Precise and Scalable Program Analysis

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

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

Feb 2022
Outline

- Some meanderings in precise and scalable analysis
- Intraprocedural analysis
- Interprocedural analysis
- Conclusions
Acknowledgements

The following people have contributed to the work presented in these slides

Alan Mycroft, Amey Karkare, Amitabha Sanyal, Anshuman Dhuliya, Bageshri Sathe, Komal Pathade, Mehul Jain,

Prashant Singh Rawat, Pritam Gharat, Rasesh Tongia, Rohan Padhye, Supratik Chakraborty Swati Jaiswal, Vini Kanvar,

...and many more have contributed indirectly
Some Meanderings
• 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

  ◦ **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

  ◦ 2022 . .

  • Pointer Analysis Musings

  • Some Meanderings

  • Intraprocedural Analysis

  • Interprocedural Analysis

  • Conclusions

  • References
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

Capture the clouds in this picture for study
Need to meet some resource constraints
Cannot represent the entire picture accurately
Abstraction Vs Approximation

Capture the clouds in this picture for study
Need to meet some resource constraints
Cannot represent the entire picture accurately

Use approximation and meet resource constraints
Usually easy and scalable, but imprecise
Some times, too imprecise to be of any use
Capture the clouds in this picture for study
Need to meet some resource constraints
Cannot represent the entire picture accurately

Use approximation and meet resource constraints
Usually easy and scalable, but imprecise
Some times, too imprecise to be of any use

Use abstraction and meet resource constraints
Usually difficult, need to dig deeper to define exactly what is needed and what can be thrown away
However, it can be precise and scalable
Abstraction Vs. Approximation in Static Analysis

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

• Approximations
  o focus on efficiency and scalability
  o 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

• Our Holy Grail
  
  *Build clean abstractions before surrendering to the approximations*
Program Analysis: Precision versus Scalability

• Ideally, an analysis should be
  ○ Sound
  ○ Precise
  ○ Scalable
Program Analysis: Precision versus Scalability

- Ideally, an analysis should be
  - Sound
  - Precise
  - Scalable

**Common belief**
- Precision and scalability cannot be achieved together for exhaustive analysis

**Common Practice**
- Trade off precision using approximations
Program Analysis: Precision versus Scalability

• Ideally, an analysis should be
  o Sound
  o Precise
  o Scalable

• The main factors enhancing the precision of an exhaustive (as against a demand-driven) analysis are
  o Flow sensitivity
  o Context sensitivity
  o Field sensitivity
  o Precise heap abstraction
  o Precise call graph
Program Analysis: Precision versus Scalability

• Ideally, an analysis should be
  o Sound
  o Precise
  o Scalable

• The main factors enhancing the precision of an exhaustive (as against a demand-driven) analysis are
  o Flow sensitivity
  o Context sensitivity
  o Field sensitivity
  o Precise heap abstraction
  o Precise call graph

Compromises on these lead to approximations
Demand-Driven Analysis Vs. Exhaustive Analysis

- **Exhaustive.** Compute all possible information
- **Demand-Driven.** Compute only the requested information (by a client)

Different from incremental analysis which also computes only some information but it updates the earlier computed solution
## The Classical Precision-Efficiency Dilemma

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>Role in precision</th>
<th>Cause of non-scalability</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>Field 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 non-scalability</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>Field sensitivity</td>
<td>Different fields of an object</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 non-scalability</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>Field sensitivity</td>
<td>Different fields of an object</td>
<td>Pointees along different fields</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>
## The First Order Effects of Approximations

<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>Field insensitivity</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>

- Approximation admits:
  - Flow insensitivity
  - Context insensitivity (or partial context sensitivity)
  - Field insensitivity
  - Imprecision in call graphs
  - Allocation site based heap abstraction
The First Order Effects of Approximations

<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>Field insensitivity</td>
<td>Spurious paths in the memory graph</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 the memory graph</td>
</tr>
</tbody>
</table>
• Approximations may create a vicious cycle

Approximation \(\rightarrow\) Imprecision \(\rightarrow\) Non-Scalability

causes

may seem to warrant

may cause
The Second Order Effect of Approximations

• Approximations may create a vicious cycle

Approximation \( \rightarrow \) Imprecision \( \rightarrow \) Non-Scalability

• Two examples of non-scalability 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
Flow Sensitivity Vs. Flow Insensitivity

Flow Sensitive

Flow Insensitive

Start

End

References

Uday Khedker
IIT Bombay

Talk Title: PSPA Research

Topic: Some Meanderings

Intraprocedural Analysis

Interprocedural Analysis

Conclusions

15/99
Flow Sensitivity Vs. Flow Insensitivity

Assumption: Statements can be executed in any order
Flow Sensitivity Vs. Flow Insensitivity

Flow Sensitive

Flow Insensitive

Start

End

0 \rightarrow S_0 \rightarrow 1 \rightarrow S_1 \rightarrow 2 \rightarrow S_2 \rightarrow 3 \rightarrow S_3 \rightarrow \cdots \rightarrow i \rightarrow S_i \rightarrow \cdots \rightarrow m \rightarrow S_m
Flow Sensitivity Vs. Flow Insensitivity

Flow-insensitive analysis is less precise than a flow-sensitive analysis

Flow-insensitive points-to information
\[ \{ x \rightarrow a, x \rightarrow b, y \rightarrow a, y \rightarrow b \} \]

Flow-sensitive points-to information
\[ \{ x \rightarrow b, y \rightarrow a \} \]
Context Sensitivity Vs. Context Insensitivity

\[ \text{Start}_s \]

\[ a = \& b \]

\[ c_i \]

\[ C_i \]

\[ R_i \]

\[ \text{End}_s \]

\[ \text{Start}_r \]

\[ \text{Start}_t \]

\[ a = \& c \]

\[ c_j \]

\[ C_j \]

\[ R_j \]

\[ \text{End}_t \]
Context Sensitivity Vs. Context Insensitivity

Start₏

a = &b

cᵢ

Cᵢ

Rᵢ

Endᵦ

Startᵦ

a = &c

cⱼ

Cⱼ

Rⱼ

Endⱼ
Context Sensitivity Vs. Context Insensitivity

\[
\begin{align*}
\text{Start}_s & \quad a = & b \\
C_i & \quad a \rightarrow b \\
R_i & \quad c_i \\
\text{End}_s & \\
\text{Start}_r & \quad a = & c \\
\text{End}_r & \\
\text{Start}_t & \\
\end{align*}
\]
Context Sensitivity Vs. Context Insensitivity

\[ \text{Start}_s \]
\[ a = \& b \]
\[ C_i \]
\[ R_i \]
\[ \text{End}_s \]

\[ \text{Start}_r \]
\[ C_j \]
\[ R_j \]
\[ \text{End}_r \]

\[ f_r \]
\[ \text{Start}_t \]
\[ a = \& c \]
\[ C_j \]
\[ R_j \]
\[ \text{End}_t \]
Context Sensitivity Vs. Context Insensitivity

\[ a = \& b \]

\[ c_i, a \rightarrow b \]

\[ C_i \]

\[ R_i \]

\[ \text{End}_s \]

\[ a \rightarrow b \]

\[ f_r \]

\[ c_j \]

\[ R_j \]

\[ \text{End}_t \]
Context Sensitivity Vs. Context Insensitivity

\[ a = \& b \]

\[ c_i \]

\[ a \rightarrow b \]

\[ C_i \]

\[ R_i \]

\[ \text{End}_s \]

\[ f_r \]

\[ a \rightarrow c \]

\[ a \rightarrow b \]

\[ a \rightarrow c \]

\[ C_j \]

\[ R_j \]

\[ \text{End}_t \]

\[ \text{Start}_t \]

\[ a = \& c \]

\[ c_j \]
Context Sensitivity Vs. Context Insensitivity

\[ \text{Start}_s \quad a = \& b \quad \text{Start}_t \quad a = \& c \]

\[ C_i \quad R_i \quad C_j \quad R_j \]

\[ \text{End}_s \quad \times \quad \text{End}_t \]
Context Sensitivity Vs. Context Insensitivity

```
Start_s
a = &b
Ci
Ci
Ri
End_s

Start_r
Start_t
a = &c
Cj
Rj
End_t
```

\[ a \mapsto b \quad a \mapsto c \quad a \mapsto c \]

\[ f_r \]

C_i \quad R_i \quad C_j \quad R_j
Context Sensitivity Vs. Context Insensitivity

Starts

\[ a = \& b \]

\[ c_i \]

\[ R_i \]

\[ End_s \]

Start_r

\[ a \rightarrow b \]

\[ a \rightarrow c \]

\[ f_r \]

\[ C_i \]

\[ R_j \]

\[ End_r \]

\[ C_j \]

\[ a \rightarrow c \]

\[ C_j \]

\[ a \rightarrow c \]

\[ R_j \]

\[ End_t \]
Context Sensitivity Vs. Context Insensitivity

- **Start\textsubscript{s}**
  - $a = \&b$
  - $c_i$
  - $C_i$
  - $R_i$
  - $\text{End}_s$

- **Start\textsubscript{r}**
  - $a = \&c$
  - $c_j$
  - $C_j$
  - $R_j$
  - $\text{End}_r$

- **End\textsubscript{s}**
  - $a \rightarrow b$

- **End\textsubscript{r}**
  - $a \rightarrow b$
  - $a \rightarrow c$

$\text{Context Sensitivity Vs. Context Insensitivity}$
Context Sensitivity Vs. Context Insensitivity

Context-insensitive analysis is less precise than a context-sensitive analysis

Context-insensitive analysis

Context-sensitive analysis

\[ \text{Start}_s \]

\[ a = \& b \]

\[ c_i \]

\[ C_i \]

\[ R_i \]

\[ \text{End}_s \]

\[ \text{Start}_r \]

\[ a \rightarrow b \]

\[ \text{Start}_r \]

\[ a \rightarrow c \]

\[ c_j \]

\[ C_j \]

\[ R_j \]

\[ \text{End}_r \]

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

\[
\text{Start}_s \quad \text{call} \quad \text{Ci} \quad \text{stop calling} \quad \text{R}_i \quad \text{return} \quad \text{End}_s
\]
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 Non-Scalability

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

\[ a = 1 \]

\[ \text{Call } P \]

\[ \text{Call } P \]

