Precise and Compact Modular Procedure Summaries for Heap Manipulating Programs

Işıl Dillig, Thomas Dillig, Alex Aiken
Stanford University

Mooly Sagiv
Tel-Aviv University

Goal:

Perform a precise flow- and context- sensitive pointer analysis that is modular and bottom-up

Advantages of Modular Pointer Analysis

- Reuse of results: Same summary can be reused in any context

  $\Rightarrow$ Each function only analyzed once (assuming no cycles)

Advantages of Modular Pointer Analysis

- Reuse of results: Same summary can be reused in any context

  $\Rightarrow$ Each function only analyzed once (assuming no cycles)

- Scalability: Summaries express only externally visible side effects

  $\Rightarrow$ Allows local reasoning

Advantages of Modular Pointer Analysis

- Reuse of results: Same summary can be reused in any context
  - $\Rightarrow$ Each function only analyzed once (assuming no cycles)
- Scalability: Summaries express only externally visible side effects
  - $\Rightarrow$ Allows local reasoning
- Natural parallelization: Functions that do not have caller-callee relationship can be independently analyzed

Unfortunately performing a modular pointer analysis is difficult!

$\Rightarrow$ particularly if we want to perform strong updates to memory locations!

Motivating Example

```
f(int** a, int **b,
  int *p, int *q)
{
  *a = p;
  *b = q;
  **a = 3;
  **b = 4;
}
```

Motivating Example

```
f(int** a, int **b,
  int *p, int *q)
{
  *a = p;
  *b = q;
  **a = 3;
  **b = 4;
}
```

Motivating Example

```
f(int** a, int **b,
  int *p, int *q)
{
  *a = p;
  *b = q;
  **a = 3;
  **b = 4;
}
```

Motivating Example

```
f(int** a, int **b,
  int *p, int *q)
{
  *a = p;
  *b = q;
  **a = 3;
  **b = 4;
}
```

Motivating Example

```
f(int** a, int **b,
  int *p, int *q)
{
  *a = p;
  *b = q;
  **a = 3;
  **b = 4;
}
```

Motivating Example

```
f(int** a, int **b,
  int *p, int *q)
{
  *a = p;
  *b = q;
  **a = 3;
  **b = 4;
}
```

- Although f is conditional and loop-free, it may have very different effects at different call sites

Motivating Example

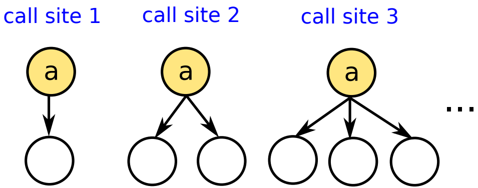
```
f(int** a, int **b,
  int *p, int *q)
{
  *a = p;
  *b = q;
  **a = 3;
  **b = 4;
}
```

- Although `f` is conditional and loop-free, it may have very different effects at different call sites
- Example: After a call to `f`, value of `*p` may be 3, 4, or remain its initial value

  ... depending on points-to facts at call site!

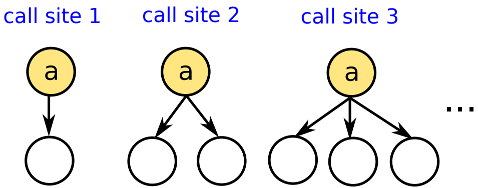
Two Main Difficulties

One difficulty: An argument `a` to a function `f` may have different number of points-to targets at different call sites of `f`



Two Main Difficulties

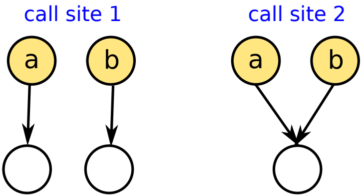
One difficulty: An argument `a` to a function `f` may have different number of points-to targets at different call sites of `f`



⇒ Unknown number of points-to targets at call sites

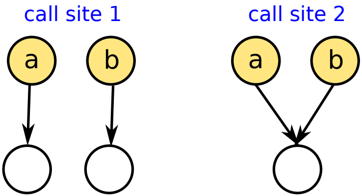
Two Main Difficulties

Another difficulty: Different aliasing patterns between arguments may exist at different call sites



Two Main Difficulties

Another difficulty: Different aliasing patterns between arguments may exist at different call sites



