The Abstraction Vs. Approximations Dilemma in Pointer Analysis

Uday Khedker
(www.cse.iitb.ac.in/~uday)

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

Nov 2017
• **Disclaimer**: This talk is
  - not about accomplishments but about opinions, and hopes
  - an idealistic view of pointer analysis
    (the destination we wish to reach)
• **Disclaimer**: This talk is
  ▶ not about accomplishments but about opinions, and hopes
  ▶ an idealistic view of pointer analysis
    (the destination we wish to reach)

• **Outline**:
  ▶ Our Meanderings
  ▶ Some short trips
  ▶ Conclusions
Part 1

Our Meanderings
Pointer Analysis Musings

• A keynote address:

“The worst thing that has happened to Computer Science is C, because it brought pointers with it . . .”

- Frances Allen, IITK Workshop (2007)

• A couple of influential papers

  ○ Which Pointer Analysis should I Use?
    Michael Hind and Anthony Pioli. ISTAA 2000

  ○ Pointer Analysis: Haven’t we solved this problem yet?
    Michael Hind PASTE 2001
Pointer Analysis Musings

• A keynote address:

“The worst thing that has happened to Computer Science is C, because it brought pointers with it . . . ”

- Frances Allen, IITK Workshop (2007)

• A couple of influential papers

  ○ Which Pointer Analysis should I Use?
  Michael Hind and Anthony Pioli. ISTAA 2000

  ○ Pointer Analysis: Haven’t we solved this problem yet?
  Michael Hind PASTE 2001
Pointer Analysis Musings

• A keynote address:

“The worst thing that has happened to Computer Science is C, because it brought pointers with it . . .”

- Frances Allen, IITK Workshop (2007)

• A couple of influential papers

○ Which Pointer Analysis should I Use?
  Michael Hind and Anthony Pioli. ISTAA 2000

○ Pointer Analysis: Haven’t we solved this problem *yet*?
  Michael Hind PASTE 2001

○ 2017 . . 😞
The Mathematics of Pointer Analysis

In the most general situation

- Alias analysis is undecidable.
  Landi-Ryder [POPL 1991], Landi [LOPLAS 1992],
  Ramalingam [TOPLAS 1994]

- Flow insensitive alias analysis is NP-hard
  Horwitz [TOPLAS 1997]

- Points-to analysis is undecidable
  Chakravarty [POPL 2003]

*Adjust your expectations suitably to avoid disappointments!*
So what should we expect?

To quote Hind [PASTE 2001]
So what should we expect?

To quote Hind [PASTE 2001]

- “Fortunately many approximations exist”
So what should we expect?

To quote Hind [PASTE 2001]

- “Fortunately many approximations exist”
- “Unfortunately too many approximations exist!”
So what should we expect?

To quote Hind [PASTE 2001]

- “Fortunately many approximations exist”
- “Unfortunately too many approximations exist!”

*Engineering of pointer analysis is much more dominant than its science*
Pointer Analysis: Engineering or Science?

- Engineering view
  - Build quick approximations
  - The tyranny of (exclusive) OR
    Precision OR Efficiency?

- Science view
  - Build clean abstractions
  - Can we harness the Genius of AND?
    Precision AND Efficiency?
Pointer Analysis: Engineering or Science?

• Engineering view
  ▶ Build quick approximations
  ▶ The tyranny of (exclusive) OR Precision OR Efficiency?

• Science view
  ▶ Build clean abstractions
  ▶ Can we harness the Genius of AND? Precision AND Efficiency?

• Most common trend as evidenced by publications
  ▶ Build acceptable approximations guided by empirical observations
  ▶ The notion of acceptability is often constrained by beliefs rather than possibilities
Abstraction Vs. Approximation in Static Analysis

- Static analysis needs to create abstract values that represent many concrete values
- Mapping concrete values to abstract values
  - **Abstraction.**
    Deciding which properties of the concrete values are essential
    Ease of understanding, reasoning, modelling etc.
  - **Approximation.**
    Deciding which properties of the concrete values cannot be represented accurately and should be summarized
    Decidability, tractability, or efficiency and scalability
Abstraction Vs. Approximation in Static Analysis

• Abstractions
  ▶ focus on precision and conciseness of modelling
  ▶ tell us what we can ignore without being imprecise

• Approximations
  ▶ focus on efficiency and scalability
  ▶ tell us the imprecision that we have to tolerate
Abstraction Vs. Approximation in Static Analysis

- **Abstractions**
  - focus on precision and conciseness of modelling
  - tell us what we can ignore without being imprecise

- **Approximations**
  - focus on efficiency and scalability
  - tell us the imprecision that we have to tolerate

- *Build clean abstractions before surrendering to the approximations*
The Basis of My Hope

- **Common belief:**

- **The possibility that I dream of:**

- **The basis of my hope:**
The Basis of My Hope

- **Common belief:**
  Pointer information is very large

- **The possibility that I dream of:**

- **The basis of my hope:**
The Basis of My Hope

- **Common belief:**
  Pointer information is very large

- **The possibility that I dream of:**
  Precision can reduce the size of pointer information to make it far more manageable

- **The basis of my hope:**
The Basis of My Hope

- **Common belief:**
  Pointer information is very large

- **The possibility that I dream of:**
  Precision can reduce the size of pointer information to make it far more manageable

- **The basis of my hope:**
  At any program point, the usable pointer information is much smaller than the total pointer information
  
  Current methods perform many repeated and possibly avoidable computations
Why Avoid Approximations?

- Approximations may create a vicious cycle
Why Avoid Approximations?

• Approximations may create a vicious cycle

Approximation \(\rightarrow\) Imprecision \(\rightarrow\) Inefficiency

Imprecision \(\rightarrow\) May cause \(\rightarrow\) Inefficiency

Approximation \(\rightarrow\) May seem to warrant

• Two examples of inefficiency cause by approximations

- \(k\)-limited call strings may create “butterfly cycles” causing spurious fixed point computations [Hakjoo, 2010]
- Imprecision in function pointer analysis overapproximates calls may create spurious recursion in call graphs
Which Approximations Would I like to Avoid?

<table>
<thead>
<tr>
<th>Approximation</th>
<th>Admits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow insensitivity</td>
<td></td>
</tr>
<tr>
<td>Context insensitivity (or partial context sensitivity)</td>
<td></td>
</tr>
<tr>
<td>Imprecision in call graphs</td>
<td></td>
</tr>
<tr>
<td>Allocation site based heap abstraction</td>
<td></td>
</tr>
</tbody>
</table>
Which Approximations Would I like to Avoid?

<table>
<thead>
<tr>
<th>Approximation</th>
<th>Admits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow insensitivity</td>
<td>Spurious intraprocedural paths</td>
</tr>
<tr>
<td>Context insensitivity (or partial context sensitivity)</td>
<td>Spurious interprocedural paths</td>
</tr>
<tr>
<td>Imprecision in call graphs</td>
<td>Spurious call sequences</td>
</tr>
<tr>
<td>Allocation site based heap abstraction</td>
<td>Spurious paths in memory graph</td>
</tr>
</tbody>
</table>

Uday Khedker

IIT Bombay
## The Classical Precision-Efficiency Dilemma

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>Role in precision</th>
<th>Cause of inefficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Distinguishes between</td>
<td>Needs to consider</td>
</tr>
<tr>
<td>Flow sensitivity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context sensitivity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precise heap abstraction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precise call structure</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# The Classical Precision-Efficiency Dilemma

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>Role in precision</th>
<th>Cause of inefficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Distinguishes between</td>
<td>Needs to consider</td>
</tr>
<tr>
<td>Flow sensitivity</td>
<td>Information at different program points</td>
<td></td>
</tr>
<tr>
<td>Context sensitivity</td>
<td>Information in different contexts</td>
<td></td>
</tr>
<tr>
<td>Precise heap abstraction</td>
<td>Different heap locations</td>
<td></td>
</tr>
<tr>
<td>Precise call structure</td>
<td>Indirect calls made to different callees from the same program point</td>
<td></td>
</tr>
</tbody>
</table>
## The Classical Precision-Efficiency Dilemma

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>Role in precision</th>
<th>Cause of inefficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Distinguishes between</td>
<td>Needs to consider</td>
</tr>
<tr>
<td>Flow sensitivity</td>
<td>Information at different program points</td>
<td>A large number of program points</td>
</tr>
<tr>
<td>Context sensitivity</td>
<td>Information in different contexts</td>
<td>Exponentially large number of contexts</td>
</tr>
<tr>
<td>Precise heap abstraction</td>
<td>Different heap locations</td>
<td>Unbounded number of heap locations</td>
</tr>
<tr>
<td>Precise call structure</td>
<td>Indirect calls made to different callees from the same program point</td>
<td>Precise points-to information</td>
</tr>
</tbody>
</table>
Flow Insensitivity in Data Flow Analysis

- Assumption: Statements can be executed in any order.

