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

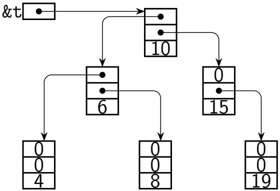
- military, medical, transportation, bank systems, ...
- hard to develop and maintain
- often buggy, e.g. a recent Mac os allows you to become a root user without a password
- testing and code review, useful, but cannot guarantee anything

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

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 $\Longrightarrow$ a program satisfies a specification

Complete

- a program satisfies a specification $\Longrightarrow$ 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., $[m_1, m_2]\overline{+}[n_1, n_2] = [m_1 + n_1, m_2 + n_2]$
- **widening $\triangledown$ for computing abstract loop invariants**

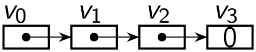
$\boxed{\text{abstract domain = abstractions + abstract operations + widening}}$

Existing analyses

- numeric analysis
- memory analysis
- ...

Points-to abstraction

Concrete memory:



Points-to abstraction:

- abstract concrete addresses with **symbolic variables**

$$v_0 \rightarrow \alpha_0 \qquad v_1 \rightarrow \alpha_1$$
$$v_2 \rightarrow \alpha_2 \qquad v_3 \rightarrow \alpha_3$$

- abstract memory cells with **points-to predicates**

$$\alpha_0 \mapsto \alpha_1 \wedge \alpha_1 \mapsto \alpha_2 \wedge \alpha_2 \mapsto \alpha_3 \wedge \alpha_3 \mapsto 0$$

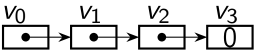
Limitation:

- hard to express **disjointness of memory cells to support strong update**

$\alpha_0 \mapsto \alpha_1$ and $\alpha_1 \mapsto \alpha_2$ describe different memory cells

Separating conjunction

Concrete memory:



Points-to abstraction with separating conjunction ($*$) (John C. Reynolds'02):

- separating conjunction ($*$) allows us to express disjointness

$$\alpha_0 \mapsto \alpha_1 * \alpha_1 \mapsto \alpha_2 * \alpha_2 \mapsto \alpha_3 * \alpha_3 \mapsto 0$$

$$\Longrightarrow$$

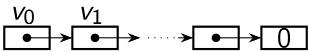
$$\forall 0 \leq i, j \leq 3, i \neq j \implies \alpha_i \neq \alpha_j$$

- separating conjunction ($*$) enables local reasoning

$$\frac{[\phi]\ P\ [\phi']}{[\phi * \psi]\ P\ [\phi' * \psi]}$$

Summarization of unbounded inductive data structures

Concrete memory:



Abstraction with summarization:

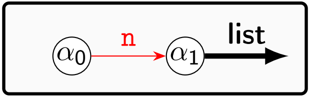
- **inductive definitions** to precisely describe dynamic data structures

$$\alpha \cdot \textbf{list} ::= \alpha = 0 \;\vee\; \alpha \neq 0 \wedge \exists \beta.\ \alpha \cdot \mathtt{n} \mapsto \beta * \beta \cdot \textbf{list}$$

- **inductive predicates** as instances of inductive definitions:

$$\boxed{\alpha_0 \cdot \mathtt{n} \mapsto \alpha_1 * \alpha_1 \cdot \textbf{list}}$$

Abstract state (formula)



Abstract state (graph)

Example: Forward analysis of a list traversal program

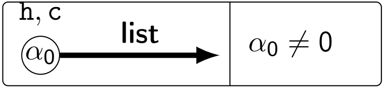

```
list* c = h;
while(c != NULL)
   c = c -> n;
```

Forward analysis:

- start from a given abstract pre-condition
- automatically compute an abstract post-condition

Example: Forward analysis of a list traversal program

```
list* c = h;
while(c != NULL)
```

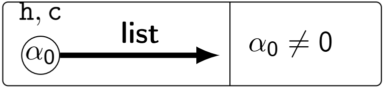



```
    c = c -> n;
```

- abstract state: shape abstraction × numerical abstraction

Example: Forward analysis of a list traversal program

```
list* c = h;
while(c != NULL)
```

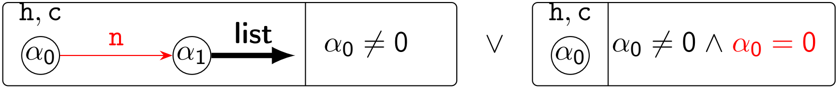



```
    c = c -> n;
```

- **Unfolding the inductive predicate**

Example: Forward analysis of a list traversal program

```
list* c = h;
while(c != NULL)
```

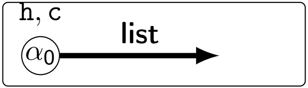



```
    c = c -> n;
```

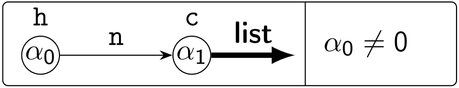
- unfolding generates case splits
- the second case is unsatisfiable

Example: Forward analysis of a list traversal program

```
list* c = h;
```



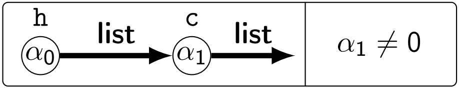

```
while(c != NULL)
    c = c -> n;
```



- widening $\triangledown$ abstract states to compute loop invariant
- widening folds back unfolded predicates

Example: Forward analysis of a list traversal program

```
list* c = h;
while(c != NULL)
```




```
    c = c -> n;
```

Abstract loop invariant

Limitation: sharing is hard to express

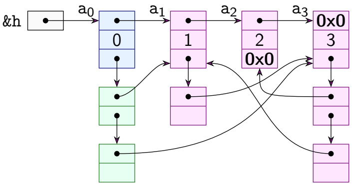
List:

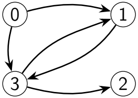
- recursive data structure
- no sharing (a node can only be dereferenced by a pointer)