⇒ Aliasing patterns exponential in number of locations

Overview of Our Approach

- Represent unknown points-to targets of locations using location variables

- Represent unknown points-to targets of locations using location variables
- To allow strong updates, ensure that locations represented by two distinct variables stand for disjoint set of locations

Overview of Our Approach

- Represent unknown points-to targets of locations using location variables
- To allow strong updates, ensure that locations represented by two distinct variables stand for disjoint set of locations
- Enforce disjointness by symbolically representing all possible aliasing relations on function entry

Location Constants vs. Variables

Distinguish between two kinds of abstract memory locations:

Location Constants vs. Variables

Distinguish between two kinds of abstract memory locations:

- Location Constants: Model memory allocations, NULL, locations of stack variables etc.

Location Constants vs. Variables

Distinguish between two kinds of abstract memory locations:

- Location Constants: Model memory allocations, NULL, locations of stack variables etc.
- Location Variables: Range over the unknown location constants pointed to by arguments at function entry

Simple Example

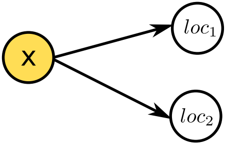
foo(int* a)

ν ranges over abstract memory locations at call sites of foo

Simple Example

foo(int* a)

...
foo(x)
...



ν ranges over abstract memory locations at call sites of foo

In this context, ν stands for location constants loc_1 and loc_2

If ν_1 and ν_2 are two distinct location variables in `f`, we can only apply strong updates to them in `f` if:

$$\gamma(\nu_1) \cap \gamma(\nu_2) = \emptyset$$

in any calling context

Strong Updates to Location Variables

If ν_1 and ν_2 are two distinct location variables in `f`, we can only apply strong updates to them in `f` if:

$$\gamma(\nu_1) \cap \gamma(\nu_2) = \emptyset$$

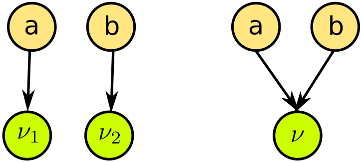
in any calling context

Why?

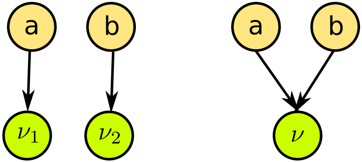
If ν_1 and ν_2 may represent an overlapping set of locations, updates to ν_1 may affect updates to ν_2

Enforcing Disjointness: Naive Solution

- If arguments `a` and `b` are potential aliases, analyze function in two different initial configurations:



- If arguments `a` and `b` are potential aliases, analyze function in two different initial configurations:



Problem:

Number of alias patterns = nth Bell number
(n= # of argument-reachable locations)

Enforcing Disjointness: Practical Solution

Encode aliasing patterns symbolically such that:

Enforcing Disjointness: Practical Solution

Encode aliasing patterns symbolically such that:

- Number of location variables, n, is the number of argument-reachable locations

Enforcing Disjointness: Practical Solution

Encode aliasing patterns symbolically such that:

- Number of location variables, n, is the number of argument-reachable locations
- Number of edges in the initial points-to graph is bound by $n^2/2$

Enforcing Disjointness: Practical Solution

Encode aliasing patterns symbolically such that:

- Number of location variables, n, is the number of argument-reachable locations
- Number of edges in the initial points-to graph is bound by $n^2/2$
- Only need to analyze each function once

Enforcing Disjointness: Practical Solution

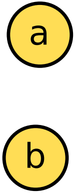
Encode aliasing patterns symbolically such that:

- Number of location variables, n, is the number of argument-reachable locations
- Number of edges in the initial points-to graph is bound by $n^2/2$
- Only need to analyze each function once

$\Rightarrow$ Since we precisely account for all aliasing patterns in any context, it is safe to apply strong updates to (non-summary) location variables

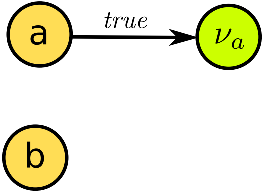
Construction of the Initial Points-to Graph

Consider function: `foo(int* a, int* b)`



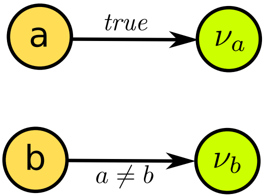
Construction of the Initial Points-to Graph