- Instead of computing point-specific data flow information, summary data flow information is computed.
  The summary information is required to be a safe approximation of point-specific information for each point.

- No data flow information is killed
  If a statement kills data flow information, there is an alternate path that excludes the statement.
Flow Insensitivity in Data Flow Analysis

Start

0 \( f_0 \)

1 \( f_1 \)

2 \( f_2 \)

3 \( f_3 \)

\( \cdots \)

\( i \) \( f_i \)

\( \cdots \)

m \( f_m \)

End
Flow Insensitivity in Data Flow Analysis

Start

\[ f_0 \]
\[ f_1 \]
\[ f_2 \]
\[ f_3 \]
\[ f_i \]
\[ f_m \]

End

0 1 2 3 \[ \ldots \] \[ i \] \[ \ldots \] m
Flow Insensitivity in Data Flow Analysis

Allows arbitrary compositions of flow functions in any order
⇒ Flow insensitivity
Flow Insensitivity in Data Flow Analysis

In practice, dependent constraints are collected in a global repository in one pass and then are solved independently.
If I am Allowed to Nitpick...

- Context sensitivity should involve all of the following

  [A] Full context sensitivity regardless of the call depth even in recursion
  [B] Ability to store data flow information parameterized by contexts at each program point
  [C] Flow sensitivity at the intraprocedural level (otherwise distinct calls to the same procedure within a procedure cannot be distinguished)
If I am Allowed to Nitpick . . .

• Context sensitivity should involve all of the following

[A] Full context sensitivity regardless of the call depth even in recursion
[B] Ability to store data flow information parameterized by contexts at each program point
[C] Flow sensitivity at the intraprocedural level (otherwise distinct calls to the same procedure within a procedure cannot be distinguished)

• In particular

- $k$-limiting violates [A]
- Treating recursion as an SCC violates [A]
- Functional approaches violate [B]
- Object sensitivity violates [C]
If I am Allowed to Nitpick . . .

- Context sensitivity should involve all of the following
  
  **[A]** Full context sensitivity regardless of the call depth even in recursion
  
  **[B]** Ability to store data flow information parameterized by contexts at each program point
  
  **[C]** Flow sensitivity at the intraprocedural level (otherwise distinct calls to the same procedure within a procedure cannot be distinguished)

- In particular
  
  - \(k\)-limiting violates [A]
  
  - Treating recursion as an SCC violates [A]
  
  - Functional approaches violate [B]
  
  - Object sensitivity violates [C]

- Object sensitivity can be completely modelled by calling context sensitivity
  
  - by a flow sensitive propagation of values representing objects, and
  
  - identifying a procedure by an \((\text{object}, \text{procedure})\) pair, and
  
  - identifying a context by a call site and the pairs defined as above
Context Sensitivity in Interprocedural Analysis

\[ x' = f_r(x) \quad y' = f_r(y) \]
Context Sensitivity in Interprocedural Analysis

\[ S_s \xrightarrow{c_i} C_i \xrightarrow{R_i} E_s \]
\[ S_r \xrightarrow{x} C_j \xrightarrow{R_j} E_t \]
\[ f_r \]

\[ S_t \xrightarrow{y} C_j \xrightarrow{R_j} E_t \]
\[ y' \]

Uday Khedker
IIT Bombay
Context Sensitivity in Interprocedural Analysis
Context Sensitivity in Interprocedural Analysis
Context Sensitivity in Interprocedural Analysis
Context Sensitivity in the Presence of Recursion

\[
\begin{align*}
\text{Start}_s & \quad \rightarrow \quad \text{call} \quad \rightarrow \quad C_i \\
\text{stop calling} & \quad \rightarrow \quad \text{R}_i \\
\text{return} & \quad \rightarrow \quad \text{End}_s
\end{align*}
\]
Context Sensitivity in the Presence of Recursion

- Paths from $Start_s$ to $End_s$ should constitute a context free language:
  \[ \text{call}^n \cdot \text{stop} \cdot \text{return}^n \]

- If we treat cycle of recursion as an SCC:
  - Calls and returns become jumps, and
  - Paths are approximated by a regular language:
    \[ \text{call}^* \cdot \text{stop} \cdot \text{return}^* \]
Context Insensitivity = Imprecision + Potential Inefficiency

\text{Start}_{\text{main}}

\begin{align*}
a &= 1\\
\text{Call } P
\end{align*}

\text{End}_{\text{main}}

\text{Start}_{\text{p}}

\begin{align*}
a &= a + 1\\
\text{End}_{\text{p}}
\end{align*}
Context Insensitivity $= \text{Imprecision} + \text{Potential Inefficiency}$

- What is the value range of $a$?
Context Insensitivity = Imprecision + Potential Inefficiency

• What is the value range of $a$?
Start_{main}

\[ a = 1 \]

Call \( P \)

\( (2, 2) \)

Start_{p}

\[ a = a + 1 \]

Call \( P \)

\( (2, 2) \)

End_{p}

\( (1, 1) \)

End_{main}

- What is the value range of \( a \)?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of \( P \)
Context Insensitivity = Imprecision + Potential Inefficiency

- What is the value range of $a$?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of $P$
Context Insensitivity = Imprecision + Potential Inefficiency

- What is the value range of $a$?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of $P$
  - Range of $a$ at $End_{main}$ is $(3, 3)$
Context Insensitivity = Imprecision + Potential Inefficiency

- What is the value range of $a$?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of $P$
  - Range of $a$ at $End_{main}$ is $(3, 3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller
Context Insensitivity = Imprecision + Potential Inefficiency

```
Start_{main}

a = 1

Call P

(2, 2)

End_{main}
```

```
Start_p

a = a + 1

(1, 2)

(1, 1)

Call P

(2, 3)

End_p
```

- What is the value range of \( a \)?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of \( P \)
  - Range of \( a \) at \( End_{main} \) is \((3, 3)\)
- Context insensitive analysis
  - Data flow value propagated back to every caller
Context Insensitivity = Imprecision + Potential Inefficiency

- What is the value range of $a$?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of $P$
  - Range of $a$ at $End_{main}$ is $(3, 3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller
Context Insensitivity = Imprecision + Potential Inefficiency

What is the value range of \( a \)?

- Context sensitive analysis
  - Data flow value propagated back to the current caller of \( P \)
  - Range of \( a \) at \( \text{End}_{main} \) is (3, 3)

- Context insensitive analysis
  - Data flow value propagated back to every caller

\[
\begin{align*}
\text{Start}_{main} & \\
a = 1 & \\
\text{Call } P & \\
(2, 3) & \\
\text{Start}_p & \\
a = a+1 & \\
(1, 3) & \\
\text{Call } P & \\
(1, 1) & \\
\text{End}_p & \\
(2, 4) & \\
\text{End}_{main} &
\end{align*}
\]
Context Insensitivity = Imprecision + Potential Inefficiency

What is the value range of $a$?

- Context sensitive analysis
  - Data flow value propagated back to the current caller of $P$
  - Range of $a$ at $End_{main}$ is $(3, 3)$

- Context insensitive analysis
  - Data flow value propagated back to every caller
Context Insensitivity = Imprecision + Potential Inefficiency

Start_{main}

\[ a = 1 \]

Call \( P \)

\((2, 4)\)

\[ a = a + 1 \]

End_{\text{main}}

\[ (1, 4) \]

Start_{p}

\[ (1, 1) \]

End_{p}

\[ (2, 5) \]

• What is the value range of \( a \)?