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**

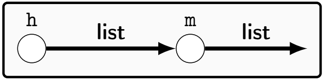




Limitation: disjunctions are necessary but costly

Without merging, disjuncts number grows exponentially in disjunctive forward analysis

unfolding



case splits: $\begin{cases} \mathtt{h} \neq \mathtt{m} \\ \vee\ \mathtt{h} = \mathtt{m} = 0 \\ \vee\ \mathtt{h} = \mathtt{m} \neq 0 \end{cases}$

$\boxed{\cdots} \vee \boxed{\cdots} \vee \boxed{\cdots}$

$\boxed{\cdots}$

```
if(...){
  ...
}
else{
  ...
}
```

$\boxed{\cdots} \vee \boxed{\cdots}$

$\boxed{\cdots}$

```
while(...){
  ...
  if(...){...}
  else{...}
```

$\boxed{\cdots} \vee \boxed{\cdots}$

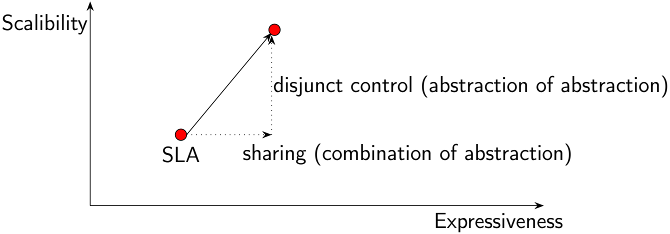
For scalability, disjuncts number should be kept small

- Fewer disjuncts means lower analysis cost
- But merging disjuncts may lose precision

Deciding how to merge disjuncts without losing too much precision is critical

Contribution of my thesis

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

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;
```

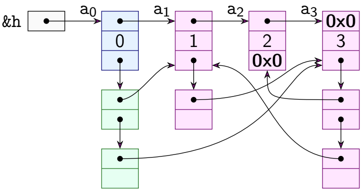
```
0  node∗ c = h;
    // start at the first node
1  while(c != NULL){
2    edge∗ s = c -> edges;
     ......
3    c = s -> dest;
4    n = c -> id;
    // random visit a successor
   }
```

Analysis goals:

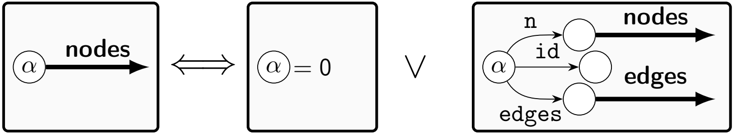
- preservation of structural properties of adjacency lists
- absence of memory errors, e.g., dereferencing of null, uninitialized, and dangling pointers

Towards precise summarization of adjacency lists

Concrete adjacency list:



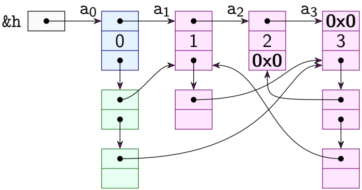
Inductive definition for adjacency lists following list of list structure:



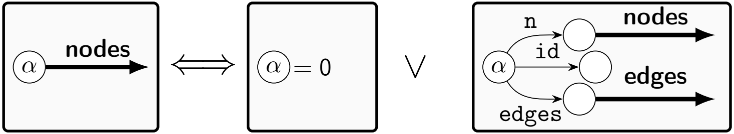
- nodes can only be dereferenced from `next` field of a previous node
- information about edge pointers is missing

Towards precise summarization of adjacency lists

Concrete adjacency list:



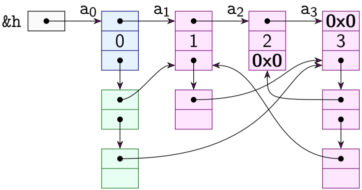
Inductive definition for adjacency lists following list of list structure:



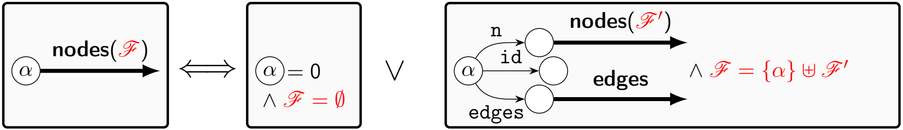
- extend the inductive definition with set parameters
- rely on set predicates to precisely capture properties of edge pointers

Towards precise summarization of adjacency lists

Concrete adjacency list:

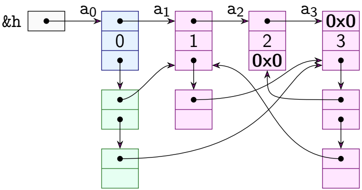


Summarize the set of node addresses by a set variable $\mathscr{F}$:

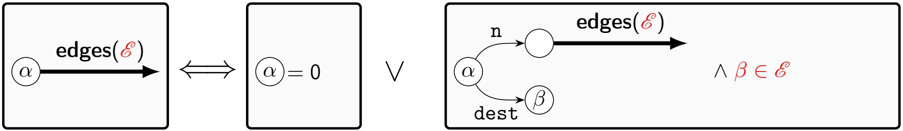


Towards precise summarization of adjacency lists

Concrete adjacency list:

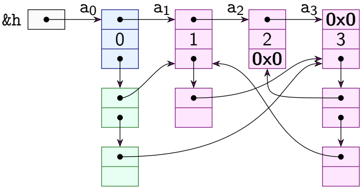


Enforce the property that each edge of a node in a set $\mathscr{E}$:

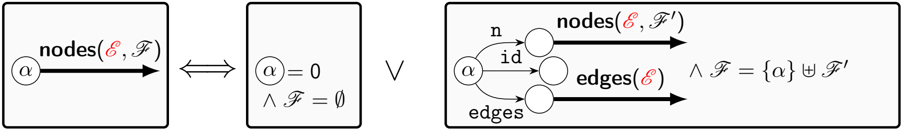


Towards precise summarization of adjacency lists

Concrete adjacency list:

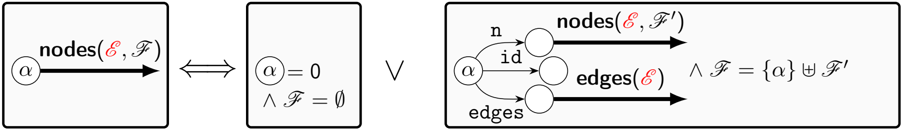


Enforce all the edges of any node in the same set:

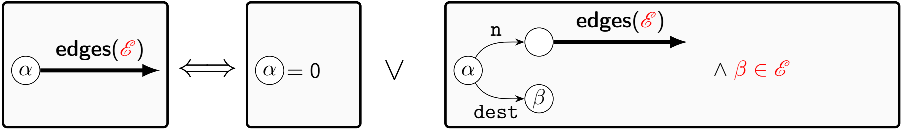


Inductive definitions of adjacency lists

Node list definition:

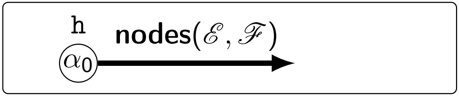


Edge list definition:



Abstract pre-condition of graph random path traversal

Precondition: `h` points-to a valid adjacency list

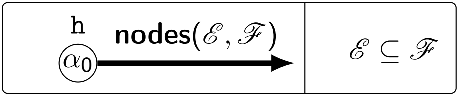


$\begin{cases} \mathscr{F} : \text{set of node addresses} \\ \mathscr{E} : \text{set of edge pointers} \end{cases}$