Consider function: `foo(int* a, int* b)`



ν_a represents points-to targets of `a` in any calling context

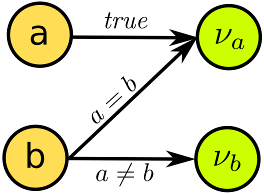
Consider function: `foo(int* a, int* b)`



ν_b represents points-to targets of b only in those contexts where a and b do not alias

Construction of the Initial Points-to Graph

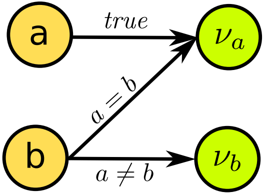
Consider function: `foo(int* a, int* b)`



ν_a also represents points-to targets of `b` in those contexts where `a` and `b` alias

Construction of the Initial Points-to Graph

Consider function: `foo(int* a, int* b)`



ν_a also represents points-to targets of `b` in those contexts where `a` and `b` alias

Observe: Construction enforces that $\gamma(\nu_a) \cap \gamma(\nu_b) = \emptyset$

Construction: The General Case

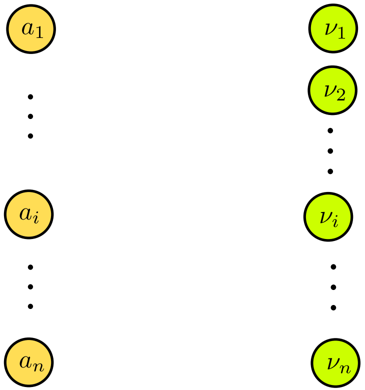
- Consider variables $a_1, \ldots, a_n$ that may alias at function entry

Construction: The General Case

- Consider variables $a_1, \ldots, a_n$ that may alias at function entry
- Impose total order such that $a_1 < a_2 \ldots < a_n$

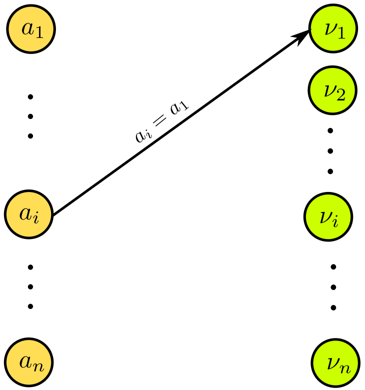
Construction: The General Case

- Consider variables $a_1, \ldots, a_n$ that may alias at function entry
- Impose total order such that $a_1 < a_2 \ldots < a_n$
- For each a_i introduce ν_i



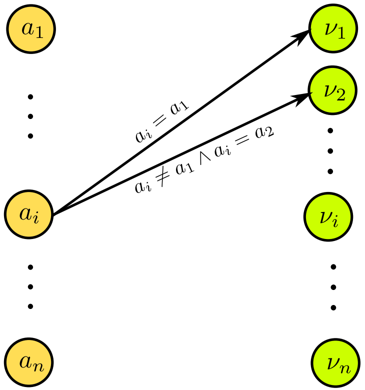
Construction: The General Case

- Consider variables $a_1, \ldots, a_n$ that may alias at function entry
- Impose total order such that $a_1 < a_2 \ldots < a_n$
- For each a_i introduce ν_i



Construction: The General Case

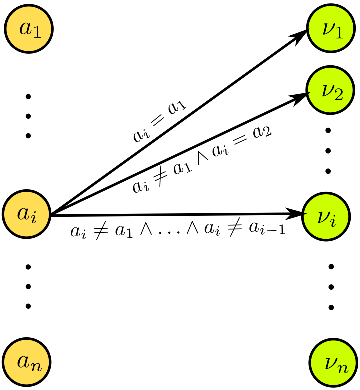
- Consider variables $a_1, \ldots, a_n$ that may alias at function entry
- Impose total order such that $a_1 < a_2 \ldots < a_n$
- For each a_i introduce ν_i