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

\[ \text{Start}_p \]

\[ a = a + 1 \]

\[ \text{End}_p \]
Context Insensitivity = Imprecision + Potential Non-Scalability

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

\[ a = 1 \]

\[ \text{Call } P \]

\[ a = a + 1 \]

\[ \text{Call } P \]

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

\[ \text{Start}_p \]

\[ \text{End}_p \]

- What is the value range of \( a \)?
Context Insensitivity = Imprecision + Potential Non-Scalability

\[ \text{Start}_{\text{main}} \]
\[ a = 1 \]
\[ \text{Call } P \]
\[ \text{End}_{\text{main}} \]
\[ \text{Start}_{p} \]
\[ a = a + 1 \]
\[ (1, 1) \]
\[ \text{Call } P \]
\[ (2, 2) \]
\[ \text{End}_{p} \]

- What is the value range of \( a \)?
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 Non-Scalability

- What is the value range of $a$?
- Context sensitive analysis
  - Data flow value propagated back to the current caller of $P$

Uday Khedker
IIT Bombay

Talk Title: PSPA Research

Topic: Some Meanderings

Intraprocedural Analysis
Interprocedural Analysis
Conclusions
References
Context Insensitivity = Imprecision + Potential Non-Scalability

- 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 $(3, 3)$
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 Non-Scalability

- 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 Non-Scalability

- 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

```
Start_{main}

a = 1

Call P

(2, 3)

Call P

(2, 3)

End_{main}

(1, 2) → (1, 1)

Start_P

a = a + 1

(2, 3)

End_P

Start_P

a = a + 1

(2, 3)

End_P
```
Context Insensitivity = Imprecision + Potential Non-Scalability

- 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
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 Non-Scalability

- 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 Non-Scalability

- 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 \((2, \ldots)\)
• 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 $(2, \ldots)$
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**
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
## Field Sensitivity Vs. Field Insensitivity

<table>
<thead>
<tr>
<th>Program</th>
<th>Field-sensitive points-to graph</th>
<th>Field-insensitive points-to graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \rightarrow f = &amp;y$</td>
<td>![Field-sensitive diagram]</td>
<td></td>
</tr>
<tr>
<td>$x \rightarrow g = &amp;z$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$w = x \rightarrow f$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Field-sensitive points-to graph:
- $x$ points to $f$ and $y$
- $x$ points to $g$ and $z$
- $w = x \rightarrow f$

Field-insensitive points-to graph:
- $x$ points to $f$
- $y$ points to $f$
- $z$ points to $g$
- $w$ points to $f$
Field Sensitivity Vs. Field Insensitivity

Program

$x \rightarrow f = \&y$
$x \rightarrow g = \&z$
$w = x \rightarrow f$

Field-sensitive points-to graph

Field-insensitive points-to graph
Field Sensitivity Vs. Field Insensitivity

<table>
<thead>
<tr>
<th>Program</th>
<th>Field-sensitive points-to graph</th>
<th>Field-insensitive points-to graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \rightarrow f = &amp;y$</td>
<td>![Field-sensitive graph]</td>
<td>![Field-insensitive graph]</td>
</tr>
<tr>
<td>$x \rightarrow g = &amp;z$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$w = x \rightarrow f$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Field-insensitive analysis is less precise than a field-sensitive analysis
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 (without the creation sites) can be modelled by call context sensitivity
  - by a flow sensitive propagation of values representing objects, and
  - identifying a procedure by an (object, procedure) pair, and
  - identifying a context by a call site and the pairs defined as above
Pointer analysis is a fertile ground for research because the factors that enhance the precision of points-to analysis (flow, context, and field sensitivity), hamper scalability
Pointer analysis is a fertile ground for research because the factors that enhance the precision of points-to analysis (flow, context, and field sensitivity), hamper scalability.

Data structures: BDDs, probabilistic

- Flow Sensitivity Increases
- Context Sensitivity Increases

Cl, CS<sub>ObjSens</sub>, CS<sub>Reclns</sub>, CS
Pointer analysis is a fertile ground for research because the factors that enhance the precision of points-to analysis (flow, context, and field sensitivity), hamper scalability.
Pointer analysis is a fertile ground for research because the factors that enhance the precision of points-to analysis (flow, context, and field sensitivity), hamper scalability.
Pointer analysis is a fertile ground for research because the factors that enhance the precision of points-to analysis (flow, context, and field sensitivity), hamper scalability.
Pointer analysis is a fertile ground for research because the factors that enhance the precision of points-to analysis (flow, context, and field sensitivity), hamper scalability.
### In Search of Abstractions for Precision Without Non-Scalability

<table>
<thead>
<tr>
<th>Desired Abstraction</th>
<th>Enabling Abstraction</th>
<th>Status of our work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow- and Field-sensitivity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context-sensitivity (actually 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>
<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- and Field-sensitivity</td>
<td>Joint liveness and points-to analysis</td>
<td>Partial accomplishment (SAS12)</td>
</tr>
<tr>
<td>Context-sensitivity (actually 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>

Restrict the computation only to the usable data. Weave liveness discovery into the analysis.
### In Search of Abstractions for Precision Without Non-Scalability

<table>
<thead>
<tr>
<th>Desired Abstraction</th>
<th>Enabling Abstraction</th>
<th>Status of our work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow- and Field-sensitivity</td>
<td>Joint liveness and points-to analysis</td>
<td>Partial accomplishment (SAS12)</td>
</tr>
<tr>
<td>Context-sensitivity (actually caller-sensitivity)</td>
<td>Bypassing irrelevant calls for liveness and points-to analysis</td>
<td>Work in progress</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 propagation of irrelevant information about pointers to callees**
<table>
<thead>
<tr>
<th>Desired Abstraction</th>
<th>Enabling Abstraction</th>
<th>Status of our work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow- and Field-sensitivity</td>
<td>Joint liveness and points-to analysis</td>
<td>Partial accomplishment (SAS12)</td>
</tr>
<tr>
<td></td>
<td>Bypassing irrelevant calls for liveness and points-to analysis</td>
<td>Work in progress</td>
</tr>
<tr>
<td></td>
<td><strong>Synergistic program analyses</strong></td>
<td>Work in progress</td>
</tr>
<tr>
<td>Context-sensitivity (actually 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>
<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><strong>Flow- and Field-sensitivity</strong></td>
<td>Joint liveness and points-to analysis</td>
<td>Partial accomplishment (SAS12)</td>
</tr>
<tr>
<td></td>
<td>Bypassing irrelevant calls for liveness and points-to analysis</td>
<td>Work in progress</td>
</tr>
<tr>
<td></td>
<td>Synergistic program analyses</td>
<td>Work in progress</td>
</tr>
<tr>
<td></td>
<td>Partially path-sensitive analysis</td>
<td>Mature accomplishment (CC18,CC19)</td>
</tr>
<tr>
<td><strong>Context-sensitivity (actually caller-sensitivity)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Precise heap abstraction</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Precise call structure</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Lifting any given analyses to a level where known infeasible paths are excluded.
## In Search of Abstractions for Precision Without Non-Scalability

<table>
<thead>
<tr>
<th>Desired Abstraction</th>
<th>Enabling Abstraction</th>
<th>Status of our work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow- and Field-sensitivity</td>
<td>Joint liveness and points-to analysis</td>
<td>Partial accomplishment (SAS12)</td>
</tr>
<tr>
<td></td>
<td>Bypassing irrelevant calls for liveness and points-to analysis</td>
<td>Work in progress</td>
</tr>
<tr>
<td></td>
<td>Synergistic program analyses</td>
<td>Work in progress</td>
</tr>
<tr>
<td></td>
<td>Partially path-sensitive analysis</td>
<td>Mature accomplishment (CC18,CC19)</td>
</tr>
<tr>
<td>Context-sensitivity (actually caller-sensitivity)</td>
<td>Value contexts</td>
<td>Mature accomplishment (CC08,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.

---

**Uday Khedker**

**IIT Bombay**

**Talk Title:** PSPA Research

**Topic:** Some Meanderings

Intraprocedural Analysis

Interprocedural Analysis

Conclusions

References
### In Search of Abstractions for Precision Without Non-Scalability

<table>
<thead>
<tr>
<th>Desired Abstraction</th>
<th>Enabling Abstraction</th>
<th>Status of our work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow- and Field-sensitivity</td>
<td>Joint liveness and points-to analysis</td>
<td>Partial accomplishment (SAS12)</td>
</tr>
<tr>
<td></td>
<td>Bypassing irrelevant calls for liveness and points-to analysis</td>
<td>Work in progress</td>
</tr>
<tr>
<td></td>
<td>Synergistic program analyses</td>
<td>Work in progress</td>
</tr>
<tr>
<td></td>
<td>Partially path-sensitive analysis</td>
<td>Mature accomplishment (CC18)</td>
</tr>
<tr>
<td>Context-sensitivity (actually caller-sensitivity)</td>
<td>Value contexts</td>
<td>Mature accomplishment (CC08, SOAP13)</td>
</tr>
<tr>
<td></td>
<td>GPG based bottom-up summary flow functions</td>
<td>Mature accomplishment (SAS16, TOPLAS20)</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 Non-Scalability

<table>
<thead>
<tr>
<th>Desired Abstraction</th>
<th>Enabling Abstraction</th>
<th>Status of our work</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Flow- and Field-sensitivity</strong></td>
<td>Joint liveness and points-to analysis</td>
<td>Partial accomplishment (SAS12)</td>
</tr>
<tr>
<td></td>
<td>Bypassing irrelevant calls for liveness and points-to analysis</td>
<td>Work in progress</td>
</tr>
<tr>
<td></td>
<td>Synergistic program analyses</td>
<td>Work in progress</td>
</tr>
<tr>
<td></td>
<td>Partially path-sensitive analysis</td>
<td>Work in progress</td>
</tr>
<tr>
<td><strong>Context-sensitivity (actually caller-sensitivity)</strong></td>
<td>Value contexts</td>
<td>Mature accomplishment (CC08, SOAP13)</td>
</tr>
<tr>
<td></td>
<td>GPG based bottom-up summary flow functions</td>
<td>Mature accomplishment (SAS16, TOPLAS20)</td>
</tr>
<tr>
<td></td>
<td><strong>A Unified model of context-sensitive methods</strong></td>
<td>Mature accomplishment (CSUR21)</td>
</tr>
<tr>
<td><strong>Precise heap abstraction</strong></td>
<td></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 Non-Scalability