```
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



$\begin{cases} \mathscr{F} : \text{set of node addresses} \\ \mathscr{E} : \text{set of edge pointers} \end{cases}$

```
node* c = h;
 // start at the first node
while(c != NULL){
  edge* s = c -> edges;
  ......
  c = s -> d;
  n = c -> id;
 // random visit a successor
}
```

Abstract states

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

$\overline{\mathrm{g}}$: **shape abstraction**

- separating conjunction of points-to, inductive and segment predicates
- **parameterized by inductive definitions with set parameters**

$\overline{\mathrm{n}}$: **numerical abstraction**

- abstracts numerical constraints, e.g., $\alpha \neq 0$, $\alpha = \beta$

$\overline{\mathrm{s}}$: **set abstraction**

- **abstracts set constraints, e.g.,** $\mathscr{E} = \mathscr{F}_1 \uplus \mathscr{F}_2$

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

  $\mathscr{E}_0 = \{\alpha_0, ..., \alpha_k\} \uplus \mathscr{E}_1 \uplus \ldots \uplus \mathscr{E}_m$

  $\mathscr{E} \subseteq \mathscr{F}, \alpha \in \mathscr{E}, \mathscr{E} = \mathscr{F}$

- **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**

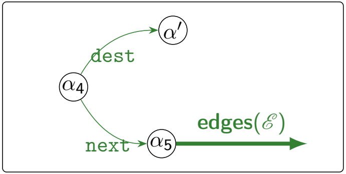
  Abstract Domains and Solvers for Sets Reasoning (LPAR'15)

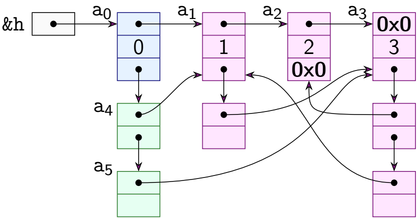
Concretization

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

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

Example:





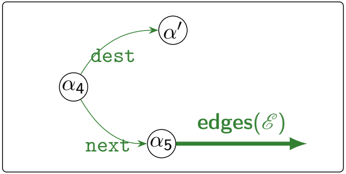
- concretization of symbolic variables and set variables
  $\alpha_4 \mapsto a_4 \quad \alpha_5 \mapsto a_5 \quad \alpha' \mapsto a_1 \quad \mathscr{E} \mapsto \{a_0, a_1, a_2, a_3\}$

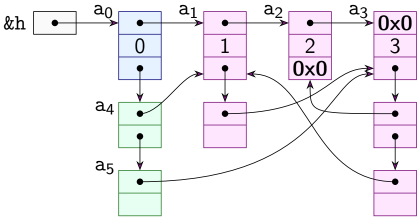
Concretization

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

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

Example:





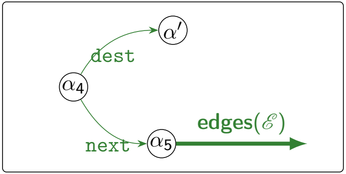
- points-to edges are concretized into concrete memory cells

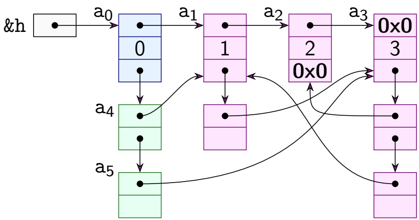
Concretization

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

- 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

Analysis principle

Extend existing abstract operations:

- abstract read, write
- **unfolding**

Support non-local unfolding

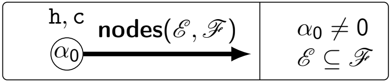
- enables dereferencing through unbounded pointers

Extend folding operations to synthesize set parameters of summary predicates:

- **joining** $\sqcup$
- widening ∇
- entailment checking $\sqsubseteq$

Unfolding

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

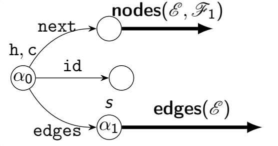



```
2       edge* s = c -> edges;    ......
3       c = s -> d;
4       n = c -> id;
     // random visit a successor
    }
```