Construction: The General Case

- Consider variables $a_1, \ldots, a_n$ that may alias at function entry
- Impose total order such that $a_1 < a_2 \ldots < a_n$
- For each a_i introduce ν_i
- Each a_i points to ν_k with $k \leq i$ under constraint:

$$\bigwedge_{j<k} a_i \neq a_j \;\wedge\; a_i = a_k$$

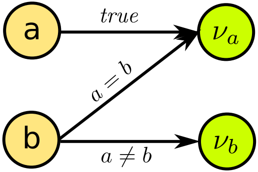


Example

```
f(int* a, int *b)
{
  *a = 1;
  *b = 2;
}
```

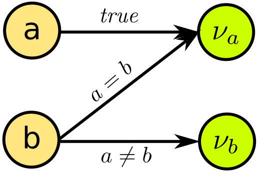
Example

```
f(int* a, int *b)
{
  *a = 1;
  *b = 2;
}
```



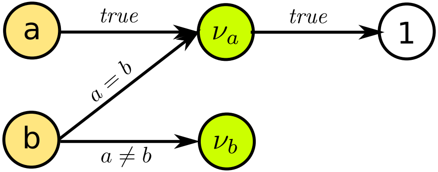
Example

```
f(int* a, int *b)
{
  *a = 1;
  *b = 2;
}
```



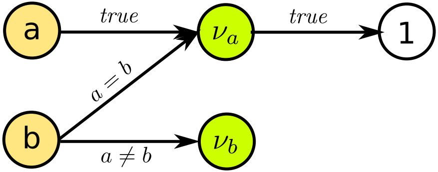
Example

```
f(int* a, int *b)
{
  *a = 1;
  *b = 2;
}
```



Example

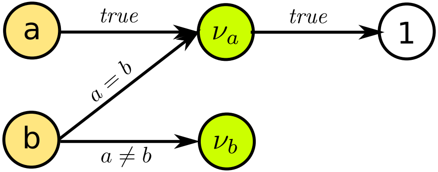
```
f(int* a, int *b)
{
  *a = 1;
  *b = 2;
}
```



Observe: `*b` has value 1 if `a` and `b` alias

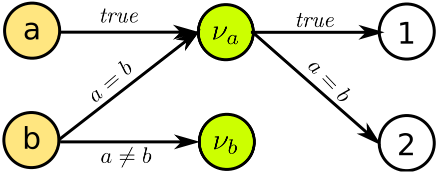
Example

```
f(int* a, int *b)
{
  *a = 1;
  *b = 2;
}
```



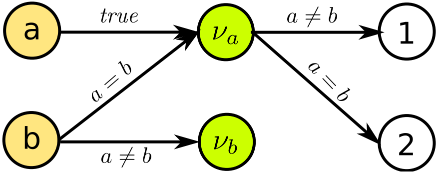
Example

```
f(int* a, int *b)
{
  *a = 1;
  *b = 2;
}
```



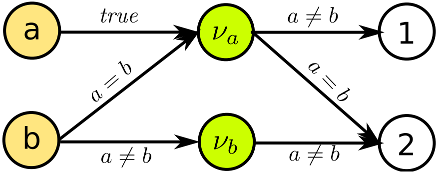
Example

```
f(int* a, int *b)
{
  *a = 1;
  *b = 2;
}
```



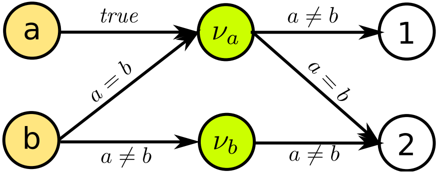
Example

```
f(int* a, int *b)
{
  *a = 1;
  *b = 2;
}
```



Example

```
f(int* a, int *b)
{
  *a = 1;
  *b = 2;
}
```



Observe: `*a` has value 1 if `a` and `b` do not alias and value 2 otherwise

Experiments

- Analyzed 4 large open-source C and C++ applications:
  - OpenSSH
  - LiteSQL
  - Inkscape Widgets
  - DigiKam

First Experiment

- Goal: Assess importance of strong updates at call sites

First Experiment

- Goal: Assess importance of strong updates at call sites
- Checked for various memory safety properties, such as buffer overruns, null dereferences, accessing deleted memory, ...

First Experiment

- Goal: Assess importance of strong updates at call sites
- Checked for various memory safety properties, such as buffer overruns, null dereferences, accessing deleted memory, ...
- Compared false positive rates of new analysis with analysis that only performs weak updates at call sites

Comparison of False Positives

- Weak updates at call sites:
  98.2% false positive rate

Comparison of False Positives

- Weak updates at call sites:
  98.2% false positive rate
- Strong updates using this technique:
  26.3% false positive rate

Comparison of False Positives

- Weak updates at call sites:
  98.2% false positive rate

- Strong updates using this technique:
  26.3% false positive rate

⇒ Modular analysis that cannot apply strong updates too imprecise!