<table>
<thead>
<tr>
<th>Desired Abstraction</th>
<th>Enabling Abstraction</th>
<th>Status of our work</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Flow- and Field-sensitivity</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Joint liveness and points-to analysis</td>
<td>Partial accomplishment (SAS12)</td>
<td></td>
</tr>
<tr>
<td>Bypassing irrelevant calls for liveness</td>
<td>Work in progress</td>
<td></td>
</tr>
<tr>
<td>and points-to analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synergistic program analyses</td>
<td>Work in progress</td>
<td></td>
</tr>
<tr>
<td>Partially path-sensitive analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Context-sensitivity</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(actually caller-sensitivity)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value contexts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPG based bottom-up sum flow functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A Unified model of context-sensitive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Interprocedural SSA form</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Precise heap abstraction</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Precise call structure</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Eliminate pointer dereferences and construct context-sensitive SSA
### In Search of Abstractions for Precision Without Non-Scalability

<table>
<thead>
<tr>
<th>Desired Abstraction</th>
<th>Enabling Abstraction</th>
<th>Status of our work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow- and Field-sensitivity</td>
<td>Joint liveness and points-to analysis</td>
<td>Partial accomplishment (SAS12)</td>
</tr>
<tr>
<td></td>
<td>Bypassing irrelevant calls for liveness and points-to analysis</td>
<td>Work in progress</td>
</tr>
<tr>
<td></td>
<td>Synergistic program analyses</td>
<td>Work in progress</td>
</tr>
<tr>
<td></td>
<td>Partially path-sensitive analysis</td>
<td>Mature accomplishment (CC18, CC19)</td>
</tr>
<tr>
<td>Context-sensitivity (actually caller-sensitivity)</td>
<td>Value contexts</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GPG based bottom-up summary flow functions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A Unified model of context-sensitive methods</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Interprocedural SSA form</td>
<td></td>
</tr>
<tr>
<td>Precise heap abstraction</td>
<td>Liveness analysis of heap</td>
<td>Partial accomplishment (TOPLAS07)</td>
</tr>
<tr>
<td>Precise call structure</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Identify the part of heap actually accessed in terms of patterns of accesses*
### In Search of Abstractions for Precision Without Non-Scalability

<table>
<thead>
<tr>
<th>Desired Abstraction</th>
<th>Enabling Abstraction</th>
<th>Status of our work</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Flow- and Field-sensitivity</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Joint liveness and points-to analysis</td>
<td></td>
<td>Partial accomplishment (SAS12)</td>
</tr>
<tr>
<td>Bypassing irrelevant calls for liveness and points-to analysis</td>
<td></td>
<td>Work in progress</td>
</tr>
<tr>
<td>Synergistic program analyses</td>
<td></td>
<td>Work in progress</td>
</tr>
<tr>
<td>Partially path-sensitive analysis</td>
<td></td>
<td>Mature accomplishment (CC18, CC19)</td>
</tr>
<tr>
<td><strong>Context-sensitivity (actually caller-sensitivity)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value contexts</td>
<td></td>
<td>Mature accomplishment (CC08, SOAP13)</td>
</tr>
<tr>
<td>GPG based bottom-up summary flow functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A Unified model of context-sensitive methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interprocedural SSA form</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Precise heap abstraction</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Liveness analysis of heap</td>
<td></td>
<td>Partial accomplishment (TOPLAS07)</td>
</tr>
<tr>
<td>Combined allocation site and access path abstraction</td>
<td></td>
<td>Mature accomplishment (ISMM17)</td>
</tr>
<tr>
<td><strong>Precise call structure</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Distinguish between heap locations based on how they are accessed apart from how they are allocated.
<table>
<thead>
<tr>
<th>Desired Abstraction</th>
<th>Enabling Abstraction</th>
<th>Status of our work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow- and Field-sensitivity</td>
<td>Joint liveness and points-to analysis</td>
<td>Partial accomplishment (SAS12)</td>
</tr>
<tr>
<td></td>
<td>Bypassing irrelevant calls for liveness and points-to analysis</td>
<td>Work in progress</td>
</tr>
<tr>
<td></td>
<td>Synergistic program analyses</td>
<td>Work in progress</td>
</tr>
<tr>
<td></td>
<td>Partially path-sensitive analysis</td>
<td>Mature accomplishment (CC18,CC19)</td>
</tr>
<tr>
<td>Context-sensitivity</td>
<td>Value contexts</td>
<td>Mature accomplishment (CC08,SOAP13)</td>
</tr>
<tr>
<td>(actually caller-sensitivity)</td>
<td>GPG based bottom-up summary flow functions</td>
<td>Mature accomplishment (SAS16, TOPLAS20)</td>
</tr>
<tr>
<td></td>
<td>A Unified model of context-sensitive methods</td>
<td>Mature accomplishment (CSUR21)</td>
</tr>
<tr>
<td></td>
<td>Interprocedural SSA form</td>
<td>Work in progress</td>
</tr>
<tr>
<td>Precise heap abstraction</td>
<td>Liveness analysis of heap</td>
<td>Partial accomplishment (TOPLAS07)</td>
</tr>
<tr>
<td></td>
<td>Combined allocation site and access path abstraction</td>
<td>Mature accomplishment (ISMM17)</td>
</tr>
<tr>
<td>Precise call structure</td>
<td>Callée sensitivity</td>
<td>Work in progress</td>
</tr>
</tbody>
</table>

Call strings record call *history*. We need to record call *future* also.
### In Search of Abstractions for Precision Without Non-Scalability

<table>
<thead>
<tr>
<th>Desired Abstraction</th>
<th>Enabling Abstraction</th>
<th>Status of our work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow- and Field-sensitivity</td>
<td>Joint liveness and points-to analysis</td>
<td>Partial accomplishment (SAS12)</td>
</tr>
<tr>
<td></td>
<td>Bypassing irrelevant calls for liveness and points-to analysis</td>
<td>Work in progress</td>
</tr>
<tr>
<td></td>
<td>Synergistic program analyses</td>
<td>Work in progress</td>
</tr>
<tr>
<td></td>
<td>Partially path-sensitive analysis</td>
<td>Mature accomplishment (CC18, CC19)</td>
</tr>
<tr>
<td>Context-sensitivity (actually caller-sensitivity)</td>
<td>Value contexts</td>
<td>Mature accomplishment (CC08, SOAP13)</td>
</tr>
<tr>
<td></td>
<td>GPG based bottom-up summary flow functions</td>
<td>Mature accomplishment (SAS16, TOPLAS20)</td>
</tr>
<tr>
<td></td>
<td>A Unified model of context-sensitive methods</td>
<td>Mature accomplishment (CSUR21)</td>
</tr>
<tr>
<td></td>
<td>Interprocedural SSA form</td>
<td></td>
</tr>
<tr>
<td>Precise heap abstraction</td>
<td>Liveness analysis of heap</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Combined allocation site and access path 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>Precise virtual call resolution</td>
<td>Mature accomplishment (SCP20)</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 Non-Scalability

<table>
<thead>
<tr>
<th>Desired Abstraction</th>
<th>Enabling Abstraction</th>
<th>Status of our work</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Flow- and Field-sensitivity</strong></td>
<td>Joint liveness and points-to analysis</td>
<td>Partial accomplishment (SAS12)</td>
</tr>
<tr>
<td></td>
<td>Bypassing irrelevant calls for liveness and points-to analysis</td>
<td>Work in progress</td>
</tr>
<tr>
<td></td>
<td>Synergistic program analyses</td>
<td>Work in progress</td>
</tr>
<tr>
<td></td>
<td>Partially path-sensitive analysis</td>
<td>Mature accomplishment (CC18, CC19)</td>
</tr>
<tr>
<td><strong>Context-sensitivity</strong></td>
<td>Value contexts</td>
<td>Mature accomplishment (CC08, SOAP13)</td>
</tr>
<tr>
<td>(actually caller- sensitivity)</td>
<td>GPG based bottom-up summary flow functions</td>
<td>Mature accomplishment (SAS16, TOPLAS20)</td>
</tr>
<tr>
<td></td>
<td>A Unified model of context-sensitive methods</td>
<td>Mature accomplishment (CSUR21)</td>
</tr>
<tr>
<td></td>
<td>Interprocedural SSA form</td>
<td>Work in progress</td>
</tr>
<tr>
<td><strong>Precise heap abstraction</strong></td>
<td>Liveness analysis of heap</td>
<td>Partial accomplishment (TOPLAS07)</td>
</tr>
<tr>
<td></td>
<td>Combined allocation site and access path 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>Precise virtual call resolution</td>
<td>Mature accomplishment (SCP20)</td>
</tr>
</tbody>
</table>
## In Search of Abstractions for Precision Without Non-Scalability

<table>
<thead>
<tr>
<th>Desired Abstraction</th>
<th>Enabling Abstraction</th>
<th>Status of our work</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Flow- and Field-sensitivity</strong></td>
<td>Joint liveness and points-to analysis</td>
<td>Partial accomplishment (SAS12)</td>
</tr>
<tr>
<td></td>
<td>Bypassing irrelevant calls for liveness and points-to analysis</td>
<td>Work in progress</td>
</tr>
<tr>
<td></td>
<td>Synergistic program analyses</td>
<td>Work in progress</td>
</tr>
<tr>
<td></td>
<td>Partially path-sensitive analysis</td>
<td>Mature accomplishment (CC18, CC19)</td>
</tr>
<tr>
<td><strong>Context-sensitivity (actually caller-sensitivity)</strong></td>
<td>Value contexts</td>
<td>Mature accomplishment (CC08, SOAP13)</td>
</tr>
<tr>
<td></td>
<td>GPG based flow functions</td>
<td>Mature accomplishment (SAS16, ...)</td>
</tr>
<tr>
<td></td>
<td>A Unified model of methods</td>
<td>Mature accomplishment (CSUR21)</td>
</tr>
<tr>
<td></td>
<td>Interprocedural SSA forms</td>
<td></td>
</tr>
<tr>
<td><strong>Precise heap abstraction</strong></td>
<td>Liveness analysis of heap</td>
<td>Mature accomplishment (TOPLAS07)</td>
</tr>
<tr>
<td></td>
<td>Combined allocation site and access path 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>Precise virtual call resolution</td>
<td>Mature accomplishment (SCP20)</td>
</tr>
</tbody>
</table>