Read a field in a summarized region

Unfolding

```
node* c = h;
 // start at the first node
while(c != NULL){
```



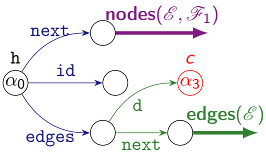
$\alpha_0 \neq 0$
$\mathscr{E} \subseteq \mathscr{F}$
$\mathscr{F} = \{\alpha_0\} \uplus \mathscr{F}_1$

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

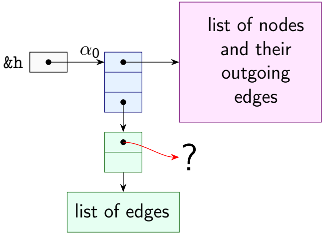
Unfolding enforces set predicates in the set abstraction

Non-local unfolding

```
node* c = h;
while(c != NULL){
  edge* s = c -> edges;
......
  c = s -> d;
```



$\mathscr{E} \subseteq \{\alpha_0\} \uplus \mathscr{F}_1$
$\alpha_3 \in \mathscr{E}$




```
  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

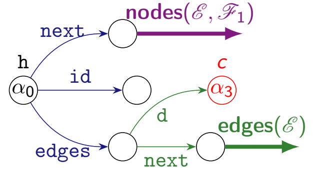
$$\alpha \in \{\alpha_0, ..., \alpha_k\} \uplus \mathscr{E}_0 \uplus \ldots \uplus \mathscr{E}_l$$

by localizing α in the shape abstraction:

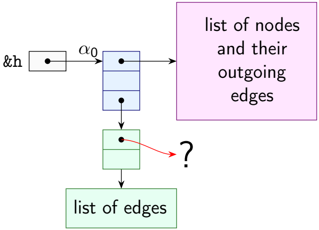
$$\begin{array}{ll} & \alpha = \alpha_0 \\ \vee & \ldots \\ \vee & \alpha = \alpha_k \\ \vee & \alpha \in \mathscr{E}_0 \\ \vee & \ldots \\ \vee & \alpha \in \mathscr{E}_l \end{array}$$

Non-local unfolding

Abstract state:



$\mathscr{E} \subseteq \{\alpha_0\} \uplus \mathscr{F}_1$
$\alpha_3 \in \mathscr{E}$

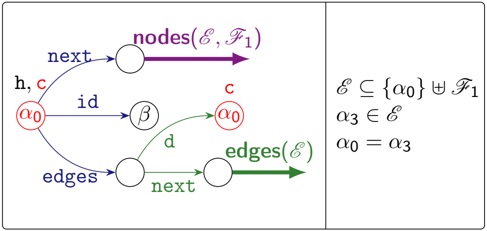


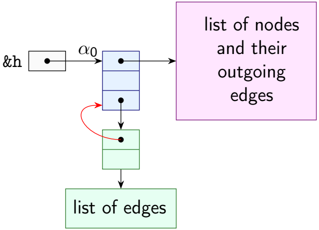
Perform non-local unfolding at α_3 according to the set abstraction constraint $\alpha_3 \in \{\alpha_0\} \uplus \mathscr{F}_1$:

- $\alpha_3 = \alpha_0$
- $\alpha_3 \in \mathscr{F}_1$

Non-local unfolding

Localize α_3 to α_0:



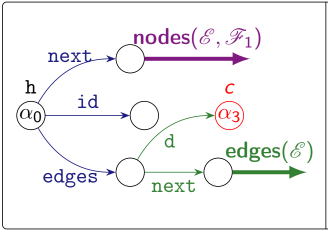


$$\overline{\textbf{read}}(\alpha_3 \cdot \texttt{id}) = \overline{\textbf{read}}(\alpha_0 \cdot \texttt{id}) = \beta$$

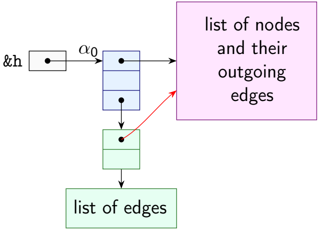
Non-local unfolding

Localize α_3 to $\mathscr{F}_1$ according to properties of the set parameter:

- $\mathscr{F}_1$ denotes the set of the addresses of the nodes described by the inductive edge
- α_3 is a node address summarized by the inductive edge



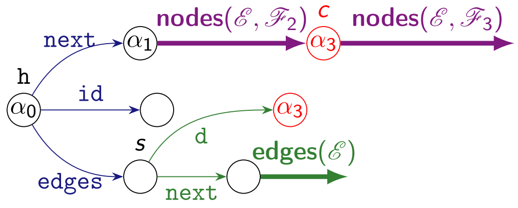
$\mathscr{E} \subseteq \{\alpha_0\} \uplus \mathscr{F}_1$
$\alpha_3 \in \mathscr{E}$



Non-local unfolding

Localize α_3 to $\mathscr{F}_1$ according to properties of the set parameter:

- $\mathscr{F}_1$ denotes the set of the addresses of the nodes described by the inductive edge
- α_3 is a node address summarized by the inductive edge



$\mathscr{E} \subseteq \{\alpha_0\} \uplus \mathscr{F}_2 \uplus \mathscr{F}_3$

$\alpha_3 \in \mathscr{F}_2 \uplus \mathscr{F}_3$

Splitting the inductive edge into a segment and an inductive edge

Joining

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

- compute a sound and common weaker abstraction $(\overline{\mathrm{g}}_o, \overline{\mathrm{s}}_o)$

$$\gamma(\overline{\mathrm{g}}_l, \overline{\mathrm{s}}_l) \subseteq \gamma(\overline{\mathrm{g}}_o, \overline{\mathrm{s}}_o)$$
$$\gamma(\overline{\mathrm{g}}_r, \overline{\mathrm{s}}_r) \subseteq \gamma(\overline{\mathrm{g}}_o, \overline{\mathrm{s}}_o)$$

- joining of abstract shapes based on graph rewriting rules

Graph rewriting rules for shape joining

Weakening identical predicates:

$$\frac{}{(\overline{\mathrm{g}}, \overline{\mathrm{s}}_l) \sqcup_{\overline{\mathcal{G}}} (\overline{\mathrm{g}}, \overline{\mathrm{s}}_r) = \overline{\mathrm{g}}}$$

Weakening guided by existing summary predicates:

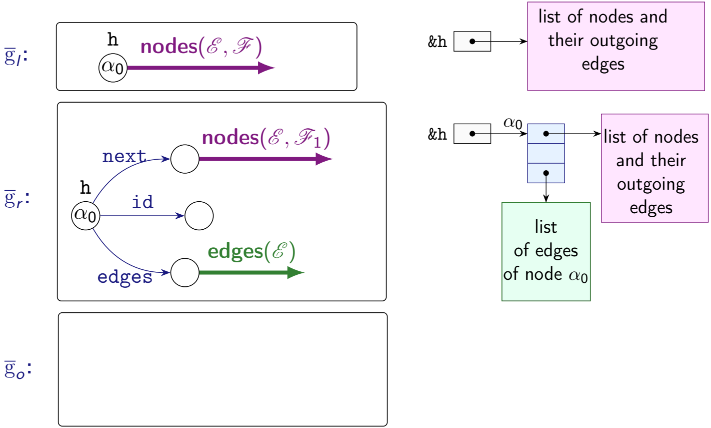
$$\frac{\overline{\mathrm{s}}'_l \text{ instantiates } \mathscr{E} \text{ in } \overline{\mathrm{s}}_l \qquad \overline{\mathrm{s}}'_l \vdash \overline{\mathrm{g}}_l \sqsubseteq \alpha \cdot \mathbf{ind}(\mathscr{E})}{(\overline{\mathrm{g}}_l, \overline{\mathrm{s}}_l) \sqcup_{\overline{\mathcal{G}}} (\alpha \cdot \mathbf{ind}(\mathscr{E}), \overline{\mathrm{s}}_r) = \alpha \cdot \mathbf{ind}(\mathscr{E})}$$

- instantiate $\mathscr{E}$: resolve the meaning of set parameter $\mathscr{E}$ 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

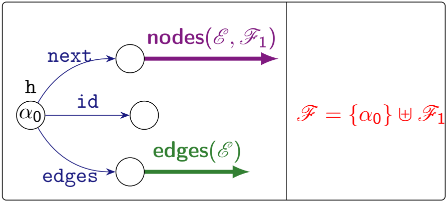
A joining example



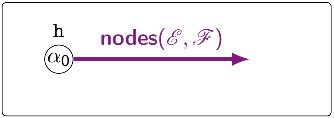
A joining example

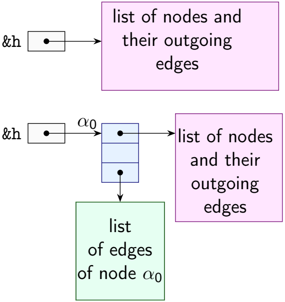
$\overline{g}_l$:

$\overline{g}_r$:



$\overline{g}_O$:





- weaken the right side abstraction into the inductive edge
- instantiate set parameter $\mathscr{F}$ according to the weakening process

Graph random path traversal

$\overset{\mathtt{h}}{\alpha_0} \xrightarrow{\mathbf{nodes}(\mathscr{E}, \mathscr{F})}$	$\mathscr{E} \subseteq \mathscr{F}$