• Context sensitive analysis
  ▶ Data flow value propagated back to the current caller of \( P \)
  ▶ Range of \( a \) at End_{\text{main}} is \((3, 3)\)

• Context insensitive analysis
  ▶ Data flow value propagated back to every caller
**Context Insensitivity = Imprecision + Potential Inefficiency**

- What is the value range of \( a \)?
- Context sensitive analysis
  - Data flow value propagated back to the *current* caller of \( P \)
  - Range of \( a \) at \( \text{End}_{main} \) is \((3, 3)\)
- Context insensitive analysis
  - Data flow value propagated back to every caller
  - Range of \( a \) at \( \text{End}_{main} \) is \((2, \ldots)\)
Context Insensitivity $=$ Imprecision $+$ Potential Inefficiency

- What is the value range of $a$?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of $P$
  - Range of $a$ at $\text{End}_{\text{main}}$ is $(3, 3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller
  - Range of $a$ at $\text{End}_{\text{main}}$ is $(2, \ldots)$
**Context Insensitivity = Imprecision + Potential Inefficiency**

- What is the value range of $a$?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of $P$
  - Range of $a$ at $End_{main}$ is $(3, 3)$
- Context insensitive analysis
  - Data flow value propagated back to every caller
  - Range of $a$ at $End_{main}$ is $(2, \ldots)$
- *Spurious interprocedural loops*
Context Insensitivity = Imprecision + Potential Inefficiency

• What is the value range of $a$?
• Context sensitive analysis
  ▶ Data flow value propagated back to the current caller of $P$
  ▶ Range of $a$ at $End_{main}$ is $(3, 3)$
• Context insensitive analysis
  ▶ Data flow value propagated back to every caller
  ▶ Range of $a$ at $End_{main}$ is $(2, \ldots)$
• Spurious interprocedural loops
• Spurious fixed point computations
Context Sensitivity in the Presence of Recursion

Start\textsubscript{s}

\begin{itemize}
  \item call
  \item stop calling
\end{itemize}

C\textsubscript{i}

\begin{itemize}
  \item return
\end{itemize}

R\textsubscript{i}

End\textsubscript{s}
Context Sensitivity in the Presence of Recursion

- Paths from $Start_s$ to $End_s$ should constitute a context free language

\[\text{call}^n \cdot \text{stop} \cdot \text{return}^n\]

- If we treat cycle of recursion as an SCC
  - Calls and returns become jumps, and
  - paths are approximated by a regular language

\[\text{call}^\ast \cdot \text{stop} \cdot \text{return}^\ast\]
Pointer Analysis: An Engineer’s Landscape

Flow Sensitivity Increases

Context Sensitivity Increases

FS

FS_{NoKill}

\( F_I \subseteq \)

\( F_I^{SSA} \)

\( F_I \equiv \)

Cl

CS_{ObjSens}

CS_{Reclns}

CS
Pointer Analysis: An Engineer’s Landscape

Data Structures: BDDs, probabilistic

Flow Sensitivity Increases

Context Sensitivity Increases

FS

FS_{NoKill}

FI_{SSA}

FI_{c}

FI_{=}
Pointer Analysis: An Engineer’s Landscape

Methods: parallel, on demand, randomized

Data Structures: BDDs, probabilistic

Flow Sensitivity Increases

FS

FS\textsubscript{NoKill}

FI\textsubscript{SSA}

FI\subseteq

FI\textsubscript{\equiv}

CI

CS\textsubscript{ObjSens}

CS\textsubscript{Reclns}

CS

Context Sensitivity Increases
Pointer Analysis: An Engineer’s Landscape

Refinement: Level-wise, bootstrapping

Methods: parallel, on demand, randomized

Data Structures: BDDs, probabilistic

Flow Sensitivity Increases

Context Sensitivity Increases

$F_{\subseteq}$

$F_{SSA}$

$F_{\text{NoKill}}$

$FS$

Context Sensitivity

$CS_{\text{ObjSens}}$

$CS_{\text{Reclns}}$

$CS$

Flow Sensitivity

$FI_{\subseteq}$

$FI_{SSA}$

$FI_{\text{NoKill}}$
Pointer Analysis: An Engineer’s Landscape

Flow Sensitivity Increases

Crowded Area

Context Sensitivity Increases
Pointer Analysis: An Engineer’s Landscape

Flow Sensitivity Increases

$FI_{\subseteq}$

$FI_{SSA}$

$FS_{NoKill}$

$FS$

Crowded Area

Context Sensitivity Increases

CI

$CS_{ObjSens}$

$CS_{Reclns}$

CS

Thinly populated
Pointer Analysis: An Engineer’s Landscape

\[ \text{FI} \subseteq \text{FI}_{\text{SSA}} \]

Flow Sensitivity Increases

Context Sensitivity Increases

That’s the corner we are trying to occupy :-)

Uday Khedker IIT Bombay
### In Search of Abstractions for Precision Without Inefficiency

<table>
<thead>
<tr>
<th>Desired Abstraction</th>
<th>Enabling Abstraction</th>
<th>Status of our work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow sensitivity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context sensitivity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Caller sensitivity)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precise heap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>abstraction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precise call</td>
<td></td>
<td></td>
</tr>
<tr>
<td>structure</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Uday Khedker, IIT Bombay*
<table>
<thead>
<tr>
<th>Desired Abstraction</th>
<th>Enabling Abstraction</th>
<th>Status of our work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow sensitivity</td>
<td>Joint liveness and points-to analysis</td>
<td>Partial accomplishment (SAS12)</td>
</tr>
<tr>
<td>Context sensitivity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Caller sensitivity)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precise heap abstraction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precise call structure</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## In Search of Abstractions for Precision Without Inefficiency

<table>
<thead>
<tr>
<th>Desired Abstraction</th>
<th>Enabling Abstraction</th>
<th>Status of our work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow sensitivity</td>
<td>Joint liveness and points-to analysis</td>
<td>Partial accomplishment (SAS12)</td>
</tr>
<tr>
<td></td>
<td>High level abstraction of memory</td>
<td>Partial accomplishment (SAS16)</td>
</tr>
<tr>
<td>Context sensitivity (Caller sensitivity)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precise heap abstraction</td>
<td>Postpone low level connections explicated by the classical points-to facts</td>
<td></td>
</tr>
<tr>
<td>Precise call structure</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### In Search of Abstractions for Precision Without Inefficiency

<table>
<thead>
<tr>
<th>Desired Abstraction</th>
<th>Enabling Abstraction</th>
<th>Status of our work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow sensitivity</td>
<td>Joint liveness and points-to analysis</td>
<td>Partial accomplishment (SAS12)</td>
</tr>
<tr>
<td></td>
<td>High level abstraction of memory</td>
<td>Partial accomplishment (SAS16)</td>
</tr>
<tr>
<td>Context sensitivity (Caller sensitivity)</td>
<td>Value contexts</td>
<td>Mature accomplishment (CC08, SAS12, SOAP13)</td>
</tr>
<tr>
<td>Precise heap abstraction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precise call structure</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Distinguish between contexts by their data flow values and not their call chains
## In Search of Abstractions for Precision Without Inefficiency

<table>
<thead>
<tr>
<th>Desired Abstraction</th>
<th>Enabling Abstraction</th>
<th>Status of our work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow sensitivity</td>
<td>Joint liveness and points-to analysis</td>
<td>Partial accomplishment</td>
</tr>
<tr>
<td></td>
<td>High level abstraction of memory</td>
<td></td>
</tr>
<tr>
<td>Context sensitivity</td>
<td>Value contexts</td>
<td>Mature accomplishment</td>
</tr>
<tr>
<td>(Caller sensitivity)</td>
<td>GPG based bottom-up summary flow functions</td>
<td>(CC08, SAS12, SOAP13)</td>
</tr>
<tr>
<td>Precise heap abstraction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precise call structure</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Avoid recomputations for each context.
- Use a higher level abstraction of memory.
## In Search of Abstractions for Precision Without Inefficiency