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

Examples of some research explorations in

• Intraprocedural Analysis
  ◦ Combined allocation site and access path abstraction for heap
  ◦ Liveness analysis of heap data
  ◦ Liveness-based points-to analysis
  ◦ Synergistic program analysis
  ◦ Partially path-sensitive analysis

• Interprocedural analysis
  ◦ Broad categories of interprocedural analysis
  ◦ Scaling top-down analysis using value contexts and bypassing
  ◦ Improving bottom-up analysis by eliminating control flow
  ◦ Precise virtual call resolution with demand-driven analysis
  ◦ Improving call graphs using callee contexts
Intraprocedural Analysis
An Outline of Research Explorations in Intraprocedural Analysis

- Combined allocation site and access path abstraction for heap
- Liveness analysis of heap data
- Liveness-based points-to analysis
- Synergistic program analysis
- Partially path-sensitive analysis
Towards A More Precise Heap Abstraction

• Challenges in static analysis of heap pointer variables
  o Unpredictable lifetime
  o Unbounded number of allocations
  o Unnamed locations

• Unbounded heap memory can be summarized by creating a finite number of abstract nodes where each abstract node represents multiple concrete nodes

• Three options to create a finite number of abstract locations with compile-time names
  1. Represent all heap locations by a single abstract heap location
  2. Represent all heap locations of a particular type by a single abstract heap location
  3. Represent all heap locations allocated at a given memory allocation site by a single abstract heap location
Allocation Site Based Abstraction of Heap Memory

Program

1: x = new A;
2: y = x

4: y → b = new B;
5: y = y → b
6: y → a = new A;
7: y = y → a

8: assert (y → b ≠ NULL)
Allocation Site Based Abstraction of Heap Memory

Program

1: \texttt{x = new A;}
2: \texttt{y = x}

\begin{itemize}
  \item Assume that the constructors initialize all pointers (within the allocated object) to NULL
\end{itemize}

4: \texttt{y \rightarrow b = new B;}
5: \texttt{y = y \rightarrow b}
6: \texttt{y \rightarrow a = new A;}
7: \texttt{y = y \rightarrow a}
8: \texttt{assert (y \rightarrow b \neq NULL)}

Memory graphs at statement 8 in different executions

\begin{itemize}
  \item Assume that the constructors initialize all pointers (within the allocated object) to NULL
\end{itemize}
Allocation Site Based Abstraction of Heap Memory

Program

1: \( x = \text{new } A; \)
2: \( y = x \)

4: \( y \rightarrow b = \text{new } B; \)
5: \( y = y \rightarrow b \)
6: \( y \rightarrow a = \text{new } A; \)
7: \( y = y \rightarrow a \)

8: assert \( (y \rightarrow b \neq \text{NULL}) \)

Memory graphs at statement 8 in different executions

- Assume that the constructors initialize all pointers (within the allocated object) to NULL
- Object A has a member field \( b \) that contains a pointer to object B
- Object B has a member field \( a \) that contains a pointer to object A
Program

1: \( x = \text{new } A; \)
2: \( y = x \)

4: \( y \rightarrow b = \text{new } B; \)
5: \( y = y \rightarrow b \)
6: \( y \rightarrow a = \text{new } A; \)
7: \( y = y \rightarrow a \)

8: \text{assert } (y \rightarrow b \neq \text{NULL})

Memory graphs at statement 8 in different executions

Allocation Site Based Abstraction of Heap Memory
Allocation Site Based Abstraction of Heap Memory

Program

1: \( x = \text{new } A \);  
2: \( y = x \)

4: \( y \rightarrow b = \text{new } B \);  
5: \( y = y \rightarrow b \)  
6: \( y \rightarrow a = \text{new } A \);  
7: \( y = y \rightarrow a \)

8: \text{assert } (y \rightarrow b \neq \text{NULL})

Memory graphs at statement 8 in different executions

Allocation Site Based Abstraction of Heap Memory
Program

1: \( x = \text{new } A; \)
2: \( y = x \)

4: \( y \rightarrow b = \text{new } B; \)
5: \( y = y \rightarrow b \)
6: \( y \rightarrow a = \text{new } A; \)
7: \( y = y \rightarrow a \)

8: assert \( (y \rightarrow b \neq \text{NULL}) \)

Memory graphs at statement 8 in different executions
Program

1: \( x = \text{new } A; \)
2: \( y = x \)

4: \( y \rightarrow b = \text{new } B; \)
5: \( y = y \rightarrow b \)
6: \( y \rightarrow a = \text{new } A; \)
7: \( y = y \rightarrow a \)

8: assert \( (y \rightarrow b \neq \text{NULL}) \)

Memory graphs at statement 8 in different executions

The assertion is false because \( y \rightarrow b \) is always NULL in statement 8
Summarizing the Unbounded Heap

We discuss the following strategies for summarizing the unbounded heap

1. **Type Abstraction (TA).** Summarization using types by creating a single abstract node per type representing all locations of the same type

2. **Allocation Site Abstraction (ASA).** Summarization using allocation sites by creating a single abstract node per allocation site representing all locations allocated at the same site

3. **Access Path Abstraction (APA).** Summarization using access paths (which also need summarization) where each summarized access path denotes a single abstract node representing all locations reached by the access path
   3.1 Summarization of access paths using finite automata
   3.2 Summarization of access paths using $k$-limiting
   3.3 Summarization of access paths using at most $k$ repetitions of a field

4. **Combined Allocation Site and Access Path Abstraction (CASAPA).** Summarization using allocation sites and access paths together
Summarizing Heap at Statement 8 Using Type Abstraction

\[ l_1 \rightarrow l_2 \rightarrow l_3 \]
\[ l_4 \rightarrow l_5 \rightarrow l_6 \rightarrow l_7 \rightarrow l_8 \]
\[ l_9 \rightarrow l_{10} \rightarrow l_{11} \rightarrow l_{12} \rightarrow l_{13} \rightarrow l_{14} \rightarrow l_{15} \]
\[ l_{16} \rightarrow l_{17} \rightarrow l_{18} \rightarrow l_{19} \rightarrow l_{20} \rightarrow \ldots \rightarrow l_n \]
Summarizing Heap at Statement 8 Using Type Abstraction
Summarizing Heap at Statement 8 Using Type Abstraction
Summarizing Heap at Statement 8 Using Type Abstraction

Since node A has an out edge labelled $b$, we cannot guarantee that $y \rightarrow b$ is NULL. Hence we cannot prove the assertion. Besides, $x$ and $y$ are not aliased at statement 8.
Summarizing Heap at Statement 8 Using Allocation Sites

\[ \text{Heap at Statement 8} \]

\[ \begin{array}{c}
  l_1 & l_2 & l_3 \\
  x & b & a & b \\
  y & & & \\
  l_4 & l_5 & l_6 & l_7 & l_8 \\
  x & b & a & b & a & b \\
  y & & & & \\
  l_9 & l_{10} & l_{11} & l_{12} & l_{13} & l_{14} & l_{15} \\
  x & b & b & b & b & a & b \\
  y & & & & & & \\
  \cdots
  & l_{16} & l_{17} & l_{18} & l_{19} & l_{20} & \cdots & l_n \\
  x & b & b & b & b & b & \cdots & b \\
  y & & & & & & \\
\end{array} \]
Summarizing Heap at Statement 8 Using Allocation Sites
Summarizing Heap at Statement 8 Using Allocation Sites

Diagram showing the summarization process with various allocation sites and connections between them.
Summarizing Heap at Statement 8 Using Allocation Sites

Since node 6 has an out edge labelled $b$, we cannot guarantee that $y \rightarrow b$ is NULL.
Hence we cannot prove the assertion.
However, this is more precise than type abstraction because $x$ and $y$ are not aliased.
• An access path is sequence of field names starting with a root variable
  ○ Some examples are:  \(x\),  \(x \cdot a \cdot a\),  \(x \cdot a \cdot b \cdot a \ldots b \cdot \ldots a \cdot b\)
  ○ The last access path is unbounded
  ○ It represents multiple access paths that have the same pattern but different lengths

• We use three methods of summarizing unbounded access paths
  ○ Summarization of access paths using finite automata (or regular expressions)
  ○ Summarization of access paths using \(k\)-limiting
  ○ Summarization of access paths using at most \(k\) repetitions of a field
Summarizing Heap at Statement 8 Using Access Paths

Paths in the Memory

Corresponding Access Paths

\[ x \cdot b \cdot a \]
\[ y \]
\[ x \cdot b \cdot a \cdot b \cdot a \]
\[ y \]
\[ x \cdot b \cdot a \cdot b \cdot a \cdot b \cdot a \]
\[ y \]
\[ x \cdot b \cdot a \cdot b \cdot a \cdot b \cdot a \cdot b \cdot a \]
\[ y \]
• The set of out access paths is
\{y, \ x \cdot b \cdot a \cdot b \cdot a, \ x \cdot b \cdot a \cdot b \cdot a \cdot b \cdot a, \ x \cdot b \cdot a \cdot b \cdot a \cdots b\}

• Since each prefix represents a subpath reaching a memory node appearing on the path, a set of access paths is considered prefix-closed and thus our set is \{y, \ x \cdot b \cdot a \cdot b \cdot a \cdots b\}
Summarizing Heap at Statement 8 Using Access Paths

- The set \( \{y, x \cdot b \cdot a \cdot b \cdot a \cdot \ldots \cdot b\} \) can be summarized in three ways
  - With \( k \)-limiting, for \( k = 2 \), it becomes
    \[
    L_2 = \{y, x \cdot b \cdot a \cdot \#^*\}
    \]
    where “\#” is the wild card symbol representing any field and \( * \) is the kleene closure operator
  - With at most \( k \) repetitions of a field, for \( k = 2 \), it becomes
    \[
    R_2 = \{y, x \cdot b \cdot a \cdot b \cdot \#^*\}
    \]
    where “\#” is the wild card symbol and \( * \) is the kleene closure operator
  - With finite automata represented by regular expressions, it becomes
    \[
    RE = \{y, x \cdot b, x \cdot (b \cdot a)^+\}
    \]
    where \( + \) is the positive closure operator
    \( (x \cdot b \) is separated from \( x \cdot (b \cdot a)^+ \) because \( y \) is not aliased to \( x \cdot b \))