```
node* c = h;
 // start at the first node
while(c != NULL){
```

$\overset{\mathtt{h}}{\alpha_0} \xrightarrow{\mathbf{nodes}(\mathscr{E}, \mathscr{X}_0)} \overset{\mathtt{c}}{\alpha_1} \xrightarrow{\mathbf{nodes}(\mathscr{E}, \mathscr{X}_1)}$	$\mathscr{E} \subseteq \mathscr{F} \wedge \mathscr{F} = \mathscr{X}_0 \uplus \mathscr{X}_1$

```
   edge* s = c -> edges;
   ......
   c = s -> dest;
   n = c -> id;
 // random visit a successor
}
```

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

Description	LOCs	"BDD" time (ms)			"BDD"	"LIN" time (ms)			"LIN"
		Total	Shape	Set	Property	Total	Shape	Set	Property
Node: add	27	44	0.3	11	**yes**	28	0.3	0.2	**yes**
Edge: add	26	31	0.2	4	**yes**	27	0.2	0.1	**yes**
Edge: delete	22	45	0.4	16	**yes**	30	0.3	0.2	**yes**
Node list traversal	25	117	1.5	87	**yes**	28	0.5	0.3	**yes**
Edge list iteration + dest. read	34	332	2.7	293	**yes**	36	3.5	2.4	**yes**
Graph path: deterministic	31	360	2.7	323	**yes**	35	2.4	2	**yes**
Graph path: random	43	765	7.1	711	**yes**	41	4.1	3	**yes**

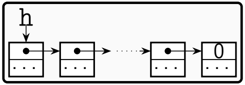
Experiment results and conclusion

- **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

Disjunction is necessary

Concrete memory:

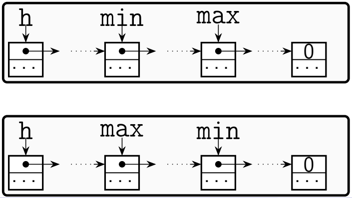


Abstract state:

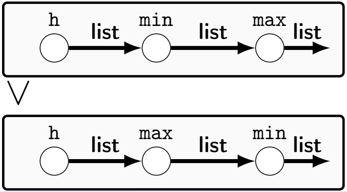
search_min_max()

```
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:



Disjunctive abstract post state:



Existing techniques to deal with disjunctions

Disjunctive completion(Cousot&Cousot'79):

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

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

$\mathcal{A}$: **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 $\Phi : \mathcal{A} \longrightarrow \mathcal{A}'$, 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 $\mathcal{P}(\mathcal{A})$, i.e. allow all disjunctive abstractions