<table>
<thead>
<tr>
<th>Desired Abstraction</th>
<th>Enabling Abstraction</th>
<th>Status of our work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow sensitivity</td>
<td>Joint liveness and points-to analysis</td>
<td>Partial accomplishment (SAS12)</td>
</tr>
<tr>
<td></td>
<td>High level abstraction of memory</td>
<td>Partial accomplishment (SAS16)</td>
</tr>
<tr>
<td>Context sensitivity (Caller sensitivity)</td>
<td>Value contexts</td>
<td>Mature accomplishment (CC08, SAS12, SOAP13)</td>
</tr>
<tr>
<td></td>
<td>GPG based bottom-up summary flow functions</td>
<td>Mature accomplishment (SAS16)</td>
</tr>
<tr>
<td>Precise heap abstraction</td>
<td>Liveness access graphs</td>
<td>Partial accomplishment (TOPLAS07)</td>
</tr>
<tr>
<td>Precise call structure</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## In Search of Abstractions for Precision Without Inefficiency

<table>
<thead>
<tr>
<th>Desired Abstraction</th>
<th>Enabling Abstraction</th>
<th>Status of our work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow sensitivity</td>
<td>Joint liveness and points-to analysis</td>
<td>Partial accomplishment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(SAS12)</td>
</tr>
<tr>
<td></td>
<td>High level abstraction of memory</td>
<td>Partial accomplishment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(SAS16)</td>
</tr>
<tr>
<td>Context sensitivity (Caller sensitivity)</td>
<td>Value contexts</td>
<td>Mature accomplishment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(CP13)</td>
</tr>
<tr>
<td></td>
<td>GPG based bottom up summary flow functions</td>
<td>Mature accomplishment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(CP13)</td>
</tr>
<tr>
<td>Precise heap abstraction</td>
<td>Liveness access graphs</td>
<td>Mature accomplishment</td>
</tr>
<tr>
<td></td>
<td>Access based abstraction</td>
<td>(ISMM17)</td>
</tr>
<tr>
<td>Precise call structure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Desired Abstraction</td>
<td>Enabling Abstraction</td>
<td>Status of our work</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------------------------------------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>Flow sensitivity</td>
<td>Joint liveness and points-to analysis</td>
<td>Partial accomplishment (SAS12)</td>
</tr>
<tr>
<td></td>
<td>High level abstraction of memory</td>
<td>Partial accomplishment (SAS16)</td>
</tr>
<tr>
<td>Context sensitivity</td>
<td>Value contexts</td>
<td>Mature accomplishment (CC08, SAS12, SOAP13)</td>
</tr>
<tr>
<td>(Caller sensitivity)</td>
<td>GPG based bottom-up summary flow functions</td>
<td>Mature accomplishment (SAS16)</td>
</tr>
<tr>
<td>Precise heap abstraction</td>
<td>Liveness access graphs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Access based abstraction</td>
<td></td>
</tr>
<tr>
<td>Precise call structure</td>
<td>Call strings record call <em>history</em>. We need to record call <em>future</em> also.</td>
<td>Work in progress</td>
</tr>
</tbody>
</table>
## In Search of Abstractions for Precision Without Inefficiency

<table>
<thead>
<tr>
<th>Desired Abstraction</th>
<th>Enabling Abstraction</th>
<th>Status of our work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow sensitivity</td>
<td>Joint liveness and points-to analysis</td>
<td>Partial accomplishment (SAS12)</td>
</tr>
<tr>
<td></td>
<td>High level abstraction of memory</td>
<td>Partial accomplishment (SAS16)</td>
</tr>
<tr>
<td>Context sensitivity</td>
<td>Value contexts</td>
<td>Mature accomplishment (CC08, SAS12, SOAP13)</td>
</tr>
<tr>
<td>(Caller sensitivity)</td>
<td>GPG based bottom-up summary flow functions</td>
<td>Mature accomplishment (SAS16)</td>
</tr>
<tr>
<td>Precise heap abstraction</td>
<td>Liveness access graphs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Access based abstraction</td>
<td></td>
</tr>
<tr>
<td>Precise call structure</td>
<td>Callee sensitivity</td>
<td>Work in progress</td>
</tr>
<tr>
<td></td>
<td>Virtual call resolution</td>
<td>Work in progress</td>
</tr>
</tbody>
</table>

Make the call graph more precise by computing a more precise set of callees
## In Search of Abstractions for Precision Without Inefficiency

<table>
<thead>
<tr>
<th>Desired Abstraction</th>
<th>Enabling Abstraction</th>
<th>Status of our work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow sensitivity</td>
<td>Joint liveness and points-to analysis</td>
<td>Partial accomplishment (SAS12)</td>
</tr>
<tr>
<td></td>
<td>High level abstraction of memory</td>
<td>Partial accomplishment (SAS16)</td>
</tr>
<tr>
<td>Context sensitivity</td>
<td>Context sensitivity (Caller sensitivity)</td>
<td>Mature accomplishment (CC08, SAS12, SOAP13)</td>
</tr>
<tr>
<td></td>
<td>Value contexts</td>
<td>Mature accomplishment (SAS16)</td>
</tr>
<tr>
<td>Precise heap abstraction</td>
<td>Access abstraction</td>
<td>Mature accomplishment (TOPLAS07)</td>
</tr>
<tr>
<td>Precise call structure</td>
<td>Callee sensitivity</td>
<td>Work in progress</td>
</tr>
<tr>
<td></td>
<td>Virtual call resolution</td>
<td>Work in progress</td>
</tr>
</tbody>
</table>

*We are destined to a long haul with no guarantees :-)*

Uday Khedker
IIT Bombay
Part 2

Some Short Trips
## In Search of Abstractions for Precision Without Inefficiency

<table>
<thead>
<tr>
<th>Desired Abstraction</th>
<th>Enabling Abstraction</th>
<th>Status of our work</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Flow sensitivity</strong></td>
<td>Joint liveness and points-to analysis</td>
<td>Partial accomplishment (SAS12)</td>
</tr>
<tr>
<td></td>
<td>High level abstraction of memory</td>
<td>Partial accomplishment (SAS16)</td>
</tr>
<tr>
<td><strong>Context sensitivity (Caller sensitivity)</strong></td>
<td>Value contexts</td>
<td>Mature accomplishment (CC08, SAS12, SOAP13)</td>
</tr>
<tr>
<td></td>
<td>GPG based bottom-up summary flow functions</td>
<td>Mature accomplishment (SAS16)</td>
</tr>
<tr>
<td><strong>Precise heap abstraction</strong></td>
<td>Liveness access graphs</td>
<td>Partial accomplishment (TOPLAS07)</td>
</tr>
<tr>
<td></td>
<td>Access based abstraction</td>
<td>Mature accomplishment (ISMM17)</td>
</tr>
<tr>
<td><strong>Precise call structure</strong></td>
<td>Callee sensitivity</td>
<td>Work in progress</td>
</tr>
<tr>
<td></td>
<td>Virtual call resolution</td>
<td>Work in progress</td>
</tr>
</tbody>
</table>
Liveness Based Pointer Analysis: Motivation

$n_1$

$x = \& y$
$y = \& z$
$z = \& u$

$n_2$

$\star z = y$

$n_3$

$z = y$

$n_4$

$y = \& x$

use $u$

use $x$
Liveness Based Pointer Analysis: Motivation

Is all this information really useful?

\[
\begin{align*}
  x &= \& y \\
  y &= \& z \\
  z &= \& u \\
\end{align*}
\]

\[
\begin{align*}
  y &= \& x \\
  \text{use } u \\
  \text{use } x \\
\end{align*}
\]

Uday Khedker
IIT Bombay
Liveness Based Pointer Analysis: Motivation

\[ x = &y \]
\[ y = &z \]
\[ z = &u \]

\[ *z = y \]

\[ y = &x \]
use \( u \)
use \( x \)

\[ \emptyset \]
Liveness Based Pointer Analysis: Motivation

\[ x = & y \\
y = & z \\
z = & u \]

\[ n_1 \]

\[ *z = y \]

\[ n_2 \]

\[ z = y \]

\[ n_3 \]

\[ y = & x \\
use u \\
use x \]

\[ n_4 \]
Liveness Based Pointer Analysis: Motivation

\[
\begin{align*}
  x &= \& y \\
  y &= \& z \\
  z &= \& u \\
  \end{align*}
\]

\[
\begin{align*}
  n_1 & : (x, y, z, u) \\
  n_2 & : (x, y, z, u) \\
  n_3 & : (x, y) \\
  n_4 & : (x, y, z, u) \\
  \end{align*}
\]
Liveness Based Pointer Analysis: Motivation

\[\begin{align*}
  x &= \& y \\
  y &= \& z \\
  z &= \& u
\end{align*}\]
Liveness Based Pointer Analysis: Motivation

\[ x \equiv \& y \\
y \equiv \& z \\
z \equiv \& u \]

\[ *z = y \]

\[ y \equiv \& x \\
use u \\
use x \]

\[ x \equiv y \equiv z \equiv u \]

\[ n_1 \]

\[ n_2 \]

\[ n_3 \]

\[ n_4 \]

\[ \emptyset \]
Liveness Based Points-to Analysis (SAS-2012)