- In each case, we create appropriate abstract nodes depending upon the sets of access paths reaching a memory location represented by the abstract node.
Summarizing Heap at Statement 8 Using Access Paths

Summarization using 2-limiting with $L_2 = \{y, x \cdot b \cdot a \cdot \#^*\}$
Summarizing Heap at Statement 8 Using Access Paths

Summarization using 2-limiting with $L_2 = \{y, x \cdot b \cdot a \cdot \#^*\}$

Since node $S_3$ has an out edge labelled $\#$, we cannot guarantee that $y \rightarrow b$ is NULL and hence we cannot prove the assertion.
Summarizing Heap at Statement 8 Using Access Paths

Summarization using at most 2 repetitions of any field with $R_2 = \{ y, x \cdot b \cdot a \cdot b \cdot \# \}$
Summarizing Heap at Statement 8 Using Access Paths

Summarization using at most 2 repetitions of any field with $R_2 = \{y, x \cdot b \cdot a \cdot b \cdot \#^*\}$

Since node $S_4$ has an out edge labelled $\#$, we cannot guarantee that $y \rightarrow b$ is NULL and hence we cannot prove the assertion.
Summarizing Heap at Statement 8 Using Access Paths

Summarization using finite automata with $RE = \{y, x \cdot b, x \cdot (b \cdot a)^+\}$
Summarizing Heap at Statement 8 Using Access Paths

Summarization using finite automata with $RE = \{y, x \cdot b, x \cdot (b \cdot a)^+\}$

Since node $S_3$ has an out edge labelled $b$, we cannot guarantee that $y \rightarrow b$ is NULL and hence we cannot prove the assertion.

Summarization using regular expressions gives the most precise summarized access paths.
Summarizing Heap at Statement 8 Using Combined Allocation Sites and Access Paths Abstraction (CASAPA) [ISMM17]

- Allocation site abstraction (ASA) groups locations that are allocated at the same site in the hope that they will be used in a similar manner.
- Access path abstraction (APA) groups locations that are accessed similarly and the program text cannot distinguish between them statically.
- Sometimes both these are too coarse so we can combine the two to get the best of both the worlds.

The idea is to make further subdivisions of the sets of locations for an allocation site using access paths.
Summarizing Heap at Statement 8 Using Combined Allocation Sites and Access Paths Abstraction (CASAPA) [ISMM17]

The key idea behind CASAPA

- Use allocation sites to partition all memory locations
  Let $L_n$ be the set of locations allocated at site $n$ 
  In general, $L_n$ may be unbounded

- Partition $S_n$ on the basis of sets of access paths reaching locations in $S_n$
  - We can distinguish between two nodes statically if different sets of access paths reach them
  - Otherwise, we cannot distinguish between them

Make the distinctions that we can, merge other locations into a single abstract location
Summarizing Heap at Statement 8 Using Combined Allocation Sites and Access Paths Abstraction (CASAPA) [ISMM17]
We split allocation site node 6 into 6 & 6' using access paths.
Summarizing Heap at Statement 8 Using Combined Allocation Sites and Access Paths Abstraction (CASAPA) [ISMM17]

We split allocation site node 6 into 6 & 6' using access paths.
Summarizing Heap at Statement 8 Using Combined Allocation Sites and Access Paths Abstraction (CASAPA) [ISMM17]

We split allocation site node 6 into 6 & 6' using access paths.

AS | AP Set
---|---
1 | \{x\}
4 | \{x \cdot b \cdot (a \cdot b)^*\}
6 | \{x \cdot (b \cdot a)^+\}
6' | \{y, x \cdot (b \cdot a)^+\}
Summarizing Heap at Statement 8 Using Combined Allocation Sites and Access Paths Abstraction (CASAPA) [ISMM17]

We split allocation site node 6 into 6 & 6' using access paths.

<table>
<thead>
<tr>
<th>AS</th>
<th>AP Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{x}</td>
</tr>
<tr>
<td>4</td>
<td>{x \cdot b \cdot (a \cdot b)^*}</td>
</tr>
<tr>
<td>6</td>
<td>{x \cdot (b \cdot a)^+}</td>
</tr>
<tr>
<td>6'</td>
<td>{y, x \cdot (b \cdot a)^+}</td>
</tr>
</tbody>
</table>
Summarizing Heap at Statement 8 Using Combined Allocation Sites and Access Paths Abstraction (CASAPA) [ISMM17]

- We split allocation site node 6 into 6 & 6' using access paths.

- It is easy to see that CASAPA is provably at least as good as ASA and APA and possibly better in many cases.

- The actual precision gain depends on the summarization technique used for access paths.

- In the worst case, we may not be able to partition the locations of allocation sites using access paths.
  - In the worst case, it is as bad as ASA but no worse.

- Alternatively, in the worst case, we may not be able to partition the locations of a set of access paths by using allocation sites.
  - In the worst case, it is as bad as APA but no worse.
An Outline of Research Explorations in Intraprocedural Analysis

- Combined allocation site and access path abstraction for heap
- **Liveness analysis of heap data**
- Liveness-based points-to analysis
- Synergistic program analysis
- Partially path-sensitive analysis
Liveness Analysis of Heap Data

• Problem.

• Our Objectives.

• Main Challenge.

• Our Key Idea.

• Current status.

• Further Work.
Liveness Analysis of Heap Data

- **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.**
Liveness Analysis of Heap Data

- **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.**
Liveness Analysis of Heap Data

• **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 (\( x.data < max \))
3. \( x = x.rptr \)
4. \( y = x.lptr \)
5. \( z = \text{New class of } z \)
6. \( y = y.lptr \)
7. \( z.sum = x.data + y.data \)
8. return \( z.sum \)
Which Heap Memory Nodes Can be Statically Marked as Live?

If the while loop is executed once.

```java
1 w = x // x points to m
2 while (x.data < max)
3 x = x.rptr
4 y = x.lptr
5 z = New class_of_z
6 y = y.lptr
7 z.sum = x.data + y.data
8 return z.sum
```
If the while loop is executed twice.

```plaintext
1. w = x  // x points to m_a
2. while (x.data < max)
3.   x = x.rptr
4. y = x.lptr
5. z = New class of z
6. y = y.lptr
7. z.sum = x.data + y.data
8. return z.sum
```
Liveness Analysis of Heap Data [TOPLAS07]

- **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.**
Liveness Analysis of Heap Data [TOPLAS07]

- **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.**
Liveness Analysis of Heap Data [TOPLAS07]

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.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 = null
8  return z.sum
   z = null

While loop is not executed even once
Liveness Analysis of Heap Data [TOPLAS07]

1. \( y = z = \text{null} \)
2. \( w = x \)
   \( w = \text{null} \)
3. \( \text{while} \ (x.\text{data} < \text{max}) \)
   \{ \( x.\text{lptr} = \text{null} \)
     \( x = x.\text{rptr} \) \}
   \( x.\text{rptr} = x.\text{lptr}.\text{rptr} = \text{null} \)
   \( x.\text{lptr}.\text{lptr}.\text{rptr} = \text{null} \)
4. \( y = x.\text{lptr} \)
   \( x.\text{lptr} = y.\text{rptr} = \text{null} \)
   \( y.\text{lptr}.\text{lptr} = y.\text{lptr}.\text{rptr} = \text{null} \)
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. \( z.\text{sum} = x.\text{data} + y.\text{data} \)
   \( x = y = \text{null} \)
8. \( \text{return } z.\text{sum} \)
   \( z = \text{null} \)

While loop is not executed even once
Liveness Analysis of Heap Data [TOPLAS07]

```plaintext
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 = 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 = null
8 return z.sum
   z = null
```

While loop is not executed even once
Liveness Analysis of Heap Data [TOPLAS07]

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 of z
  z.lptr = z.rptr = null
5 y = y.lptr
  y.lptr = y.rptr = null
6 y = y.lptr
  y.lptr = y.rptr = null
7 z.sum = x.data + y.data
  x = y = null
8 return z.sum
  z = null

While loop is not executed even once
y = z = null
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 = null
8 return z.sum
   z = null

While loop is not executed even once
Liveness Analysis of Heap Data [TOPLAS07]

```
y = z = null
w = x
w = null
2 while (x.data < max)
    { x.lptr = null
      x.rptr = x.lptr.rptr = null
      x.lptr.lptr.lptr = null
      x.lptr.lptr.rptr = null
      3 x = x.rptr
  }
x.rptr = z.lptr.rptr = null
x.lptr.lptr.lptr = null
x.lptr.lptr.rptr = null
y = x.lptr
x.lptr = y.rptr = null
y.lptr.lptr = y.lptr.rptr = null
z = New class_of_z
z.lptr = z.rptr = null
6 y = y.lptr
y.lptr = y.rptr = null
z.sum = x.data + y.data
x = y = null
7 return z.sum
8 z = null
```

While loop is not executed even once
Liveness Analysis of Heap Data [TOPLAS07]

```
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 = x.lptr.rptr = null
3   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 = null
8   return z.sum
   z = null
```

While loop is not executed even once
Liveness Analysis of Heap Data [TOPLAS07]

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
3 x.lptr = y.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 = null
8 return z.sum
   z = null

While loop is executed once
Liveness Analysis of Heap Data [TOPLAS07]

```
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 = x.lptr.rptr = null
3 y = x.lptr
   x.lptr = y.rptr = null
   y.lptr.lptr = y.lptr.rptr = null
4 z = New class_of_z
   z.lptr = z.rptr = null
5 y = y.lptr
   y.lptr = y.rptr = null
6 z.sum = x.data + y.data
   x = y = null
7 return z.sum
8 z = null
```

While loop is executed twice
Liveness Analysis of Heap Data [TOPLAS07]

- **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.**
Liveness Analysis of Heap Data [TOPLAS07]

- **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 Analysis of Heap Data [TOPLAS07]

- **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.
Research Explorations in Intraprocedural Analysis

- Combined allocation site and access path abstraction for heap
- Liveness analysis of heap data
- Liveness-based points-to analysis
- Synergistic program analysis
- Partially path-sensitive analysis
Our Motivating Example for FCPA

int w;
int *u, *v, *x;
int **y, **z;

\[
y = v \\
z = u \\
x = w
\]

\[
*y = x \\
z = y
\]

\[
*z = x \\
Use u
\]
int w;
int *u, *v, *x;
int **y, **z;