Comparison of Running Times

- Weak updates at call sites: 20.0 min average running time on single CPU

Comparison of Running Times

- Weak updates at call sites:
  20.0 min average running time
  on single CPU
- Strong updates using this technique:
  15.2 min average running time
  on single CPU

Comparison of Running Times

- Weak updates at call sites:
  20.0 min average running time
  on single CPU

- Strong updates using this technique:
  15.2 min average running time
  on single CPU

⇒ More precise actually analysis runs faster

Analysis can be parallelized

- Also ran this analysis on 8 CPUs

Analysis can be parallelized

- Also ran this analysis on 8 CPUs
- Functions with no caller-callee relationship analyzed in parallel

Analysis can be parallelized

- Also ran this analysis on 8 CPUs
- Functions with no caller-callee relationship analyzed in parallel
- Average speed-up over 1 CPU: $4.2\times$ speedup

Second Experiment

- Goal: Assess scalability of summary-based analysis

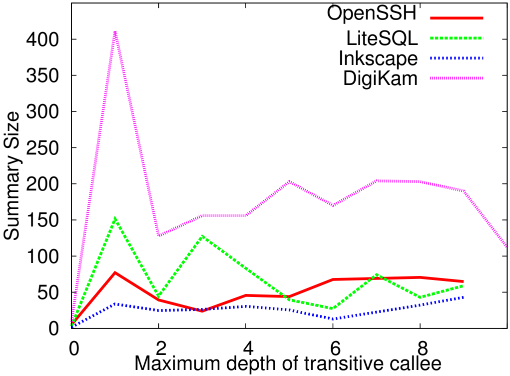
Second Experiment

- Goal: Assess scalability of summary-based analysis
- Explored growth of heap summaries vs. depth of call chain

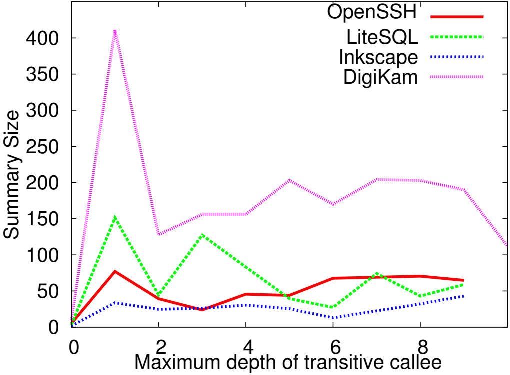
Second Experiment

- Goal: Assess scalability of summary-based analysis
- Explored growth of heap summaries vs. depth of call chain
- Measured summary size as the number of points-to edges weighted according to the size of the edge constraints

Results


OpenSSH
LiteSQL
Inkscape
DigiKam
Summary Size
0
50
100
150
200
250
300
350
400
0
2
4
6
8
Maximum depth of transitive callee

Results



Local reasoning by focusing only on externally-visible side effects

Conclusion

- Presented a modular, strictly bottom-up pointer analysis

Conclusion

- Presented a modular, strictly bottom-up pointer analysis
- Technique capable of performing strong updates at call sites

Conclusion

- Presented a modular, strictly bottom-up pointer analysis
- Technique capable of performing strong updates at call sites
- Demonstrated practicality of technique for verifying memory safety on four applications

Thanks!

Chatterjee, R., Ryder, B., Landi, W.:
Relevant context inference.
In: POPL, ACM (1999) 133–146

Whaley, J., Rinard, M.:
Compositional pointer and escape analysis for Java programs.
In: OOPSLA, ACM (1999) 187–206

Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.:
Compositional shape analysis by means of bi-abduction.
POPL (2009) 289–300

Cousot, P., Cousot, R.:
Modular static program analysis.
In: CC. (2002) 159–178

Gulwani, S., Tiwari, A.:
Computing procedure summaries for interprocedural analysis.
ESOP (2007) 253–267

Yorsh, G., Yahav, E., Chandra, S.:
Generating precise and concise procedure summaries.
POPL **43**(1) (2008) 221–234