- Mutual dependence of liveness and points-to information
  - Define points-to information only for live pointers
  - For pointer indirections, define liveness information using points-to information

- Use call strings method for full flow and context sensitivity
  - Value based termination of call strings construction (CC-2008)

- Use strong liveness
Liveness Based Interprocedural Points-to Analysis: Empirical Measurements

- Observations on SPEC CPU 2006 benchmarks in GCC 4.6.0 (Prashant Singh Rawat, IITB 2012)

Usable pointer information is small and sparse

<table>
<thead>
<tr>
<th>No of Points-to pairs</th>
<th>Percentable of basic blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64-96%</td>
</tr>
<tr>
<td>1-4</td>
<td>9-25%</td>
</tr>
<tr>
<td>5-8</td>
<td>0-10%</td>
</tr>
<tr>
<td>8+</td>
<td>0-4%</td>
</tr>
</tbody>
</table>
Liveness Based Interprocedural Points-to Analysis: Empirical Measurements

- Observations on SPEC CPU 2006 benchmarks in GCC 4.6.0 (Prashant Singh Rawat, IITB 2012)

Usable pointer information is small and sparse

<table>
<thead>
<tr>
<th>No of Points-to pairs</th>
<th>Percentable of basic blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64-96%</td>
</tr>
<tr>
<td>1-4</td>
<td>9-25%</td>
</tr>
<tr>
<td>5-8</td>
<td>0-10%</td>
</tr>
<tr>
<td>8+</td>
<td>0-4%</td>
</tr>
</tbody>
</table>
Liveness Based Interprocedural Points-to Analysis: Empirical Measurements

- Observations on SPEC CPU 2006 benchmarks in GCC 4.6.0 (Prashant Singh Rawat, IITB 2012)

Usable pointer information is small and sparse

<table>
<thead>
<tr>
<th>No of Points-to pairs</th>
<th>Percentable of basic blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64-96%</td>
</tr>
<tr>
<td>1-4</td>
<td>9-25%</td>
</tr>
<tr>
<td>5-8</td>
<td>0-10%</td>
</tr>
<tr>
<td>8+</td>
<td>0-4%</td>
</tr>
</tbody>
</table>

- Independently implemented and verified in
  - LLVM (Dylan McDermott, Cambridge, 2016) and
  - GCC 4.7.2 (Priyanka Sawant, IITB, 2016)

Multiple interprocedural paths reaching the procedure

Start

End

Multiple interprocedural paths reaching the procedure

Multiple interprocedural paths reaching the procedure

Multiple interprocedural paths reaching the procedure

Multiple interprocedural paths reaching the procedure

Start

End

Data flow values

Start

End

\( x \)

\( y \)

\( y' \)

\( x' \)

\( y' \)

Data flow values $x$, $y$, $y'$, $y'$, $z$, $z'$

Contexts of the callers

Call sites

Data flow values

Contexts

Start

End

Uday Khedker

Contexts of the callers

Call sites

Data flow values

Contexts

Start

End

Context transition graph

\[ X_i \xrightarrow{c_0} X_0 \]

Context transition graph:
- \( X_i \) \( \xrightarrow{c_0} \) \( X_0 \)
- \( X_j \) \( \xrightarrow{c_1} \)
- \( X_k \) \( \xrightarrow{c_2} \)
- \( X_l \) \( \xrightarrow{c_3} \)

Data flow values:
- Contexts of the callers
- Call sites
- Contexts

Contexts of the callers

Call sites

Data flow values

Context transition graph

Analyze a procedure once for an input data flow value

- The number of times a procedure is analyzed reduces dramatically
- Similar to the tabulation based method of functional approach [Sharir-Pnueli, 1981]

However,

- Value contexts record calling contexts too
  - Useful for context matching across program analyses
- Can avoid some reprocessing even when a new input value is found
Empirical Observations About Value Contexts

- Reaching definitions analysis in GCC 4.2.0 (CC-2008)
  
  Analysis of Towers of Hanoi
  
  - Time brought down from $3.973 \times 10^6$ ms to 2.37 ms
  - No of call strings brought down from $10^6+$ to 8
Empirical Observations About Value Contexts

- Reaching definitions analysis in GCC 4.2.0 (CC-2008)
  Analysis of Towers of Hanoi
  - Time brought down from $3.973 \times 10^6$ ms to 2.37 ms
  - No of call strings brought down from $10^6+$ to 8

- Generic Interprocedural Analysis Framework in SOOT (SOAP-2013)
  Empirical observations on SPECJVM98 and DaCapo 2006 benchmarks for on-the-fly call graph construction
  - Average number of contexts per procedure lies in the range 4-25
  - Much fewer long call chains than in the default call graph constructed using SPARK
    For length 7, less than 50%
    For length 10, less than 5%
Classical Points-to Facts: A Low Level Abstraction of Memory for Points-to Analysis

All variables are global

Red nodes are known named locations
Classical Points-to Facts: A Low Level Abstraction of Memory for Points-to Analysis

All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

f()
{
    *x = y
}

x \rightarrow \phi_1 \rightarrow \phi_2 \rightarrow y
Classical Points-to Facts: A Low Level Abstraction of Memory for Points-to Analysis

```c
f()
{
    *x = y
}
```

All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations
Classical Points-to Facts: A Low Level Abstraction of Memory for Points-to Analysis

```c
f()
{
    *x = y
}
```

All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations
Classical Points-to Facts: A Low Level Abstraction of Memory for Points-to Analysis

All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations
Classical Points-to Facts: A Low Level Abstraction of Memory for Points-to Analysis

All variables are global
Red nodes are known named locations
Blue nodes are placeholders denoting unknown locations
Classical Points-to Facts: A Low Level Abstraction of Memory for Points-to Analysis

All variables are global
Red nodes are known named locations
Blue nodes are placeholders denoting unknown locations

```
f()
{
    *x = y
}
```
Classical Points-to Facts: A Low Level Abstraction of Memory for Points-to Analysis

All variables are global

Red nodes are known named locations
Blue nodes are placeholders denoting unknown locations
Generalized Points-to Facts: A High Level Abstraction of Memory for Points-to Analysis (SAS-2016)

Blue arrows are low level view of memory in terms of classical points-to facts

f() {
    *x = y
}

x → φ₁ → φ₂ → y
Generalized Points-to Facts: A High Level Abstraction of Memory for Points-to Analysis (SAS-2016)

Blue arrows are low level view of memory in terms of classical points-to facts
Black arrows are high level view of memory in terms of generalized points-to facts

```c
f()
{
    *x = y
}
```
Generalized Points-to Facts: A High Level Abstraction of Memory for Points-to Analysis (SAS-2016)

f()
{
    *x = y
}

Blue arrows are low level view of memory in terms of classical points-to facts
Black arrows are high level view of memory in terms of generalized points-to facts
Blue arrows are low level view of memory in terms of classical points-to facts
Black arrows are high level view of memory in terms of generalized points-to facts
Blue arrows are low level view of memory in terms of classical points-to facts
Black arrows are high level view of memory in terms of generalized points-to facts
Generalized Points-to Facts: A High Level Abstraction of Memory for Points-to Analysis (SAS-2016)

```c
f()
{
    *x = y
}
```

Blue arrows are low level view of memory in terms of classical points-to facts
Black arrows are high level view of memory in terms of generalized points-to facts
Generalized Points-to Facts: A High Level Abstraction of Memory for Points-to Analysis (SAS-2016)

Blue arrows are low level view of memory in terms of classical points-to facts
Black arrows are high level view of memory in terms of generalized points-to facts

```plaintext
f()
{
    *x = y
}
```
Generalized Points-to Facts: A High Level Abstraction of Memory for Points-to Analysis (SAS-2016)

Blue arrows are low level view of memory in terms of classical points-to facts
Black arrows are high level view of memory in terms of generalized points-to facts
Generalized Points-to Graphs (GPGs) for Points-to Analysis (SAS-2016)

Construction of bottom up summary flow functions using GPGs

- **Issues at intraprocedural level**
  Flow sensitivity, strong and weak updates, efficiency using SSA form

- **Issues at interprocedural level**
  Context sensitivity: Composition of callee’s GPGs within callers
  Efficiency using bypassing of irrelevant information

- **Handling advanced features**
  Function Pointers, Heap, Structures, Union, Arrays, Pointer Arithmetic

- **Theoretical issues.** Soundness and complexity

- **Implementation and measurements**
  Using LTO framework in GCC 4.7.2 scaling to 158 KLoC
Heap Reference Analysis [TOPLAS 2007]

- Problem.

