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

June 2019
Outline

• **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
• 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

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

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

  o 2017 . . 😞
The Mathematics of Pointer Analysis

In the most general situation

- Alias analysis is undecidable.

- 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

![Diagram showing the cycle: Approximation → Imprecision → Inefficiency → may seem to warrant → Approximation]
Why Avoid Approximations?

• Approximations may create a vicious cycle

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

• Two examples of inefficiency cause by approximations
  
  o \(k\)-limited call strings may create “butterfly cycles” causing spurious fixed point computations \cite{Hakjoo, 2010}
  
  o 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>
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

End

f_0
f_1
f_2
f_3
f_i
f_m
Flow Insensitivity in Data Flow Analysis

![Diagram of flow insensitivity in data flow analysis.](attachment:image.png)
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

  o $k$-limiting violates [A]
  o Treating recursion as an SCC violates [A]
  o Functional approaches violate [B]
  o Object sensitivity violates [C]

• Object sensitivity can be completely modelled by calling context sensitivity

  o by a flow sensitive propagation of values representing objects, and
  o identifying a procedure by an (object, procedure) pair, and
  o identifying a context by a call site and the pairs defined as above
Context Sensitivity in Interprocedural Analysis

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

\[ S_s \xrightarrow{c_i} C_i \xrightarrow{x} S_r \]

\[ R_i \xrightarrow{x'} E_s \]

\[ f_r \xrightarrow{y} S_t \]

\[ C_j \xrightarrow{y'} R_j \xrightarrow{c_j} E_t \]
Context Sensitivity in Interprocedural Analysis

\[ S_s \rightarrow C_i \rightarrow R_i \rightarrow E_s \]
\[ S_r \]
\[ E_r \]
\[ S_t \rightarrow C_j \rightarrow R_j \rightarrow E_t \}

\[ x \]
\[ y \]
\[ f_r \]
\[ c_i \]
\[ c_j \]
Context Sensitivity in Interprocedural Analysis
Context Sensitivity in Interprocedural Analysis


Context Sensitivity in the Presence of Recursion

\[ \text{Start}_s \]

\[ \text{call} \]

\[ C_i \]

\[ \text{stop calling} \]

\[ R_i \]

\[ \text{return} \]

\[ \text{End}_s \]
Context Sensitivity in the Presence of Recursion

- Paths from $Start_s$ to $End_s$ should constitute a context free language
  
  $$ call^n \cdot stop \cdot return^n $$

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

  $$ call^* \cdot stop \cdot return^* $$
Context Insensitivity = Imprecision + Potential Inefficiency

\[ \text{Start}_{\text{main}} \]
\[ a = 1 \]
\[ \text{Call } P \]
\[ \text{Start}_p \]
\[ a = a+1 \]
\[ \text{End}_p \]
\[ \text{Call } P \]
\[ \text{End}_{\text{main}} \]
Context Insensitivity = Imprecision + Potential Inefficiency

- What is the value range of $a$?

```plaintext
Start_{main}  
\[ a = 1 \]  
\[ \text{Call } P \]  
\[ \text{End}_{main} \]

\[ \text{Start}_p \]  
\[ a = a + 1 \]  
\[ \text{End}_p \]  
\[ \text{Call } P \]  
```
Context Insensitivity = Imprecision + Potential Inefficiency

- What is the value range of $a$?
**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 $P$

Diagram:

- **Start$_{main}$**
  - $a = 1$
  - Call $P$
  - $(2, 2)$
  - Call $P$
  - **End$_{main}$**

- **Start$_{p}$**
  - $a = a + 1$
  - $(1, 1)$
  - $(2, 2)$
  - **End$_{p}$**

Uday Khedker

IIT Bombay
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

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

\[ a = 1 \]

\text{Call } P

\[ (2, 2) \]

\[ a = a + 1 \]

\text{Call } P

\[ (2, 3) \]

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

- 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
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
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 $End_{main}$ 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

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

![Diagram of the program flow]

- **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, ...)