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

first, use a sound normalization $\Phi : \mathcal{A} \longrightarrow \mathcal{A}'$, 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 $\mathcal{B} \longrightarrow \mathcal{A}$, with finite $\mathcal{B}$

- partition criteria include control flow, context, etc
- such criteria tend to not work effectively for shape analysis

$\mathcal{A}$: analysis domain
$\mathcal{B}$: 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 $\Phi : \mathcal{A} \longrightarrow \mathcal{A}'$, 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)

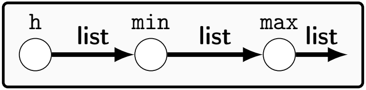
effectively use a lattice of the form $\mathcal{B} \longrightarrow \mathcal{A}$, with finite $\mathcal{B}$

We need a semantic technique to improve disjunction handling

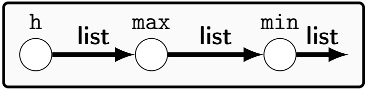
Precision loss in join

Abstract states:

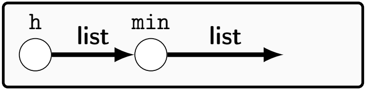
$\overline{\mathrm{m}}_0$



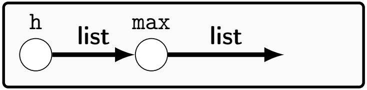
$\overline{\mathrm{m}}_1$



Imprecise upper bounds:



pointer `max` is lost



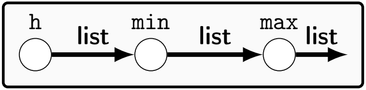
pointer `min` is lost

Abstract states $\overline{\mathrm{m}}_0$ and $\overline{\mathrm{m}}_1$ have several incomparable, imprecise upper bounds, but no precise least upper bound
Joining them will lose precision

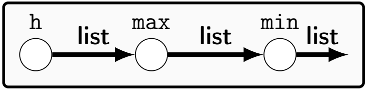
Precision loss in join

Abstract states:

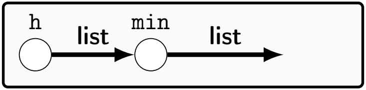
$\overline{\mathrm{m}}_0$



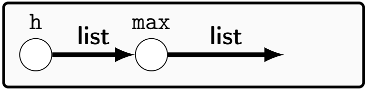
$\overline{\mathrm{m}}_1$



Imprecise upper bounds:



pointer `max` is lost



pointer `min` is lost

Observation

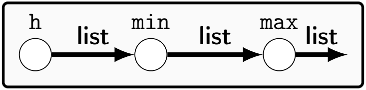
Pointers' order in data structures has a big impact on join precision

When pointers are in different orders, join tends to be very imprecise as no abstract state can preserve different pointers' orders

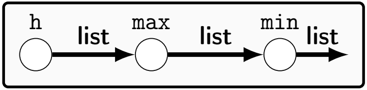
Precision loss in join

Abstract states:

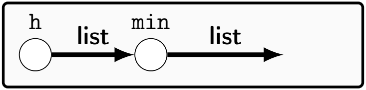
$\overline{m}_0$



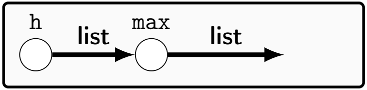
$\overline{m}_1$



Imprecise upper bounds:



pointer `max` is lost



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

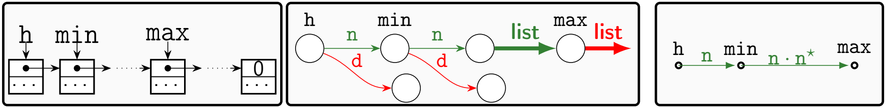
Silhouette graph

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

$$e ::= \epsilon \mid \mathtt{f} \mid (\mathtt{f}_0 + \ldots + \mathtt{f}_n)^\star \mid e \cdot e$$

Silhouette is an abstraction of abstract states.

Concrete memory: **Abstract state:** **Silhouette:**



all the red edges are abstracted away.
all the green edges are abstracted by access paths.
the **list** predicate is abstracted by access path $\mathtt{n}^\star$.

Silhouette-based weak abstract entailment check

Entailment check algorithms

$\sqsubseteq_{\overline{\mathcal{M}_\omega}}$: decide inclusion of abstract states(complex rewriting rules)
$\sqsubseteq_{\overline{\mathfrak{S}}}$: decide inclusion of silhouettes(classical inclusion of constraints)

- Silhouette entailment check $\sqsubseteq_{\overline{\mathfrak{S}}}$ offers a weak characterization for abstract states entailment check $\sqsubseteq_{\overline{\mathcal{M}_\omega}}$

$$\begin{array}{rcl} \sqsubseteq_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_0, \overline{\mathrm{m}}_1) = \textbf{true} & \Longrightarrow & \sqsubseteq_{\overline{\mathfrak{S}}}(\textbf{sil}(\overline{\mathrm{m}}_0), \textbf{sil}(\overline{\mathrm{m}}_1)) = \textbf{true} \\ \sqsubseteq_{\overline{\mathfrak{S}}}(\textbf{sil}(\overline{\mathrm{m}}_0), \textbf{sil}(\overline{\mathrm{m}}_1)) = \textbf{false} & \Longrightarrow & \sqsubseteq_{\overline{\mathcal{M}_\omega}}(\overline{\mathrm{m}}_0, \overline{\mathrm{m}}_1) = \textbf{false} \end{array}$$