- Our Objectives.

- Main Challenge.

- Our Key Idea.

- Current status.

- Further Work.
Heap Reference Analysis [TOPLAS 2007]

• Problem. A lot of unused data remains unclaimed even in the best of garbage collectors. In C/C++, memory leaks is a major problem

• Our Objectives.

• Main Challenge.

• Our Key Idea.

• Current status.

• Further Work.
Heap Reference Analysis [TOPLAS 2007]

- **Problem.** A lot of unused data remains unclaimed even in the best of garbage collectors. In C/C++, memory leaks is a major problem.

- **Our Objectives.** Static analysis of heap data to improve garbage collection and plug memory leaks.

- **Main Challenge.**

- **Our Key Idea.**

- **Current status.**

- **Further Work.**
Heap Reference Analysis [TOPLAS 2007]

- **Problem.** A lot of unused data remains unclaimed even in the best of garbage collectors. In C/C++, memory leaks is a major problem.

- **Our Objectives.** Static analysis of heap data to improve garbage collection and plug memory leaks.

- **Main Challenge.** Unlike stack and static data,
  - heap data accessible to any procedure is unbounded. Hence,
  - the mapping between object names and their addresses needs to change at runtime.

- **Our Key Idea.**

- **Current status.**

- **Further Work.**
Which Heap Memory Nodes Can be Statically Marked as Live?

If the while loop is not executed even once.

1. \( w = x \)  // \( x \) points to \( m_a \)
2. while \( (x.data < \text{max}) \)
3. \( x = x.rptr \)
4. \( y = x.lptr \)
5. \( z = \text{New class of } z \)
6. \( y = y.lptr \)
7. \( z.sum = x.data + y.data \)
Which Heap Memory Nodes Can be Statically Marked as Live?

If the while loop is executed once.

1. \( w = x \)  // \( x \) points to \( m_a \)
2. \( \text{while} \ (x\.\text{data} < \text{max}) \)
3. \( x = x\.\text{rptr} \)
4. \( y = x\.\text{lptr} \)
5. \( z = \text{New class of } z \)
6. \( y = y\.\text{lptr} \)
7. \( z\.\text{sum} = x\.\text{data} + y\.\text{data} \)
Which Heap Memory Nodes Can be Statically Marked as Live?

If the while loop is executed twice.

1. \( w = x \) // \( x \) points to \( m \)
2. while \( (x.data < \text{max}) \)
3. \( x = x.rptr \)
4. \( y = x.lptr \)
5. \( z = \text{New class of } z \)
6. \( y = y.lptr \)
7. \( z.\text{sum} = x.data + y.data \)
Heap Reference Analysis [TOPLAS 2007]

- **Problem.** A lot of unused data remains unclaimed even in the best of garbage collectors. In C/C++, memory leaks is a major problem.

- **Our Objectives.** Static analysis of heap data to improve garbage collection and plug memory leaks.

- **Main Challenge.** Unlike stack and static data,
  - heap data accessible to any procedure is unbounded. Hence,
  - the mapping between object names and their addresses needs to change at runtime.

- **Our Key Idea.**

- **Current status.**

- **Further Work.**
Heap Reference Analysis [TOPLAS 2007]

- **Problem.** A lot of unused data remains unclaimed even in the best of garbage collectors. In C/C++, memory leaks is a major problem.

- **Our Objectives.** Static analysis of heap data to improve garbage collection and plug memory leaks.

- **Main Challenge.** Unlike stack and static data,
  - heap data accessible to any procedure is unbounded. Hence,
  - the mapping between object names and their addresses needs to change at runtime.

- **Our Key Idea.** Build bounded abstractions of heap data in terms of graphs and perform analysis using these graphs as data flow values.

- **Current status.**

- **Further Work.**
Heap Reference Analysis: Our Solution

```plaintext
y = z = null
1 w = x
   w = null
2 while (x.data < max)
   {
      x.lptr = null
      x = x.rptr
   }
3 x.rptr = x.lptr.rptr = null
   x.lptr.lptr.lptr = null
   x.lptr.lptr.rptr = null
4 y = x.lptr
   x.lptr = y.rptr = null
   y.lptr.lptr = y.lptr.rptr = null
5 z = New class of z
   z.lptr = z.rptr = null
6 y = y.lptr
   y.lptr = y.rptr = null
7 z.sum = x.data + y.data
   x = y = z = null
```

While loop is not executed even once
Heap Reference Analysis: Our Solution

```
y = z = null
1 w = x
   w = null
2 while (x.data < max)
   { x.lptr = null
     x = x.rptr
   }
   x.rptr = x.lptr.rptr = null
   x.lptr.lptr.lptr = null
   x.lptr.lptr.rptr = null
3 y = x.lptr
   x.lptr = y.rptr = null
   y.lptr.lptr = y.lptr.rptr = null
4 z = New class
   z.lptr = z.rptr = null
5 y = y.lptr
   y.lptr = y.rptr = null
6 z.sum = x.data + y.data
7 x = y = z = null
```

While loop is not executed even once
Heap Reference Analysis: Our Solution

```
y = z = null
1 w = x
    w = null
2 while (x.data < max)
    { x.lptr = null
3      x = x.rptr
      x.rptr = x.lptr.rptr = null
      x.lptr.lptr.lptr = null
      x.lptr.lptr.rptr = null
4      y = x.lptr
      x.lptr = y.rptr = null
      y.lptr.lptr = y.lptr.rptr = null
5      z = New class of z
      z.lptr = z.rptr = null
6    y = y.lptr
      y.lptr = y.rptr = null
7    z.sum = x.data + y.data
    x = y = z = null
```
Heap Reference Analysis: Our Solution

While loop is not executed even once

```
y = z = null
1 w = x
   w = null
2 while (x.data < max)
   { x.lptr = null
3   x = x.rptr   }
x.rptr = x.lptr.rptr = null
x.lptr.lptr.lptr = null
x.lptr.lptr.rptr = null
4 y = x.lptr
   x.lptr = y.rptr = null
   y.lptr.lptr = y.lptr.rptr = null
5 z = New class of z
   z.lptr = z.rptr = null
6 y = y.lptr
   y.lptr = y.rptr = null
7 z.sum = x.data + y.data
   x = y = z = null
```

Uday Khedker
Heap Reference Analysis: Our Solution

y = z = null
1 w = x
   w = null
2 while (x.data < max)
   { x.lptr = null
     x = x.rptr
   }
   x.rptr = x.lptr.rptr = null
   x.lptr.lptr.lptr = null
   x.lptr.lptr.rptr = null
3 w = x
4 y = x.lptr
   x.lptr = y.rptr = null
   y.lptr.lptr = y.lptr.rptr = null
5 z = New class
   z.lptr = z.rptr = null
6 y = y.lptr
   y.lptr = y.rptr = null
7 z.sum = x.data + y.data
   x = y = z = null

While loop is not executed even once
y = z = null
1 w = x
   w = null
2 while (x.data < max)
   { x.lptr = null
   x = x.rptr
   x.rptr = x.lptr.rptr = null
   x.lptr.lptr.lptr = null
   x.lptr.lptr.rptr = null
3
4 y = x.lptr
   x.lptr = y.rptr = null
5 z = New class_of_z
   z.lptr = z.rptr = null
6 y = y.lptr
   y.lptr = y.rptr = null
7 z.sum = x.data + y.data
   x = y = z = null