Is All This Information Useful

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

\[
*z = x \\
z = y \\
\]

\[
\text{Use } u \\
\]
int w;
int *u, *v, *x;
int **y, **z;

Is All This Information Useful

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

\[
* z = x \\
z = y
\]

\[
\text{Use } u
\]
int w;
int *u, *v, *x;
int **y, **z;

Is All This Information Useful
int w;
int *u, *v, *x;
int **y, **z;

\[
y = &v \\
z = &u \\
x = &w
\]
int w;
int *u, *v, *x;
int **y, **z;

Is All This Information Useful

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

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

Use u
int w;
int *u, *v, *x;
int **y, **z;

Is All This Information Useful

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

\[ *z = x \]
\[ z = y \]

Use \( u \)
int w;
int *u, *v, *x;
int **y, **z;

Is All This Information Useful

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

\[
x \quad \text{n1}
\]

\[
z \quad u
\]

\[
\star z = x \quad \text{n2}
\]

\[
z = y \quad \text{n3}
\]

\[
w \quad \text{u}
\]

\[
\text{Use } u \quad \text{n4}
\]
Liveness-Based Points-to Analysis (LFCPA) [SAS12]

- 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 strong liveness
  - Use of a pointer in a non-assignment statement
  - Indirect pointer assignment statement
Motivating Example Revisited [SAS12]

- For convenience, we show complete sweeps of liveness and points-to analysis repeatedly.
- This is not required by the computation.
- The data flow equations define a single fixed point computation.
First Round of Liveness Analysis and Points-to Analysis

```c
int w;
int *u, *v, *x;
int **y, **z;
```

```
y = &v
z = &u
x = &w

*z = x
```

```
Use u
```

References
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```
First Round of Liveness Analysis and Points-to Analysis

```c
int w;
int *u, *v, *x;
int **y, **z;
```

Diagram:

```
\begin{center}
\begin{tikzpicture}
  \node (n1) {n_1};
  \node (n2) [below of=n1] {n_2};
  \node (n3) [right of=n1] {n_3};
  \node (n4) [below of=n2, anchor=north] {Use ~u};
  \draw[<->, thick, blue] (n1) -- (n2);
  \draw[<->, thick, blue] (n1) -- (n3);
  \draw[<->, thick, blue] (n2) -- (n4);
  \draw[<->, thick, blue] (n3) -- (n4);
\end{tikzpicture}
\end{center}
```

1. \( y = \& v \)
2. \( z = \& u \)
3. \( x = \& w \)
4. \( *z = x \)
5. \( \{ u \} \)
First Round of Liveness Analysis and Points-to Analysis

int w;
int *u, *v, *x;
int **y, **z;

Liveness Analysis

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

\[
\ast z = x \\
z = y \\
\]

Use u

\{u\}
First Round of Liveness Analysis and Points-to Analysis

```c
int w;
int *u, *v, *x;
int **y, **z;

y = &v
z = &u
x = &w

Liveness Analysis

{n1}

{z}   {u}
n2

*z = x

{n3}

z = y

{n4}

Use u

{n4}

{u}

{n4}

{u}

{n4}

{u}

{n4}

{u}
```
First Round of Liveness Analysis and Points-to Analysis

int w;
int *u, *v, *x;
int **y, **z;

Liveness Analysis

Liveness of u is killed because pointees of z are not known
z is made live

Strong liveness: y is not made live because z is not live

44/99
First Round of Liveness Analysis and Points-to Analysis

```c
int w;
int *u, *v, *x;
int **y, **z;
```

```
y = &v
z = &u
x = &w
```

Liveness Analysis

```
{z}
{n_2}
{u}
```

```
{u}
{n_3}
```

```
{u, z}
{n_1}
```

```
{u, z}
{n_1}
```

```
{u}
{n_4}
```

```
{u}
{n_4}
```

```
Use u
{n_4}
```

References
First Round of Liveness Analysis and Points-to Analysis

```c
int w;
int *u, *v, *x;
int **y, **z;
```

```
1
y = &v
2
z = &u
3
x = &w
```

```
// Liveness Analysis
4
{u}
5
{u, z}
6
{z}
7
{u}
8
n1
```

```
// Points-to Analysis
9
*n2 = z
10
z = y
```

```
// Uses
11
Use u
```

int w;
int *u, *v, *x;
int **y, **z;

First Round of Liveness Analysis and Points-to Analysis

\[
\begin{align*}
&\text{int } w; \\
&\text{int } *u, *v, *x; \\
&\text{int } **y, **z;
\end{align*}
\]
First Round of Liveness Analysis and Points-to Analysis

```c
int w;
int *u, *v, *x;
int **y, **z;

int w;
int *u, *v, *x;
int **y, **z;

{u}
{u, z}

{z}

{u}

{u}

{u}

{u}

Use u
```

```
\begin{align*}
y &= \& v \\
z &= \& u \\
x &= \& w
\end{align*}
```

```
n_1

\begin{align*}
y &= \& v \\
z &= \& u \\
x &= \& w
\end{align*}
```

```
n_2

*\texttt{z} = \texttt{x}
```

```
n_3

z = y
```

```
n_4

\text{Use } u
```
First Round of Liveness Analysis and Points-to Analysis

```c
int w;
int *u, *v, *x;
int **y, **z;

n1
y = &v
z = &u
x = &w

\{u, z\}
\{u\}
\{u\}
\{u\}

n2
*z = x

n3
z = y

\{u\}

n4
Use u
```

Points-to Analysis:
- `y = &v` \{u\} \(\rightarrow\) ?
- `z = &u` \{u, z\} \(\rightarrow\) ?
- `*z = x` \{u\} \(\rightarrow\) ?
- `z = y` \{u\} \(\rightarrow\) ?

References
First Round of Liveness Analysis and Points-to Analysis

```c
int w;
int *u, *v, *x;
int **y, **z;

y = &v
z = &u
x = &w

n1

{u}  u → ?

{u, z}  z → u → ?

*z = x

n2

{z}  u → ?

{n}

{n}

{n}

{n}

n3

z = y

{n}

{n}

{n}

{n}

{n}

{n}

n4

Use u
```

**Intraprocedural Analysis**

**Interprocedural Analysis**

**Conclusions**

**References**
First Round of Liveness Analysis and Points-to Analysis

\[
\begin{align*}
&\text{int } w; \\
&\text{int } *u, *v, *x; \\
&\text{int } **y, **z;
\end{align*}
\]

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

\[
\begin{align*}
&\{u\} \quad u \rightarrow ? \\
&\{u, z\} \quad z \rightarrow u \rightarrow ? \\
&\{z\} \\
&\{u\} \quad \text{Use } u \\
&\{u\} \quad u \rightarrow ? \\
&\{u\} \quad \text{Use } u \\
&\{u\} \quad \text{Use } u
\end{align*}
\]
int w;
int *u, *v, *x;
int **y, **z;

First Round of Liveness Analysis and Points-to Analysis

\begin{align*}
    n_1: & y = \&v \\
    & z = \&u \\
    & x = \&w \\
    & \{u\} \\
    & \{u, z\} \\
    & \text{?} \\
\end{align*}

\begin{align*}
    n_2: & \ast z = x \\
    & \{u\} \\
    & \text{?} \\
\end{align*}

\begin{align*}
    n_3: & z = y \\
    & \{u\} \\
    & \text{?} \\
\end{align*}

\begin{align*}
    n_4: & \text{Use } u \\
    & \{u\} \\
\end{align*}
First Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```

```
y = &v
z = &u
x = &w
```

```
\text{Points-to Analysis:}
\{u\} \rightarrow \{x\} \rightarrow \{u\} \rightarrow \{z\} \rightarrow u \rightarrow ?
```

```
z = y
```

```
\text{Use } u
```

Reference:

44/99
int w;
int *u, *v, *x;
int **y, **z;

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

First Round of Liveness Analysis and Points-to Analysis

Since \( z \) is not live its points-to relations are killed.
First Round of Liveness Analysis and Points-to Analysis

```c
int w;
int *u, *v, *x;
int **y, **z;
```

```
\( y = &v \)
\( z = &u \)
\( x = &w \)
```

Points-to Analysis

```
\( z \rightarrow u \)
\( \{z\} \)
```

```
\( \ast z = x \)
```

```
\( z = y \)
```

Since \( z \) is not live its points-to relations are killed

```
\{u\}
```

```
Use \( u \)
```

```c
u
```

References

Conclusions

Interprocedural Analysis

Talk Title: PSPA Research

Uday Khedker
IIT Bombay

Topic: Some Meanderings
int w;
int *u, *v, *x;
int **y, **z;

Second Round of Liveness Analysis and Points-to Analysis

\[ n_1 \]
\[
\begin{align*}
    & y = & v \\
    & z = & u \\
    & x = & & w
\end{align*}
\]
\[
\{ u \} \rightarrow ?
\]
\[
\{ u, z \} \rightarrow ?
\]
\[
\{ z \} \rightarrow u
\]
\[
\{ u \} \rightarrow ?
\]
\[
\{ u \} \rightarrow ?
\]
\[
\{ u \} \rightarrow ?
\]

\[ n_2 \]
\[
\begin{align*}
    & \*z = & x
\end{align*}
\]
\[
\{ u \} \rightarrow ?
\]

\[ n_3 \]
\[
\begin{align*}
    & z = & y
\end{align*}
\]
\[
\{ u \} \rightarrow ?
\]

\[ n_4 \]
\[
\text{Use } u
\]
\[
\{ u \} \rightarrow ?
\]
int w;
int *u, *v, *x;
int **y, **z;

Second Round of Liveness Analysis and Points-to Analysis

\begin{align*}
y &= & v \\
z &= & u \\
x &= & w \\
\end{align*}

\begin{tikzpicture}
  \node (n1) {\(y = \& v\)};
  \node (n2) {\(z = \& u\)};
  \node (n3) {\(x = \& w\)};
  \node (n4) {Use \(u\)};

  \draw [->, thick] (n1) -- (n2) node [midway, above] {\(u, z\)};
  \draw [->, thick] (n2) -- (n3) node [midway, above] {\(u\)};
  \draw [->, thick] (n3) -- (n4) node [midway, above] {\(u\)};

  \draw [->, thick] (n4) -- (n1) node [midway, above] {\(u\)};

  \draw [->, thick] (n2) -- (n4) node [midway, above] {\(u\)};

  \draw [->, thick] (n3) -- (n4) node [midway, above] {\(u\)};

  \draw [->, thick] (n4) -- (n1) node [midway, above] {\(u\)};

\end{tikzpicture}
Second Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```

```
y = &v  
z = &u  
x = &w
```

```
\{u, x, z\}  
z -> u  
\{u\}  
u -> ?
```

```
\{x, z\}  
z -> u
```

```
\{u\}  
u -> ?
```

\(n_1\)

```
*\(z\) = \(x\)  
\{u\}  
\(u\) -> ?
```

\(n_2\)

```
z = \(y\)  
\{u\}  
\(u\) -> ?
```

\(n_3\)

\(\{u\}\)

\(n_4\)

```
Use \(u\)  
\{u\}  
\(u\) -> ?
```
Second Round of Liveness Analysis and Points-to Analysis

```c
int w;
int *u, *v, *x;
int **y, **z;
```

- `y = &v`
- `z = &u`
- `x = &w`

Points-to Analysis:
- `z` points to `u` and `z`.
- `x` points to `z`.
- `w` points to `u`.