![Diagram](attachment:image.png)

- $Start_{main}$
- $a = 1$
- Call $P$
- $Start_p$
- $a = a + 1$
- $End_p$
- Call $P$
- $End_{main}$
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

Ci

Ri

End

call

stop calling

return

Uday Khedker
IIT Bombay
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
    
    \[\text{call}^* \cdot \text{stop} \cdot \text{return}^*\]
Pointer Analysis: An Engineer’s Landscape

Flow Sensitivity Increases

\( \text{FI}_{\subseteq} \)

\( \text{FS}_{\text{NoKill}} \)

\( \text{FI}_{\text{SSA}} \)

\( \text{FI} \subseteq \text{FI}_{\text{SSA}} \subseteq \text{FI} \subseteq \text{FI}_{\text{SSA}} \subseteq \text{FS}_{\text{NoKill}} \subseteq \text{FS} \)

Context Sensitivity Increases
Pointer Analysis: An Engineer’s Landscape

Data Structures: BDDs, probabilistic

Flow Sensitivity Increases

Context Sensitivity Increases

FS

FS_{NoKill}

FI_{SSA}

FI_{c}

FI_{e}

CI

CS_{ObjSens}

CS_{Reclns}

CS
Pointer Analysis: An Engineer’s Landscape

Data Structures: BDDs, probabilistic

Methods: parallel, on demand, randomized

Flow Sensitivity Increases

Context Sensitivity Increases

FI_\subseteq \subseteq FI_{SSA} \subseteq FI_{NoKill} \subseteq FS

FI_\subseteq \subseteq CS_{ObjSens} \subseteq CS_{Reclns} \subseteq CS
Pointer Analysis: An Engineer’s Landscape

Data Structures: BDDs, probabilistic

- Refinement: Level-wise, bootstrapping
- Methods: parallel, on demand, randomized

FS
FS_{NoKill}
FI_{SSA}
FI_{≤}
FI_{=} = SSA
CI
CS_{ObjSens}
CS_{Reclns}
CS

Flow Sensitivity Increases
Context Sensitivity Increases
Pointer Analysis: An Engineer’s Landscape

Flow Sensitivity Increases

Crowded Area

Context Sensitivity Increases
Pointer Analysis: An Engineer’s Landscape

\[ \text{Flow Sensitivity Increases} \]

\[ \text{Context Sensitivity Increases} \]

\[ \text{Crowded Area} \]

Uday Khedker
IIT Bombay
Pointer Analysis: An Engineer’s Landscape

- Flow Sensitivity Increases
- Context Sensitivity Increases

Crowded Area

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

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></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 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>Context sensitivity (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>

- **Flow sensitivity**: Restrict the computation only to the usable data. Weave liveness discovery into the analysis.

---

*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>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></td>
<td></td>
</tr>
<tr>
<td>Precise call structure</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Postpone low level connections explicated by the classical points-to facts

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>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.
<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 (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></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>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><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</strong></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><strong>Precise heap abstraction</strong></td>
<td>Liveness access graphs</td>
<td>Mature accomplishment (ISMM17)</td>
</tr>
<tr>
<td></td>
<td>Access based abstraction</td>
<td></td>
</tr>
<tr>
<td><strong>Precise call structure</strong></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</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>Precise call structure</td>
<td>Access based abstraction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Callee sensitivity</td>
<td>Work in progress</td>
</tr>
</tbody>
</table>

Call strings record call *history*. We need to record call *future* also.
<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></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 (Caller sensitivity)</td>
<td>Context sensitivity</td>
<td>Mature accomplishment (SAS12, SOAP13)</td>
</tr>
<tr>
<td></td>
<td>Maturer accomplishment (CC08, SAS16)</td>
<td>Mature accomplishment (CC08, SAS16, SOAP13)</td>
</tr>
<tr>
<td>Precise heap abstraction</td>
<td>Access abstraction</td>
<td>Mature accomplishment (SPLAS07)</td>
</tr>
<tr>
<td></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>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></td>
<td>Access based abstraction</td>
<td>Mature accomplishment (ISMM17)</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>
Liveness Based Pointer Analysis: Motivation

\[
\begin{aligned}
  n_1 & : x = & \& y \\
       & y = & \& z \\
       & z = & \& u \\
\end{aligned}
\]

\[
\begin{aligned}
  n_2 & : *z = y \\
\end{aligned}
\]

\[
\begin{aligned}
  n_3 & : z = y \\
\end{aligned}
\]

\[
\begin{aligned}
  n_4 & : y = & \& x \\
       & use & u \\
       & use & x \\
\end{aligned}
\]

Uday Khedker

IIT Bombay
Liveness Based Pointer Analysis: Motivation

Is all this information really useful?