While loop is not executed even once

Heap Reference Analysis: Our Solution

Stack

Heap
Heap Reference Analysis: Our Solution

```
y = z = null
1 w = x
   w = null
2 while (x.data < max)
   {  x.lptr = null
      x = x.rptr       }
x.rptr = x.lptr.rptr = null
x.lptr.lptr.lptr = null
x.lptr.lptr.rptr = null
3 y = x.lptr
   y.lptr = y.rptr = null
4 z = New class
   z.lptr = z.rptr = null
5 y = y.lptr
   y.lptr = y.rptr = null
6 z.sum = x.data + y.data
7 x = y = z = null
```

While loop is not executed even once
Heap Reference Analysis: Our Solution

\[
y = z = \text{null}
\]

1. \[w = x\]
   \[w = \text{null}\]

2. \[\text{while } (x.\text{data} < \text{max})\]
   \[
   \{\quad x.\text{lptr} = \text{null}\]
   \[x = x.\text{rptr}\]
   \[x.\text{rptr} = x.\text{lptr}.\text{rptr} = \text{null}\]
   \[x.\text{lptr}.\text{lptr}.\text{lptr} = \text{null}\]
   \[x.\text{lptr}.\text{lptr}.\text{rptr} = \text{null}\]

3. \[x = x.\text{rptr}\]

4. \[y = x.\text{lptr}\]
   \[x.\text{lptr} = y.\text{rptr} = \text{null}\]
   \[y.\text{lptr}.\text{lptr} = y.\text{rptr}.\text{rptr} = \text{null}\]

5. \[z = \text{New class of z}\]
   \[z.\text{lptr} = z.\text{rptr} = \text{null}\]

6. \[y = y.\text{lptr}\]
   \[y.\text{lptr} = y.\text{rptr} = \text{null}\]

7. \[z.\text{sum} = x.\text{data} + y.\text{data}\]
   \[x = y = z = \text{null}\]

While loop is executed once
Heap Reference Analysis: Our Solution

Stack

Heap

While loop is executed twice

1. \( y = z = \text{null} \)
2. \( w = x \)
   \( w = \text{null} \)
3. \( \text{while (x.data < max)} \) 
   \( \{ \) 
   \( \quad x.lptr = \text{null} \) 
   \( \quad x = x.rptr \) 
   \( \} \)
   \( x.rptr = x.lptr.rptr = \text{null} \)
   \( x.lptr.lptr.lptr = \text{null} \)
   \( x.lptr.lptr.rptr = \text{null} \)
4. \( y = x.lptr \)
   \( x.lptr = y.rptr = \text{null} \)
   \( y.lptr.lptr = y.lptr.rptr = \text{null} \)
5. \( z = \text{New class of z} \)
   \( z.lptr = z.rptr = \text{null} \)
6. \( y = y.lptr \)
   \( y.lptr = y.rptr = \text{null} \)
7. \( z.sum = x.data + y.data \)
   \( x = y = z = \text{null} \)
Heap Reference Analysis [TOPLAS 2007]

- **Problem.** A lot of unused data remains unclaimed even in the best of garbage collectors. In C/C++, memory leaks is a major problem.

- **Our Objectives.** Static analysis of heap allocated data to improve garbage collection and plug memory leaks.

- **Main Challenge.** Unlike stack and static data,
  - heap data accessible to any procedure is unbounded. Hence,
  - the mapping between object names and their addresses needs to change at runtime.

- **Our Key Idea.** Build bounded abstractions of heap data in terms of graphs and perform analysis using these graphs as data flow values.

- **Current status.**

- **Further Work.**
Heap Reference Analysis [TOPLAS 2007]

- **Problem.** A lot of unused data remains unclaimed even in the best of garbage collectors. In C/C++, memory leaks is a major problem.

- **Our Objectives.** Static analysis of heap allocated data to improve garbage collection and plug memory leaks.

- **Main Challenge.** Unlike stack and static data,
  - heap data accessible to any procedure is unbounded. Hence,
  - the mapping between object names and their addresses needs to change at runtime.

- **Our Key Idea.** Build bounded abstractions of heap data in terms of graphs and perform analysis using these graphs as data flow values.

- **Current status.** Theory and prototype implementation (at the intraprocedural level) ready for Java.

- **Further Work.**
 Heap Reference Analysis [TOPLAS 2007]

• **Problem.** A lot of unused data remains unclaimed even in the best of garbage collectors. In C/C++, memory leaks is a major problem.

• **Our Objectives.** Static analysis of heap allocated data to improve garbage collection and plug memory leaks.

• **Main Challenge.** Unlike stack and static data,
  ▶ heap data accessible to any procedure is unbounded. Hence,  
  ▶ the mapping between object names and their addresses needs to change at runtime.

• **Our Key Idea.** Build bounded abstractions of heap data in terms of graphs and perform analysis using these graphs as data flow values.

• **Current status.** Theory and prototype implementation (at the intraprocedural level) ready for Java.

• **Further Work.** Liveness based interprocedural alias analysis.
Precise Construction of Call Graphs (or Constructing Callee Contexts)

- **Problem.** Presence of function pointers obscures the caller-callee relationship between procedures.
  - Significant imprecision in the result of any analysis
  - Efficiency and scalability is adversely affected

- **Main Challenges.**

- **Research Goals.**

- **Additional Benefits.**
What Does A Callee Context Mean?

\[ a = ["P", "Q"] \]

\[ \text{Start}_{\text{main}} \]

\[ \text{Start}_{P} \]

\[ \text{x} \ast y \]

\[ \text{End}_{P} \]

\[ \text{C}_{1} \]

\[ \text{call} \ast fp \]

\[ \text{End}_{Q} \]

\[ \text{Start}_{Q} \]

\[ \text{x} \ast y \]

\[ \text{End}_{main} \]

\[ n_{1} \]

\[ fp = a[i] \]

\[ n_{2} \]

\[ i = i + 1 \]

\[ n_{3} \]

\[ \text{if } i < n \]
What Does A Callee Context Mean?

```
Start_{main}  

\[ a = ['P'', 'Q'''] \]

Start_{P}  \[ x \ast y \]

\[ n_1 \]
\[ fp = a[i] \]

C_1 \[ call \ast fp \]

\[ x \ast y \] Start_{Q}

\[ \text{Is } x \ast y \text{ available?} \]

End_{P}

\[ \rightleftharpoons \]

End_{main}

\[ \rightleftharpoons \]

End_{Q}

\[ n_2 \]
\[ i = i + 1 \]

\[ n_3 \]
\[ \text{if } i < n \]

\[ \rightleftharpoons \]

End_{main}
```

Uday Khedker

IIT Bombay
What Does A Callee Context Mean?

\[ a = ["P"', "Q"] \]

---

**Diagram:**

- **Start**
  - \( Start_{main} \)
  - \( Start_P \)
  - \( Start_Q \)

- **Variables**
  - \( i \)
  - \( n \)

- **Operations**
  - \( fp = a[i] \)
  - \( call \ * fp \)
  - \( i = i + 1 \)
  - \( if \ i < n \)

- **Flow**
  - \( x*y \)
  - \( Is \ x*y \ available? \)
  - \( Invalid \ execution \ path! \)

---

Uday Khedker
What Does A Callee Context Mean?

\[ a = ["P", "Q"] \]

**Start\(_{main}\)**

- \( n_1 \)
  - \( fp = a[i] \)

**Start\(_{P}\)**

- \( \times \ast y \)

**C\(_1\)**

- \( call \ast fp \)

**End\(_{P}\)**

**R\(_1\)**

- \( i = i + 1 \)

**End\(_{main}\)**

**Valid execution path!**

**Start\(_{Q}\)**

- \( \times \ast y \)

**End\(_{Q}\)**

**Is \( \times \ast y \) available?**

- \( if \ i < n \)
What Does A Callee Context Mean?