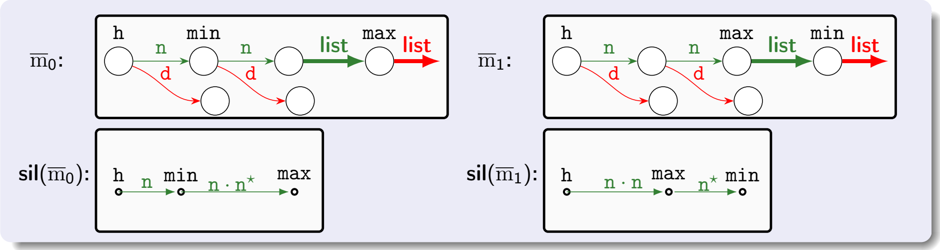
- Silhouette entailment check $\sqsubseteq_{\overline{\mathfrak{S}}}$ is much cheaper

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

Silhouette-based weak abstract entailment check

Entailment check algorithms

- $\sqsubseteq_{\overline{\mathcal{M}_\omega}}$: decide inclusion of abstract states(complex rewriting rules)
- $\sqsubseteq_{\overline{\mathfrak{S}}}$: decide inclusion of silhouettes(classical inclusion of constraints)



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

Silhouette join

Silhouette join

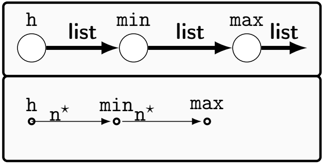
- Offers a weak, but precise characterization for abstract states join

  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:

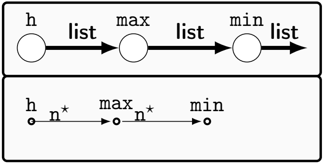
$\overline{\mathrm{m}}_0$:

sil($\overline{\mathrm{m}}_0$):



$\overline{\mathrm{m}}_1$:

sil($\overline{\mathrm{m}}_1$):



The silhouettes of $\overline{\mathrm{m}}_0$ and $\overline{\mathrm{m}}_1$ cannot be joined precisely

Abstract states $\overline{\mathrm{m}}_0$ and $\overline{\mathrm{m}}_1$ cannot also be joined precisely

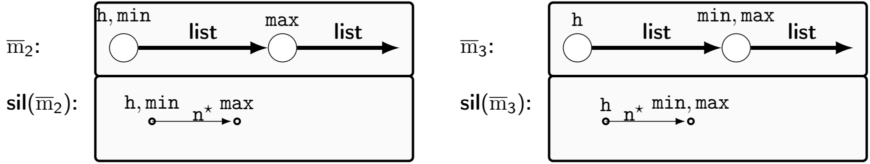
Silhouette join

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:



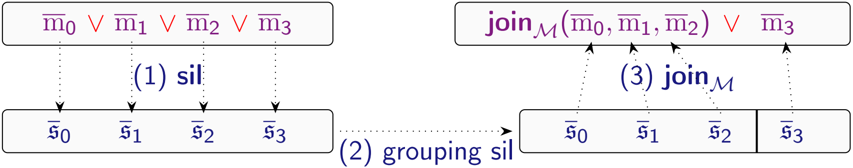
The silhouettes of $\overline{m}_2$ and $\overline{m}_3$ can be joined precisely

Abstract states $\overline{m}_2$ and $\overline{m}_3$ can also be joined precisely

Table of Contents

Silhouette guided clumping

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



Silhouette association relation $\bowtie$:

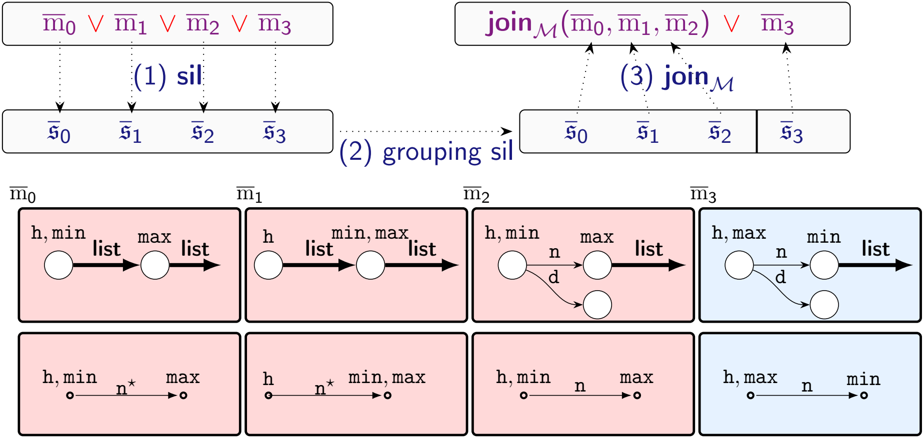
Let $\bar{\mathfrak{s}}_0, \bar{\mathfrak{s}}_1$ be two silhouettes with the same set of nodes N. We write $\bar{\mathfrak{s}}_0 \bowtie \bar{\mathfrak{s}}_1$ if and only if there exist $\mathrm{N}_0, \mathrm{N}_1$ such that $\mathrm{N} = \mathrm{N}_0 \cup \mathrm{N}_1$ and:

$$\begin{array}{ll} & \bar{\mathfrak{s}}'_0 = \bar{\mathfrak{s}}'_{0\lceil \mathrm{N}_0} \cup \bar{\mathfrak{s}}'_{0\lceil \mathrm{N}_1} \quad \wedge \quad \bar{\mathfrak{s}}'_{0\lceil \mathrm{N}_0} \sqsubseteq_{\mathbb{S}} \bar{\mathfrak{s}}'_{1\lceil \mathrm{N}_0} \\ \wedge & \bar{\mathfrak{s}}'_1 = \bar{\mathfrak{s}}'_{1\lceil \mathrm{N}_0} \cup \bar{\mathfrak{s}}'_{1\lceil \mathrm{N}_1} \quad \wedge \quad \bar{\mathfrak{s}}'_{1\lceil \mathrm{N}_1} \sqsubseteq_{\mathbb{S}} \bar{\mathfrak{s}}'_{0\lceil \mathrm{N}_1} \end{array}$$

Characterizes precise joins that can be computed based on weakening rules guided by existing predicates