Use of `u`:
```c
Use u
```

Conclusions and References.
Second Round of Liveness Analysis and Points-to Analysis

```c
int w;
int *u, *v, *x;
int **y, **z;

x = &w
z = &u
y = &v

n1

Points-to Analysis

\( x \to w \)
\( z \to u \)
\( \{ x, z \} \)

n2

\*z = x
\{ u, x, z \}

n3

z = y
\{ u \}

n4

Use u
\{ u \}

References

Conclusions

Interprocedural Analysis

Some Meanderings

Talk Title: PSPA Research

Uday Khedker
IIT Bombay
Second Round of Liveness Analysis and Points-to Analysis

int w;
int *u, *v, *x;
int **y, **z;

\[
\begin{align*}
\text{\texttt{n1}} & : & y &= \& v \\
& & z &= \& u \\
& & x &= \& w
\end{align*}
\]

\[
\begin{align*}
\text{\texttt{n2}} & : & *z &= \text{x} \\
\text{\texttt{n3}} & : & z &= y
\end{align*}
\]

Points-to Analysis

Use u

References
Second Round of Liveness Analysis and Points-to Analysis

```
int w;
int *u, *v, *x;
int **y, **z;
```
• Usable pointer information is very small and sparse

• Data flow propagation in real programs seems to involve only a small subset of all possible data flow values

• Earlier approaches reported inefficiency and non-scalability because they computed far more information than the actual usable information
LFCPA 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>Percentage 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>
LFCPA 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>Percentage 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>
LF CPA 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>Percentage 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
  o LLVM (Dylan McDermott, Cambridge, 2016) and
  o GCC 4.7.2 (Priyanka Sawant, IITB, 2016)
Research Explorations in Intraprocedural Analysis

- Combined allocation site and access path abstraction for heap
- Liveness analysis of heap data
- Liveness-based points-to analysis
- **Synergistic program analysis**
- Partially path-sensitive analysis
The value of $d$ in the loop is 1, the condition fails, and $x$ does not point to $b$ at any time
Collaboration Between Constant Propagation and Points-to Analysis [WiP]

We have three options to enable interaction (illustrated next)

- **Conventional Cascading.** Perform analyses in a fixed sequence
  
  - CP $\rightarrow$ transform the program $\rightarrow$ PTA
  
  - PTA $\rightarrow$ transform the program $\rightarrow$ CP

  This method fails on our example

- **Simultaneous Analyses (Lerner’s method)**
  
  Perform CP and PTA in locked steps and transform the program whenever possible, repeat the analyses as long as transformations are possible

  This method also fails on our example

- **Interleaved Synergistic Program Analysis (SPAN)**

  Interleave the analyses on a need basis, use data flow values to achieve the effect of transforming the program (without actually transforming it)

  This method succeeds on our example
Collaboration Between Constant Propagation and Points-to Analysis [WiP]

```
a = 1
b = a - 2
c = a
x = &a
```

```
if (b < 10)
```

```
d = *x
if (d \leq 0)
```

```
x = &b
```

```
x = &c
```

```
b++
```

7 return d
If we perform constant propagation first,

- We do not know the pointees of \( x \) in node 3, hence we assume all variables as possible pointees
- Thus the value of \( d \) is \( \perp \) and the branch outcome is uncertain and no path is ruled out
Collaboration Between Constant Propagation and Points-to Analysis [WiP]

If we perform constant propagation first,

- We do not know the pointees of $x$ in node 3, hence we assume all variables as possible pointees
- Thus the value of $d$ is $\perp$ and the branch outcome is uncertain and no path is ruled out

Then, when we perform points-to analysis,

- The pointees of $x$ are found to be $a$, $b$, and $c$
- $d = *x$ cannot be simplified
- A subsequent round of constant propagation will find $d$ to be $\perp$
If we perform constant propagation first,

- We do not know the pointees of $x$ in node 3, hence we assume all variables as possible pointees
- Thus the value of $d$ is $\perp$ and the branch outcome is uncertain and no path is ruled out

Then, when we perform points-to analysis,

- The pointees of $x$ are found to be $a$, $b$, and $c$
- $d = \ast x$ cannot be simplified
- A subsequent round of constant propagation will find $d$ to be $\perp$
Collaboration Between Constant Propagation and Points-to Analysis [WiP]

If we perform points-to analysis first,

- The pointees of $x$ are found to be $a$, $b$, and $c$ because both branch outcomes are possible
- $d = \ast x$ cannot be simplified
Collaboration Between Constant Propagation and Points-to Analysis [WiP]

If we perform points-to analysis first,

- The pointees of $x$ are found to be $a$, $b$, and $c$ because both branch outcomes are possible
- $d = *x$ cannot be simplified

Then, when we perform constant propagation

- The value of $d$ is $\perp$ and the branch outcome is uncertain and no path is ruled out
- A subsequent round of points-to analysis will find the pointees of $x$ as $a$, $b$ and $c
Collaboration Between Constant Propagation and Points-to Analysis [WiP]

- The precision of the two analyses depends on each other’s results.
- If we perform them together, we can rule out the T branch out of node 3, x points to a and c, and both are 1.
Collaboration Between Constant Propagation and Points-to-Analysis [WiP]

- The precision of the two analyses depends on each other’s results
- If we perform them together, we can rule out the $T$ branch out of node 3, $x$ points to $a$ and $c$, and both are 1
- SPAN achieves this
Collaboration Between Constant Propagation and Points-to Analysis [WiP]

- The precision of the two analyses depends on each other’s results
- If we perform them together, we can rule out the T branch out of node 3, x points to a and c, and both are 1
- SPAN achieves this

```
1
 a = 1
 b = a - 2
 c = a
 x = &a
 (⊥, ⊥, ⊥, ⊥), (x, V)
 (1, -1, 1, ⊥), (x, {a})

2 if (b < 10)
   T
   d = *x
     if (d ≤ 0)
       T, T
       (1, ⊥, 1, ⊥), (x, {a, c})
     F
     (1, ⊥, 1, ⊥), (x, {a, c})
   F