\[ a = \left[ \text{"P"}, \text{"Q"} \right] \]

\[ \text{Start}_{\text{main}} \]

\[ \text{Call}_{p} \]
\[ \text{Start}_{P} \]
\[ \text{x} \ast \text{y} \]

\[ \text{End}_{P} \]

\[ \text{C}_{1} \]
\[ \text{call} \ast \text{fp} \]

\[ \text{Call}_{p} \]
\[ \text{Start}_{Q} \]
\[ \text{x} \ast \text{y} \]

\[ \text{End}_{Q} \]

\[ \text{Valid execution path!} \]

\[ \text{Is x \ast y available?} \]

\[ \text{End}_{\text{main}} \]

\[ \text{Start}_{main} \]
\[ \text{fp} = \text{a}[i] \]

\[ \text{R}_{1} \]
\[ i = i + 1 \]

\[ \text{If i < n} \]

\[ \text{Is x \ast y available?} \]
Precise Construction of Call Graphs (or Constructing Callee Contexts)

- **Problem.** Presence of function pointers obscures the caller-callee relationship between procedures.
  - Significant imprecision in the result of any analysis
  - Efficiency and scalability is adversely affected

- **Main Challenges.**

- **Research Goals.**

- **Additional Benefits.**
Precise Construction of Call Graphs (or Constructing Callee Contexts)

- **Problem.** Presence of function pointers obscures the caller-callee relationship between procedures.
  - Significant imprecision in the result of any analysis
  - Efficiency and scalability is adversely affected

- **Main Challenges.** Precise and efficient interprocedural analysis of
  - pointers, and
  - data structure hierarchy declaration and usage

- **Research Goals.**

- **Additional Benefits.**
Precise Construction of Call Graphs (or Constructing Callee Contexts)

- **Problem.** Presence of function pointers obscures the caller-callee relationship between procedures.
  - Significant imprecision in the result of any analysis
  - Efficiency and scalability is adversely affected

- **Main Challenges.** Precise and efficient interprocedural analysis of
  - pointers, and
  - data structure hierarchy declaration and usage

- **Research Goals.** Order sensitive call disambiguation analysis
  - Flow and context sensitive data structure analysis
  - Creating a mechanism to identify the exact caller to which information should be propagated

- **Additional Benefits.**
Precise Construction of Call Graphs (or Constructing Callee Contexts)

- **Problem.** Presence of function pointers obscures the caller-callee relationship between procedures.
  - Significant imprecision in the result of any analysis
  - Efficiency and scalability is adversely affected

- **Main Challenges.** Precise and efficient interprocedural analysis of
  - pointers, and
  - data structure hierarchy declaration and usage

- **Research Goals.** Order sensitive call disambiguation analysis
  - Flow and context sensitive data structure analysis
  - Creating a mechanism to identify the exact caller to which information should be propagated

- **Additional Benefits.** Precise analysis of programs in object oriented languages
Part 3

Conclusions
Observations

- Data flow propagation in real programs seems to involve a much smaller subset of all possible data flow values. *In large programs that work properly, pointer usage is very disciplined and the core information is very small!*

- Earlier approaches reported inefficiency and non-scalability because they computed far more information than required because they
  - did not separate the usable information from unusable information, and
  - used low level abstractions of memory

Their focus was on
  - approximating information to reduce the size, or
  - storing and accessing the information more efficiently
A Spectrum of Possible Ways of Performing Computation

- exhaustive computation
- computation restricted to usable information
- avoiding redundant computation
- demand driven computation

What should be computed?
- Maximum Computation
- Minimum Computation

When should it be computed?
- Early Computation
- Late Computation
A Spectrum of Possible Ways of Performing Computation

exhaustive computation

computation restricted to usable information

avoiding redundant computation

demand driven computation

What should be computed?

Maximum Computation

Minimum Computation

When should it be computed?

Early Computation

Late Computation

Do not compute what you don’t need!

Who defines what is needed?
A Spectrum of Possible Ways of Performing Computation

- **exhaustive computation**
- computation restricted to usable information
- avoiding redundant computation
- demand driven computation

What should be computed?

Maximum Computation

Minimum Computation

When should it be computed?

Early Computation

Late Computation

*Do not compute what you don’t need!*

*Who defines what is needed?* Client

Uday Khedker

IIT Bombay
A Spectrum of Possible Ways of Performing Computation

- exhaustive computation
- computation restricted to usable information
- avoiding redundant computation
- demand driven computation

What should be computed?

Maximum Computation

Early Computation

When should it be computed?

Minimum Computation

Late Computation

Do not compute what you don’t need!

Who defines what is needed?

Algorithm, Data Structure
A Spectrum of Possible Ways of Performing Computation

- **exhaustive computation**
- **computation restricted to usable information**
- **avoiding redundant computation**
- **demand driven computation**
- **splitting into pre and post computation**
- **incremental computation**

**Maximum Computation**

**Early Computation**

**What should be computed?**

**When should it be computed?**

- **Do not compute what you don’t need!**
- **Who defines what is needed?**

**Other examples:**
- Bottom up summary flow functions
- Value contexts
- Work list based methods
- BDDs

**Algorithm, Data Structure**
A Spectrum of Possible Ways of Performing Computation

- exhaustive computation
- computation restricted to usable information
- avoiding redundant computation
- demand driven computation
- incremental computation
- splitting into pre and post computation

What should be computed?

Maximum Computation

Early Computation

When should it be computed?

Minimum Computation

Late Computation

Do not compute what you don’t need!

Who defines what is needed?

Definition of Analysis

Uday Khedker
A Spectrum of Possible Ways of Performing Computation

- exhaustive computation
- computation restricted to usable information
- avoiding redundant computation
- demand driven computation

What should be computed?

Maximum Computation

When should it be computed?

Early Computation

Minumum Computation

Late Computation

Do not compute what you don’t need!

Who defines what is needed? No One!
A Spectrum of Possible Ways of Performing Computation

- exhaustive computation
- computation restricted to usable information
- avoiding redundant computation
- demand driven computation

What should be computed?

When should it be computed?

Maximum Computation

Minimum Computation

Early Computation

Late Computation

Do not compute what you don’t need!

Who defines what is needed?

These seem orthogonal and may be used together

Uday Khedker
Conclusions

- Building quick approximations and compromising on precision may not be necessary for efficiency.
- Building clean abstractions to separate the necessary information from redundant information is much more significant.
Conclusions

• Building quick approximations and compromising on precision may not be necessary for efficiency

• Building clean abstractions to separate the necessary information from redundant information is much more significant

Our experience of points-to analysis shows that

▶ Use of liveness reduced the pointer information . . .
▶ which reduced the number of contexts required . . .
▶ which reduced the liveness and pointer information . . .
Conclusions

- Building quick approximations and compromising on precision may not be necessary for efficiency
- Building clean abstractions to separate the necessary information from redundant information is much more significant

Our experience of points-to analysis shows that
  - Use of liveness reduced the pointer information . . .
  - which reduced the number of contexts required . . .
  - which reduced the liveness and pointer information . . .

This encouraged us to explore bottom summary flow functions for points-to analysis
  - which reduced the number of times a procedure is processed and . . .
  - gave rise to generalized points-to facts . . .
  - which reduced the size of intermediate points-to graphs . . .

Uday Khedker
Conclusions

- Building quick approximations and compromising on precision may not be necessary for efficiency.
- Building clean abstractions to separate the necessary information from redundant information is much more significant.

Our experience of points-to analysis shows that:

- Use of liveness reduced the pointer information.
- Which reduced the number of contexts required.
- Which reduced the liveness and pointer information.

This encouraged us to explore bottom summary flow functions for points-to analysis:
- Which reduced the number of times a procedure is processed and...
- Gave rise to generalized points-to facts...
- Which reduced the size of intermediate points-to graphs...

Approximations should come after building abstractions and not before.
Acknowledgements

Alan Mycroft,
Alefiya Lightwala
Amey Karkare,
Amitabha Sanyal,
Avantika Gupta
Bageshri Sathe,
Prachee Yogi,
Prashant Singh Rawat,

Pritam Gharat,
Priyanka Sawant,
Rohan Padhye,
Shubhangi Agrawal,
Sudakshina Das,
Swati Rathi,
Vini Kanvar,
Vinit Deodhar

...and many more
Last But Not the Least

Thank You!