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

\[ n_1 \]

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

\[ n_4 \]
Liveness Based Pointer Analysis: Motivation

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

\[ n_1 \]

\[ \emptyset \]

\[ x \rightarrow y \rightarrow z \rightarrow u \]

\[ n_2 \]

\[ *z = y \]

\[ x \rightarrow y \rightarrow z \rightarrow u \]

\[ n_3 \]

\[ z = y \]

\[ x \rightarrow y \rightarrow z \]

\[ n_4 \]

\[ y = \& x \]

use \( u \)

use \( x \)

Uday Khedker

IIT Bombay
Liveness Based Pointer Analysis: Motivation

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

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

use \( u \)

use \( x \)
Liveness Based Pointer Analysis: Motivation
Liveness Based Pointer Analysis: Motivation

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

\[ x \rightarrow y \rightarrow z \rightarrow u \]

\[ n_1 \]

\[ \emptyset \]

\[ *z = y \]

\[ n_2 \]

\[ z = y \]

\[ n_3 \]

\[ y = \& x \]

\[ use u \]

\[ use x \]

\[ n_4 \]
Liveness Based Pointer Analysis: Motivation

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

\[ n_1 \]

\[ n_2 \]
\[ *z = y \]

\[ n_3 \]
\[ z = y \]

\[ n_4 \]
\[ y = \& x \]
\[ use \ u \]
\[ use \ x \]
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

Multiple interprocedural paths reaching the procedure

Multiple interprocedural paths reaching the procedure

Start

End

Multiple interprocedural paths reaching the procedure

Multiple interprocedural paths reaching the procedure

Start

End

Data flow values

Contexts

Start

End

$X_0$

Data flow values

Contexts

Start

End

Contexts of the callers

Call sites

Data flow values

Contexts

Start

End

X_0 \rightarrow X_i \rightarrow X_j \rightarrow X_k \rightarrow X_l \rightarrow X_m

c_0 \rightarrow c_1 \rightarrow c_2 \rightarrow c_3 \rightarrow c_4

Uday Khedker

IIT Bombay

Contexts of the callers
- \( X_i \)
- \( X_j \)
- \( X_k \)
- \( X_l \)
- \( X_m \)

Call sites
- \( c_0 \)
- \( c_1 \)
- \( c_2 \)
- \( c_3 \)
- \( c_4 \)

Data flow values
- \( x \)
- \( y \)
- \( y' \)
- \( y' \)
- \( z' \)

Context transition graph
- \( X_i \xrightarrow{c_0} X_0 \)
- \( X_j \xrightarrow{c_1} \)
- \( X_k \xrightarrow{c_2} X_1 \)
- \( X_l \xrightarrow{c_3} \)

Uday Khedker
IIT Bombay

Context transition graph:

- $X_i \xrightarrow{c_0} X_0$
- $X_j \xrightarrow{c_1} X_1$
- $X_k \xrightarrow{c_2} X_1$
- $X_l \xrightarrow{c_3} X_2$
- $X_m \xrightarrow{c_4} X_2$

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

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

All variables are global

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

```
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

f()
{
    *x = y
}

Information from callers

x → φ₁ → φ₂

a → φ₁

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
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
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

\[
f() \\
\{ \\
    *x = y \\
\}
\]
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
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
}
```

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

\[
\begin{align*}
\text{f()} \\
\{ \\
\quad *x &= y \\
\} \\
\end{align*}
\]

