm
   - Experimental evaluation

3 Semantic-directed clumping of disjunctive abstract states
   - Silhouettes
   - Silhouette guided clumping and joining
   - Experimental evaluation

4 Conclusion and future directions
Silhouette guided clumping

Algorithm: group silhouettes based on an equivalence clumping relation which captures precise silhouette join (cheap to compute)

\[
\bar{m}_0 \lor \bar{m}_1 \lor \bar{m}_2 \lor \bar{m}_3
\]

(1) sil

\[
\bar{s}_0 \bar{s}_1 \bar{s}_2 \bar{s}_3
\]

(2) grouping sil

\[
\text{join}_M(\bar{m}_0, \bar{m}_1, \bar{m}_2) \lor \bar{m}_3
\]

(3) join

Silhouette association relation \(\Join\):
Let \(\bar{s}_0, \bar{s}_1\) be two silhouettes with the same set of nodes \(N\). We write \(\bar{s}_0 \Join \bar{s}_1\) if and only if there exist \(N_0, N_1\) such that \(N = N_0 \cup N_1\) and:

\[
\bar{s}_0' = \bar{s}_0'_{|N_0} \cup \bar{s}_0'_{|N_1} \land \bar{s}_0'_{|N_0} \sqsubseteq \bar{s}_1'_{|N_0} \\
\land \bar{s}_1' = \bar{s}_1'_{|N_0} \cup \bar{s}_1'_{|N_1} \land \bar{s}_1'_{|N_1} \sqsubseteq \bar{s}_0'_{|N_1}
\]

Characterizes precise joins that can be computed based on weakening rules guided by existing predicates
**Algorithm:** group silhouettes based on an equivalence clumping relation which captures precise silhouette join (cheap to compute)

\[
\overline{m}_0 \lor \overline{m}_1 \lor \overline{m}_2 \lor \overline{m}_3
\]

\( (1) \) sil

\[
\overline{s}_0 \quad \overline{s}_1 \quad \overline{s}_2 \quad \overline{s}_3
\]

\( (2) \) grouping sil

\[
\text{join}_M(\overline{m}_0, \overline{m}_1, \overline{m}_2) \lor \overline{m}_3
\]

\( (3) \) join\(_M\)

Silhouette groups:
\{\(\overline{s}_0, \overline{s}_1, \overline{s}_2\)\}, \{\(\overline{s}_3\)\}
Silhouette guided clumping

Algorithm: group silhouettes based on an equivalence clumping relation which captures precise silhouette join (cheap to compute)

(1) sil

(2) grouping sil

Clumping result: \( \text{join}_{\mathcal{M}}(\overline{m}_0, \overline{m}_1, \overline{m}_2) \lor \overline{m}_3 \)
Abstract states join without silhouette

Join abstract states $\text{join}_M(\overline{m}_0, \overline{m}_1)$:

- Compute an over-approximation of $\overline{m}_0, \overline{m}_1$.
- Existing: often rely on syntactic based rewriting rules.
  Different orders of rewriting rules produce different results.

Abstract states:

Join results (rewriting with different orders):

Imprecise  Imprecise  Imprecise  Precise
Silhouette guided abstract states join

Silhouette join guides abstract states join to be precise.
select which rewriting rules to apply.
help to synthesize inductive predicates.

Silhouette join is often an abstraction of precise abstract states join.

Abstract states and silhouette join:

Join results (rewriting with different orders):

Imprecise    Imprecise    Imprecise    Precise
Table of Contents

1 Introduction

2 Shape analysis for unstructured sharing
   - Abstract states
   - Analysis algorithm
   - Experimental evaluation

3 Semantic-directed clumping of disjunctive abstract states
   - Silhouettes
   - Silhouette guided clumping and joining
   - Experimental evaluation

4 Conclusion and future directions
Experimental evaluation

Evaluation goals:

- Clumping with guided joining effectively avoid precision loss.
- Clumping computation has reasonable overhead.
- Clumping limits disjunctive explosion.

