Precise and Compact Modular Procedure Summaries for Heap Manipulating Programs

Işıl Dillig, Thomas Dillig, Alex Aiken
Stanford University

Mooly Sagiv
Tel-Aviv University
Our Goal

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
  - Each function only analyzed once (assuming no cycles)
Advantages of Modular Pointer Analysis

- **Reuse of results**: Same summary can be reused in any context
  - Each function only analyzed once (assuming no cycles)

- **Scalability**: Summaries express only externally visible side effects
  - Allows local reasoning
Advantages of Modular Pointer Analysis

- **Reuse of results**: Same summary can be reused in any context
  - Each function only analyzed once (assuming no cycles)

- **Scalability**: Summaries express only externally visible side effects
  - 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!

⇒ particularly if we want to perform strong updates to memory locations!
Motivating Example

```c
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!
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!
Motivating Example

\begin{verbatim}
f(int** a, int **b, int *p, int *q)
{
    *a = p;
    *b = q;
    **a = 3;
    **b = 4;
}
\end{verbatim}

Although \texttt{f} is conditional and loop-free, it may have very different effects at different call sites. Example: After a call to \texttt{f}, value of \texttt{*p} may be 3, 4, or remain its initial value... depending on points-to facts at call site!
Motivating Example

```c
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 \( f \), the value of \( *p \) may be 3, 4, or remain its initial value... depending on points-to facts at call site!
Motivating Example

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

Although \textit{f} is conditional and loop-free, it may have very different effects at different call sites. Example: After a call to \textit{f}, the value of \texttt{*p} may be 3, 4, or remain its initial value... depending on points-to facts at call site!
Motivating Example

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

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

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 \( f \)

```
call site 1     call site 2     call site 3
  \( a \)       \( a \)       \( a \)
      \downarrow   \downarrow   \downarrow
```

⇒ Unknown number of points-to targets at call sites
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
Another difficulty: Different aliasing patterns between arguments may exist at different call sites
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
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
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.
Distinguish between two kinds of abstract memory locations:
Distinguish between two kinds of abstract memory locations:

- **Location Constants**: Model memory allocations, NULL, locations of stack variables etc.
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
foo(int* a) a ν

ν ranges over abstract memory locations at call sites of foo
Simple Example

In this context, $\nu$ stands for location constants $loc_1$ and $loc_2$.

$\nu$ ranges over abstract memory locations at call sites of $\text{foo}$

foo(int* $a$) $\text{foo}(x)$

$\nu$ $\text{foo}(x)$
If \( \nu_1 \) and \( \nu_2 \) 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
If $\nu_1$ and $\nu_2$ 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 $\nu_1$ and $\nu_2$ may represent an overlapping set of locations, updates to $\nu_1$ may affect updates to $\nu_2$
If arguments \( a \) and \( b \) are potential aliases, analyze function in two different initial configurations:

\[
\begin{align*}
\nu_1 & \quad \nu_2 \\
\end{align*}
\]

Problem: Number of alias patterns = \( n \)th Bell number (\( n \) = # of argument-reachable locations)
If arguments \( a \) and \( b \) are potential aliases, analyze function in two different initial configurations:

Problem:
Number of alias patterns = \( n \)th Bell number
\((n = \# \text{ of argument-reachable locations})\)
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 \( \frac{n^2}{2} \).
- Only need to analyze each function once.

⇒ Since we precisely account for all aliasing patterns in any context, it is safe to apply strong updates to (non-summary) location variables.
Enforcing Disjointness: Practical Solution

Encode aliasing patterns symbolically such that:

- Number of location variables, $n$, is the number of argument-reachable locations
Encode aliasing patterns symbolically such that:

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

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

⇒ Since we precisely account for all aliasing patterns in any context, it is safe to apply strong updates to (non-summary) location variables
Consider function:  \texttt{foo(int* a, int* b)}

\begin{center}
\begin{tikzpicture}
    \node[fill=yellow] (a) at (0,0) {a};
    \node[fill=yellow] (b) at (1,-1) {b};
\end{tikzpicture}
\end{center}
Consider function: \texttt{foo(int* a, int* b)}

\[
\nu_a \text{ represents points-to targets of } a \text{ in any calling context}
\]
Consider function: \( \text{foo}(\text{int* } a, \text{int* } b) \)

- \( a \rightarrow \nu_a \) with \( true \)
- \( b \rightarrow \nu_b \) with \( a \neq b \)

\( \nu_b \) represents points-to targets of \( b \) only in those contexts where \( a \) and \( b \) do not alias.
Consider function: \texttt{foo(int* a, int* b)}

\begin{itemize}
\item \(\nu_a\) also represents points-to targets of \(b\) in those contexts where \(a\) and \(b\) alias
\end{itemize}
Consider function: $\text{foo}(\text{int* a, int* b})$

\[ a \quad \text{true} \quad \nu_a \]

\[ a = b \quad a \neq b \quad \nu_a \quad \nu_b \]

$\nu_a$ also represents points-to targets of $b$ in those contexts where $a$ and $b$ alias

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

$\nu_1$

$\nu_2$

$\nu_i$

$\nu_n$
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 $\nu_i$.

\[ a_i \quad \text{points to} \quad \nu_k \quad \text{with} \quad k \leq i \quad \text{under constraint:} \]
\[ \bigwedge_{j < k} a_i \neq a_j \land a_i = a_k \]
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 $\nu_i$.
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 $\nu_i$.

Each $a_i$ points to $\nu_k$ with $k \leq i$ under constraint:

$$\bigwedge_{j < k} a_i \neq a_j \land a_i = a_k$$
Example

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

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

Diagram:

- Node `a` connected to `ν_a` with `true`.
- Node `b` connected to `ν_b` with `a ≠ b`.
- Node `a` connected to `b` with `a = b`.
Example

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

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

Observe: \( *b \) has value 1 if \( a \) and \( b \) alias
Example

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

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

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

```c
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

⇒ Modular analysis that cannot apply strong updates too imprecise!
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

⇒ More precise actually analysis runs faster
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
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

![Diagram showing Summary Size vs. Maximum depth of transitive callee for OpenSSH, LiteSQL, Inkscape, and DigiKam.]

- OpenSSH
- LiteSQL
- Inkscape
- DigiKam

Local reasoning by focusing only on externally-visible side effects.
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!