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

```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
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
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)

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

f()
{
    *x = y
}

$\phi_1$
$\phi_2$
a
b
x
y

Uday Khedker
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
A GPG is a graph with

- Nodes are generalized points-to blocks (GPBs)
  - A GPB contains a set of GPUs
- Edges represent control flow between GPBs
Generalized Points-to Graphs (GPGs) 1

A GPG is a graph with

- Nodes are generalized points-to blocks (GPBs)
  - A GPB contains a set of GPUs
- Edges represent control flow between GPBs

A GPG is analogous to a CFG of a procedure
Generalized Points-to Graphs (GPGs)

A GPG is a graph with:

- Nodes are generalized points-to blocks (GPBs)
  - A GPB contains a set of GPUs
- Edges represent control flow between GPBs

A GPG is analogous to a CFG of a procedure

First difference:

- GPUs in a GPB represent parallel assignments
- Assignments in a basic block are sequential
A GPG is a graph with:

- **Nodes** are generalized points-to blocks (GPBs)
  - A GPB contains a set of GPUs
- **Edges** represent control flow between GPBs

A GPG is analogous to a CFG of a procedure

### Second difference:

- CFGs contain call basic blocks
- GPGs do not have call GPBs

A GPG is analogous to a CFG of a procedure
Construction of Initial GPGs:

- Non-pointer assignments and condition tests are removed
- Each pointer assignment is transliterated to its GPU
- A separate GPB is created for assignment in the CFG
- GPG edges are induced from the control flow of the CFG
- GPGs contain only variables that are shared across procedures

GPGs then undergo extensive optimizations
GPGs Across Optimizations

CFG of proc f

\[ x = &a; \]
\[ g(); \]
\[ x = &b; \]

CFG of proc g

\[ y = x; \]
GPGs Across Optimizations

CFG of proc f

\[ x = \&a; \]
\[ g(); \]
\[ x = \&b; \]

Initial GPG of proc f

\[ x \xrightarrow{0} a \]
\[ g(); \]
\[ x \xrightarrow{0, 8} b \]

CFG of proc g

\[ y = x; \]
GPGs Across Optimizations

CFG of proc f

\[ x = \&a; \]
\[ g(); \]
\[ x = \&b; \]

Initial GPG of proc f

\[ x \xrightarrow{0} \]
\[ a \]
\[ g(); \]
\[ x \xrightarrow{0} \]
\[ b \]

After Call Inlining

\[ x \xrightarrow{0} \]
\[ a \]
\[ y \xrightarrow{1} \]
\[ x \]
\[ x \xrightarrow{0} \]
\[ b \]

CFG of proc g

\[ y = x; \]
GPGs Across Optimizations

CFG of proc f

\[
\begin{align*}
x &= \&a; \\
g(); \\
x &= \&b;
\end{align*}
\]

Initial GPG of proc f

\[
\begin{align*}
x \\ g(); \\
x \rightarrow b
\end{align*}
\]

After Call Inlining

\[
\begin{align*}
x \\ y \rightarrow x \\
x \rightarrow b \\
x \rightarrow b
\end{align*}
\]

After Strength Reduction

\[
\begin{align*}
x \\
y \rightarrow a \\
x \rightarrow b
\end{align*}
\]

CFG of proc g

\[
y = x;
\]

Uday Khedker
IIT Bombay
GPGs Across Optimizations

CFG of proc f
\[ x = &a; \]
\[ g(); \]
\[ x = &b; \]

Initial GPG of proc f
\[ x \xrightarrow{1,0} a \]
\[ g(); \]
\[ x \xrightarrow{1,0} b \]

After Call Inlining
\[ x \xrightarrow{1,0} a \]
\[ y \xrightarrow{1,1} x \]
\[ x \xrightarrow{1,0} b \]

After Strength Reduction
\[ x \xrightarrow{1,0} a \]
\[ y \xrightarrow{1,0} a \]
\[ x \xrightarrow{1,0} b \]

CFG of proc g
\[ y = x; \]

After Dead GPU Elimination
\[ y \xrightarrow{1,0} a \]
\[ x \xrightarrow{1,0} b \]
GPGs Across Optimizations

CFG of proc f

\[ x = \&a; \]

\[ g(); \]

\[ x = \&b; \]

Initial GPG of proc f

\[ x \rightarrow { 1|0 \atop 1} a \]

\[ g(); \]

\[ x \rightarrow { 1|0 \atop 8} b \]

After Call Inlining

\[ x \rightarrow { 1|0 \atop 1} a \]

\[ y \rightarrow { 1|1 \atop 2} x \]

\[ x \rightarrow { 1|0 \atop 8} b \]

After Strength Reduction

\[ x \rightarrow { 1|0 \atop 1} a \]

\[ y \rightarrow { 1|0 \atop 2} a \]

\[ x \rightarrow { 1|0 \atop 8} b \]

After Dead GPU Elimination

\[ y \rightarrow { 1|0 \atop 2} a \]

\[ x \rightarrow { 1|0 \atop 8} b \]

After Empty GPB Elimination

\[ y \rightarrow { 1|0 \atop 2} a \]

\[ x \rightarrow { 1|0 \atop 8} b \]

Uday Khedker

IIT Bombay
GPGs Across Optimizations

CFG of proc f

\[ x = \&a; \]
\[ g(); \]
\[ x = \&b; \]

Initial GPG of proc f

\[
\begin{align*}
\text{CFG of proc f} & \quad x \quad y \\
& \quad a \quad 1|0 \to 1 \\
& \quad g(); \\
& \quad x \quad b \quad 1|0 \to 8 \\
\end{align*}
\]

After Call Inlining

\[
\begin{align*}
\text{After Call Inlining} & \quad x \quad a \\
& \quad y \quad 1|1 \to 2 \\
& \quad x \quad b \quad 1|0 \to 8 \\
\end{align*}
\]

After Strength Reduction

\[
\begin{align*}
\text{After Strength Reduction} & \quad x \quad a \\
& \quad y \quad 1|0 \to 2 \\
& \quad x \quad b \quad 1|0 \to 8 \\
\end{align*}
\]

CFG of proc g

\[ y = x; \]

After Dead GPU Elimination

\[
\begin{align*}
\text{After Dead GPU Elimination} & \quad y \quad a \\
& \quad x \quad b \\
\end{align*}
\]

After Empty GPB Elimination

\[
\begin{align*}
\text{After Empty GPB Elimination} & \quad y \quad a \\
& \quad x \quad b \\
\end{align*}
\]

After Coalescing

\[
\begin{align*}
\text{After Coalescing} & \quad y \quad a \\
& \quad x \quad b \\
\end{align*}
\]