26 Benchmarks (varies in both structures and implementations)

- GDSL: binary search tree, list (insert, delete, ...)
- BSD: red-black tree, splay tree (insert, delete, ...)
- JSW: AVL tree (insert, ...)
- ......
Clumping improves precision

**Several strategies:** (MemCAD baseline: widening to one disjunct)

<table>
<thead>
<tr>
<th>Guided joining</th>
<th>Clumping</th>
<th>Canonicalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>ClumpG</td>
<td>CanonG</td>
</tr>
<tr>
<td>N</td>
<td>Clump</td>
<td>Canon</td>
</tr>
</tbody>
</table>

Number of verified benchmarks using each strategy:

- Clumping with guided joining improves precision.
- Both clumping and guided joining have an impact.
Clumping has low overhead

Percentage of the analysis time spent on clumping, abstract states join, and the others:

- Silhouette computation + silhouette join = a few percent of the analysis time
- Abstract states join is very expensive analysis operation.
Clumping limits disjunctive explosion

- **Path**: the number of acyclic control-flow paths
- **Fix-disj**: maximum disjuncts number of loop invariants
- **Max-disj**: maximum disjuncts number at any program point
- **Post-disj**: disjuncts number at program exit

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Path</th>
<th>Fix-disj</th>
<th>Max-disj</th>
<th>Post-disj</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDSL (Binary tree)</td>
<td>insert</td>
<td>7680</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>delete</td>
<td>23040</td>
<td>1</td>
<td>69</td>
</tr>
<tr>
<td>BSD (splay tree)</td>
<td>delete</td>
<td>448</td>
<td>3</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>insert</td>
<td>43</td>
<td>3</td>
<td>42</td>
</tr>
<tr>
<td>BSD (red-black tree)</td>
<td>insert</td>
<td>3036</td>
<td>3</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>delete</td>
<td>1.e + 8</td>
<td>3</td>
<td>108</td>
</tr>
<tr>
<td>JSW (avl-tree)</td>
<td>insert</td>
<td>1.e + 8</td>
<td>3</td>
<td>120</td>
</tr>
</tbody>
</table>

Disjunction size does not explode exponentially when analyzing series of basic operations (insert / search / ...)
Clumping limits disjunctive explosion

- **Path**: the number of acyclic control-flow paths
- **Fix-disj**: maximum disjuncts number of loop invariants
- **Max-disj**: maximum disjuncts number at any program point
- **Post-disj**: disjuncts number at program exit

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Path</th>
<th>Fix-disj</th>
<th>Max-disj</th>
<th>Post-disj</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GDSL</strong>&lt;br&gt;(Binary tree)</td>
<td>insert</td>
<td>7680</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>delete</td>
<td>23040</td>
<td>1</td>
<td>69</td>
</tr>
<tr>
<td><strong>BSD</strong>&lt;br&gt;(splay tree)</td>
<td>delete</td>
<td>448</td>
<td>3</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>insert</td>
<td>43</td>
<td>3</td>
<td>42</td>
</tr>
<tr>
<td><strong>BSD</strong>&lt;br&gt;(red-black tree)</td>
<td>insert</td>
<td>3036</td>
<td>3</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>delete</td>
<td>1.e + 8</td>
<td>3</td>
<td>108</td>
</tr>
<tr>
<td><strong>JSW</strong>&lt;br&gt;(avl-tree)</td>
<td>insert</td>
<td>1.e + 8</td>
<td>3</td>
<td>120</td>
</tr>
</tbody>
</table>

Huisong Li, Francois Berenger, Bor-Yuh Evan Chang, Xavier Rival

Semantic-directed clumping of disjunctive abstract states (POPL’17)
Conclusion and future directions

Table of Contents

1. Introduction

2. Shape analysis for unstructured sharing
   - Abstract states
   - Analysis algorithm
   - Experimental evaluation

3. Semantic-directed clumping of disjunctive abstract states
   - Silhouettes
   - Silhouette guided clumping and joining
   - Experimental evaluation

4. Conclusion and future directions
Conclusion and future directions

Separation-logic based shape analysis for unstructured sharing

- existing separation-logic based memory abstractions can only abstract some local sharing
- combination of shape abstraction with set abstraction to capture some kind of unstructured sharing
- keep local reasoning while reasoning about some complex sharing properties

Future directions: extending our abstraction to capture other kinds of unbounded sharing

- DAGs which has a topological ordering of nodes
- sharing among several different data structures
Conclusion and future directions

Semantic-directed clumping of disjunctive abstract states

- in the analysis of programs manipulating dynamic data structures
  disjunctions are necessary
  but existing disjunction control are often syntactic, heuristic

- semantic and general disjunction clumping
  rely on silhouettes to detect imprecise join

- silhouettes guided join algorithm
  more precise than existing joining which relies on syntactic rewriting rules

Future directions: silhouette-guided weakening of other abstractions

- array abstractions
- dictionary abstractions
Thank you for your attention