Silhouette guided clumping

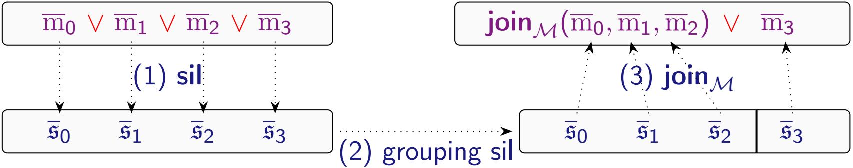
Algorithm: group silhouettes based on an equivalence clumping relation which captures precise silhouette join (cheap to compute)

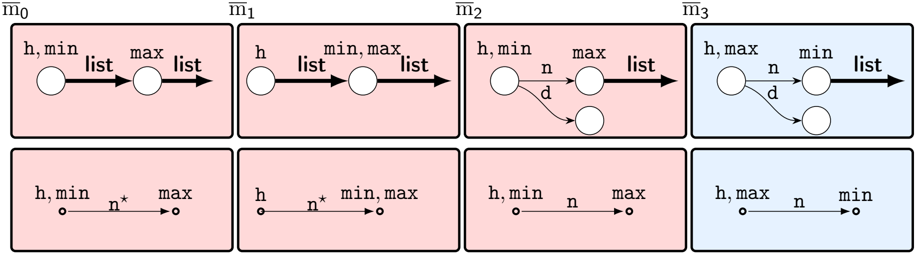


Silhouette groups: $\{\bar{\mathfrak{s}}_0, \bar{\mathfrak{s}}_1, \bar{\mathfrak{s}}_2\}$, $\{\bar{\mathfrak{s}}_3\}$

Silhouette guided clumping

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





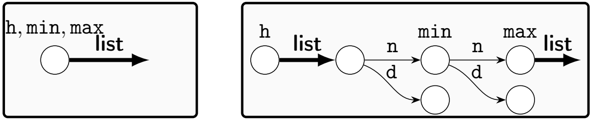
Clumping result: $\mathbf{join}_{\mathcal{M}}(\overline{m}_0, \overline{m}_1, \overline{m}_2) \vee \overline{m}_3$

Abstract states join without silhouette

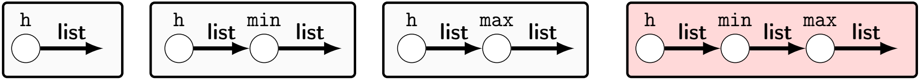
Join abstract states $\mathbf{join}_{\mathcal{M}}(\overline{\mathrm{m}}_0, \overline{\mathrm{m}}_1)$:

- **Compute an over-approximation of $\overline{\mathrm{m}}_0$, $\overline{\mathrm{m}}_1$.**
- **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

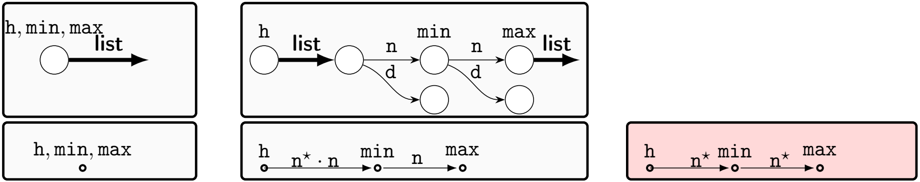
Silhouette join guides abstract states join to be precise.

select which rewriting rules to apply.

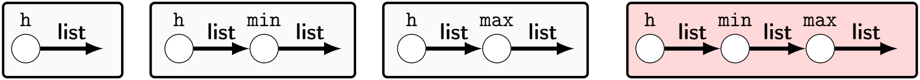
help to synthesize inductive predicates.

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

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)

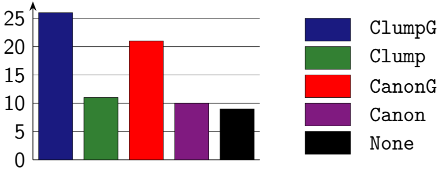
- GDSL: binary search tree, list (insert, delete, ...)
- BSD: red-black tree, splay tree (insert, delete, ...)
- JSW: AVL tree (insert, ...)
- ......

Clumping improves precision

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

		Clumping	Canonicalization	
Guided joining	Y	ClumpG	CanonG	
	N	Clump	Canon	MemCAD baseline

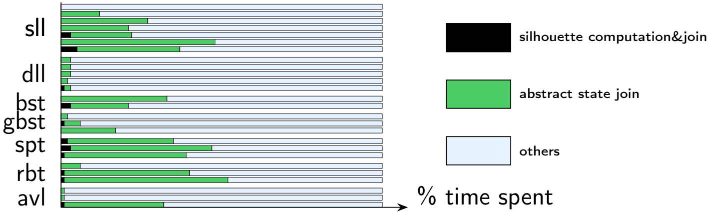
Number of verified benchmarks using each strategy:



- **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

Benchmark		Path	Fix-disj	Max-disj	Post-disj
GDSL (Binary tree)	insert	7680	2	4	1
	delete	23040	1	69	1
BSD (splay tree)	delete	448	3	42	1
	insert	43	3	42	1
BSD (red-black tree)	insert	3036	3	51	1
	delete	$1.e+8$	3	108	1
JSW (avl-tree)	insert	$1.e+8$	3	120	1

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

Benchmark		Path	Fix-disj	Max-disj	Post-disj
GDSL (Binary tree)	insert	7680	2	4	1
	delete	23040	1	69	1
BSD (splay tree)	delete	448	3	42	1
	insert	43	3	42	1
BSD (red-black tree)	insert	3036	3	51	1
	delete	$1.e+8$	3	108	1
JSW (avl-tree)	insert	$1.e+8$	3	120	1

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

Semantic-directed clumping of disjunctive abstract states (POPL'17)

Table of Contents

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