GPGs Across Optimizations

- All GPGs represent sound and precise summary of procedure f for points-to analysis
- Structurally, all GPGs are different but their application computes identical results
- A series of optimizations increases the compactness of GPGs significantly
Factors affecting Scalability

Three issues that cause non-scalability

• Modelling indirect accesses of pointees that are defined in callers without examining their code
  ▶ GPUs track indirection levels that relate (transitively indirect) pointees of a variable with those of other variables

• Preserving data dependence between memory updates
  ▶ Maintain minimal control flow between memory updates ensuring soundness and precision

• Incorporating the effect of summaries of the callee procedures transitively
  ▶ Series of GPG optimizations gives compactness that mitigate the impact of transitive inlining
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
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

Minimum Computation

Early Computation

Late Computation

When should it be computed?

Do not compute what you don’t need!

Who defines what is needed?

Algorithm, Data Structure

Uday Khedker
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?
- Late Computation

Do not compute what you don't need!

Who defines what is needed?
- Algorithm, Data Structure

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

**Definition of Analysis**

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
- Incremental computation splitting into pre and post 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? No One!

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

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?

These seem orthogonal and may be used together

Uday Khedker

IIT Bombay
The Road Ahead

- And yet, this is not sufficient to scale points-to analysis
- We found GPGs with 742 nodes, 377 calls, 59747 edges containing ONLY 2 GPUs!!
- Our explorations in both top-down and bottom-up approaches of interprocedural analysis lead us to observe that
The Road Ahead

- And yet, this is not sufficient to scale points-to analysis
- We found GPGs with 742 nodes, 377 calls, 59747 edges containing ONLY 2 GPUs!!
- Our explorations in both top-down and bottom-up approaches of interprocedural analysis lead us to observe that

  The real killer of scalability in program analysis is not
  - the **data that needs to be computed** but
  - the **control flow that it is subjected to** in search of precision
The Road Ahead

- And yet, this is not sufficient to scale points-to analysis
- We found GPGs with 742 nodes, 377 calls, 59747 edges containing ONLY 2 GPUs!!
- Our explorations in both top-down and bottom-up approaches of interprocedural analysis lead us to observe that

  The real killer of scalability in program analysis is not
  - the **data that needs to be computed** but
  - the **control flow that it is subjected to** in search of precision

- For scaling program analysis, we need to optimize away the part of the control flow that does not contribute to data flow
The Next Holy Grail in Search of Scalability?

Under-approximated Control Flow

Original Control Flow

Over-approximated Control Flow
The Next Holy Grail in Search of Scalability?

- Under-approximated Control Flow
  - Missing control flow paths

- Original Control Flow

- Over-approximated Control Flow
  - Flow insensitivity
  - Context insensitivity
The Next Holy Grail in Search of Scalability?

- Under-approximated Control Flow: Missing control flow paths
- Original Control Flow: Sound and Precise
- Over-approximated Control Flow: Flow insensitivity, Context insensitivity

Unsound
The Next Holy Grail in Search of Scalability?

Under-approximated Control Flow

Original Control Flow

Over-approximated Control Flow

Precision

Low

High

Low

High

Scalability

Uday Khedker

IIT Bombay
The Next Holy Grail in Search of Scalability?

- Under-approximated Control Flow
- Original Control Flow
- Over-approximated Control Flow

Precision vs. Scalability:
- High Precision, Low Scalability
- Low Precision, High Scalability
The Next Holy Grail in Search of Scalability?

How to eliminate (i.e. over-approximate) redundant control flow for scalability without compromising on soundness and precision?
The Next Holy Grail in Search of Scalability?

Under-approximated Control Flow  |  Original Control Flow  |  Over-approximated Control Flow

Precision:
- Low
- High

Scalability:
- Low
- High

How to eliminate (i.e. over-approximate) redundant control flow for scalability without compromising on soundness and precision?

Uday Khedker
IIT Bombay
The Next Holy Grail in Search of Scalability?

How to eliminate (i.e. over-approximate) redundant control flow for scalability without compromising on soundness and precision? 
The Next Holy Grail in Search of Scalability?

How to eliminate (i.e. over-approximate) redundant control flow for scalability without compromising on soundness and precision?
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 . . .
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,
Alefya 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!