3 T, T
   x = &b
4 T, T
   b++
5 x = &c
6 (1, ⊥, 1, ⊥), (x, {c})
7 return d
```
Collaboration Between Constant Propagation and Points-to Analysis [WiP]

SPAN is more general than Lerner’s method because

- SPAN does not transform the program but uses data flow values (Lerner’s method tries to transform $d = x\star x$ and fails)
- The analyses need not be performed in locked steps and hence forward and backward analyses can be combined
- The need of interaction is inferred automatically and the user does not need to specify it
- Arbitrary data flow analyses can be added to the system at will

Each analysis must specify the statements that it can (conceptually) simplify and the statements that it cannot simplify.
Research Explorations in Intraprocedural Analysis

- Combined allocation site and access path abstraction for heap
- Liveness analysis of heap data
- Liveness-based points-to analysis
- Synergistic program analysis
- Partially path-sensitive analysis
Excluding the Known Infeasible Paths [CC18, CC19]

- Every path containing $\rho:(2, 4, 6)$ is infeasible.
  It could lead to imprecision (e.g. $d$ is spuriously marked live at the exit of node 2)
- We cannot delete any edge to exclude this path

Such deletion could lead to unsoundness
Excluding the Known Infeasible Paths [CC18, CC19]

- Every path containing $\rho:(2,4,6)$ is infeasible
  It could lead to imprecision (e.g. $d$ is spuriously marked live at the exit of node 2)
- We cannot delete any edge to exclude this path
  - If we delete edge (2, 4), it excludes a feasible path (1, 2, 4, 5, 7)

Such deletion could lead to unsoundness
Excluding the Known Infeasible Paths [CC18, CC19]

- Every path containing \( \rho : (2, 4, 6) \) is infeasible.
  - It could lead to imprecision (e.g. \( d \) is spuriously marked live at the exit of node 2)
- We cannot delete any edge to exclude this path:
  - If we delete edge (2, 4), it excludes a feasible path (1, 2, 4, 5, 7)
  - If we delete edge (4, 6), it excludes a feasible path (1, 3, 4, 6, 7)

Such deletion could lead to unsoundness.
Excluding the Known Infeasible Paths [CC18, CC19]

1. if \( x < 0 \)

2. \( \text{a} = \text{a} + \text{y} \)
   \( \text{x} = \text{a} + \text{z} \)

3. • Every path containing \( \rho: (2, 4, 6) \) is infeasible
   It could lead to imprecision (e.g. \( d \) is spuriously marked live at the exit of node 2)

4. • We cannot delete any edge to exclude this path
   - If we delete edge (2, 4), it excludes a feasible path (1, 2, 4, 5, 7)
   - If we delete edge (4, 6), it excludes a feasible path (1, 3, 4, 6, 7)

5. • Such deletion could lead to unsoundness

6. • Our solution: At each edge, distinguish the data flow value of \( \rho \) from other values so that it is not allowed to go out of \( \rho \) on an infeasible path
Excluding the Known Infeasible Paths [CC18, CC19]

Our Notation: \( \langle \text{dfv of } \rho, \text{other dfv} \rangle \)

Edges 5 \( \rightarrow \) 7 and 6 \( \rightarrow \) 7 are not a part of \( \rho \)

Hence the data flow value in the first component is \( \top \)
Excluding the Known Infeasible Paths [CC18, CC19]

Our Notation: \( \langle \text{dfv of } \rho, \text{other dfv} \rangle \)

Edge 4 \( \rightarrow \) 5 is not a part of \( \rho \) and the first component is \( T \).

Edge 4 \( \rightarrow \) 6 is a part of \( \rho \) but edge 6 \( \rightarrow \) 7 is not a part of \( \rho \) (i.e. the effect of \( \rho \) begins here) so the data flow value shifts from the second component to the first component.
Excluding the Known Infeasible Paths [CC18, CC19]

Our Notation: \( \langle \text{dfv of } \rho, \text{other dfv} \rangle \)

Edge 2 \( \rightarrow \) 4 is a part of \( \rho \) hence it will continue to hold the data flow value of \( \rho \) coming from edge 4 \( \rightarrow \) 6 which is also a part of \( \rho \).

The data flow value generated in node 4 or the data flow value coming from edge 4 \( \rightarrow \) 5 go to the second component because a path that does not include \( \rho \) completely, is not infeasible.
Excluding the Known Infeasible Paths [CC18, CC19]

Our Notation: $\langle \text{dfv of } \rho, \text{other dfv} \rangle$

Edge 3 → 4 is not a part of $\rho$ hence the first component is $T$ and all data flow values move to the second component.
Excluding the Known Infeasible Paths [CC18, CC19]

Our Notation: \( \langle \text{dfv of } \rho, \text{other dfv} \rangle \)

Edge 1 → 2 is not a part of \( \rho \) hence the first component is \( \top \) and all data flow values move to the second component.

Since \( d \) belongs to \( \rho \), it is blocked and is not propagated further because the path \( (1, 2, 4, 6, 7) \) is infeasible.

This separation and blocking of values gives a more precise solution than the usual MFP solution.
Excluding the Known Infeasible Paths [CC18, CC19]

Our Notation: \( \langle \text{dfv of } \rho, \text{other dfv} \rangle \)

Edge 1 \( \rightarrow \) 3 is not a part of \( \rho \) hence the first component is \( T \) and all data flow values move to the second component.
Excluding the Known Infeasible Paths [CC18, CC19]

• Infeasibility is a property of the control flow graph and not that of an analysis

• Our method takes as input the information about the minimal infeasible path segments in program

• Our method is very general
  ○ It handles multiple minimal infeasible path segments that may overlap with each other
  ○ It lifts any data flow analysis to an analysis that excludes the effect of known infeasible paths

• Existing approaches to remove the effect of infeasible paths are either analysis specific or involve CFG restructuring
  
  Our approach avoids CFG restructuring and still achieves a generic solution
Interprocedural Analysis
Research Explorations in Interprocedural Analysis

- Broad categories of interprocedural analysis
- Scaling top-down analysis using value contexts and bypassing
- A unified model of context-sensitive methods
- Improving bottom-up analysis by eliminating control flow
- Precise virtual call resolution with demand-driven analysis
- Improving call graphs using callee contexts
Understanding Context Sensitivity

Interprocedurally valid path

\[ S_s \rightarrow C_i \rightarrow R_i \rightarrow E_s \rightarrow S_r \rightarrow C_j \rightarrow R_j \rightarrow E_t \rightarrow S_t \]
Understanding Context Sensitivity

![Diagram showing the flow of context sensitivity between different states and conditions.]

- **$S_s$**
- **$C_i$**
- **$R_i$**
- **$E_s$**
- **$S_r$**
- **$C_j$**
- **$R_j$**
- **$E_t$**

**Talk Title:** PSPA Research

**Topic:** Some Meanderings

- Intraprocedural Analysis
- Interprocedural Analysis

**Conclusions**

**References**
Precise interprocedural analysis aims to achieve the effect of inlining.
Understanding Context Sensitivity

Interprocedurally valid path
Understanding Context Sensitivity

Interprocedurally valid path
Understanding Context Sensitivity

Interprocedurally invalid path
Understanding Context Sensitivity

Interprocedurally invalid path
Context Sensitivity Vs. Context Insensitivity (Example for Live Variables Analysis)

<table>
<thead>
<tr>
<th>Context-sensitive Analysis</th>
<th>Context-insensitive Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_s$</td>
<td>$S_t$</td>
</tr>
<tr>
<td>$r()$</td>
<td>$r()$</td>
</tr>
<tr>
<td>$E_s$</td>
<td>$E_t$</td>
</tr>
<tr>
<td>Data flow values of distinct contexts are kept as separate values</td>
<td>Data flow values of all contexts are merged into a single value</td>
</tr>
</tbody>
</table>
Context Sensitivity Vs. Context Insensitivity (Example for Live Variables Analysis)

- **Context-sensitive Analysis**
  - $S_s$
  - $r()$
  - $E_s$
  - Data flow values of distinct contexts are kept as separate values

- **Context-insensitive Analysis**
  - $S_t$
  - $r()$
  - Use $a$
  - $E_t$
  - Data flow values of all contexts are merged into a single value
Context Sensitivity Vs. Context Insensitivity (Example for Live Variables Analysis)

**Context-sensitive Analysis**
- $S_s$
- $r()$
- $E_s$
- Data flow values of distinct contexts are kept as separate values

**Context-insensitive Analysis**
- $S_t$
- $r()$
- Use $a$
- $E_t$
- Data flow values of all contexts are merged into a single value
Context Sensitivity Vs. Context Insensitivity (Example for Live Variables Analysis)

**Context-sensitive Analysis**

- \( S_s \)
- \( r() \)
- \( E_s \)

**Context-insensitive Analysis**

- \( S_t \)
- \( r() \)
- \( E_t \)
- Use \( a \)

Data flow values of distinct contexts are kept as separate values

Data flow values of all contexts are merged into a single value
Context Sensitivity Vs. Context Insensitivity (Example for Live Variables Analysis)

Context-sensitive Analysis

- $S_s$
- $r()$
- $E_s$

Data flow values of distinct contexts are kept as separate values

Context-insensitive Analysis

- $S_t$
- $r()$
- Use $a$
- $E_t$

Data flow values of all contexts are merged into a single value
Context Sensitivity Vs. Context Insensitivity (Example for Live Variables Analysis)

Context-sensitive Analysis

Data flow values of distinct contexts are kept as separate values

Context-insensitive Analysis

Data flow values of all contexts are merged into a single value
Context Sensitivity Vs. Context Insensitivity (Example for Live Variables Analysis)

**Context-sensitive Analysis**

- $S_s$
- $r()$
- $S_r$
- $E_r$
- $E_s$

Data flow values of distinct contexts are kept as separate values

**Context-insensitive Analysis**

- $S_t$
- $r()$
- $S_r$
- $E_r$
- $E_s$

Data flow values of all contexts are merged into a single value

- $a = 5$
- $E_r$
- $E_s$
- $E_t$
Context Sensitivity Vs. Context Insensitivity (Example for Live Variables Analysis)

**Context-sensitive Analysis**

Data flow values of distinct contexts are kept as separate values

**Context-insensitive Analysis**

Data flow values of all contexts are merged into a single value
Context Sensitivity Vs. Context Insensitivity (Example for Live Variables Analysis)

Context-sensitive Analysis

Context-insensitive Analysis

Data flow values of distinct contexts are kept as separate values

Data flow values of all contexts are merged into a single value
Context Sensitivity Vs. Context Insensitivity (Example for Live Variables Analysis)

Data flow values of distinct contexts are kept as separate values

Data flow values of all contexts are merged into a single value
Understanding Context Sensitivity

The effect of inlining is achieved by

- call-return matching (**call strings method**),
- computing the summary of a procedure and incorporating it at the call point (**functional method**), or
- analyzing a procedure for a particular data flow value and using the analysed result at the call point (**graph reachability, value context method**)
Top-down Vs. Bottom-up (Context-Sensitive) Procedure Summaries

Top-down Live Variables Analysis

\[ \text{Start}_p \]

\[ \text{Call } q \]

\[ \text{Use } a \]

\[ \text{End}_p \]

\[ \text{Start}_q \]

\[ a = \ldots \]

\[ \text{Use } b \]

\[ \text{End}_q \]

\[ \text{Start}_r \]

\[ \text{Call } q \]

\[ \text{Use } c \]

\[ \text{End}_r \]
Top-down Vs. Bottom-up (Context-Sensitive) Procedure Summaries

Procedure $q$ is analysed multiple times

Contexts are created explicitly

Variable $b$ is live at $S_q$ and hence at $S_p$
Variables $a$ and $c$ are not live at $S_q$ and hence not live at $S_p$
Top-down Vs. Bottom-up (Context-Sensitive) Procedure Summaries

Procedure $q$ is analysed multiple times.

Contexts are created explicitly.

Context $\sigma_2$

Variables $b$ and $c$ are live at $S_q$ and hence at $S_r$.

Variable $a$ is not live at $S_q$ and hence not live at $S_r$. 

Top-down Live Variables Analysis

Start $p$ → Call $q$ → Use $a$ → End $p$
Top-down Vs. Bottom-up (Context-Sensitive) Procedure Summaries

Bottom-Up Live Variables Analysis

- **Start**$_p$
  - Call $q$
  - Use $a$
  - **End**$_p$

- **Start**$_q$
  - $a = ...$
  - Use $b$
  - **End**$_q$

- **Start**$_r$
  - Call $q$
  - Use $c$
  - **End**$_r$
Top-down Vs. Bottom-up (Context-Sensitive) Procedure Summaries

Using procedure summary of $q$ at call sites

Using procedure summary of $q$ at call sites
**Top-down Vs. Bottom-up (Context-Sensitive) Procedure Summaries**

**Bottom-Up Live Variables Analysis**

**Procedure**

$q$ is analysed once

**Contexts**

are left implicit

Variable $b$ is live at $S_p$

Variables $a$ and $c$ are not live at $S_p$
Top-down Vs. Bottom-up (Context-Sensitive) Procedure Summaries

Variables $b$ and $c$ are live at $S_r$
Variable $a$ is not live at $S_r$
• Broad categories of interprocedural analysis
• **Scaling top-down analysis using value contexts and bypassing**

  Next Topic

• A unified model of context-sensitive methods
• Improving bottom-up analysis by eliminating control flow
• Precise virtual call resolution with demand-driven analysis
• Improving call graphs using callee contexts
Value Contexts [CC08, SOAP13]
Value Contexts [CC08, SOAP13]

- Start
- Multiple interprocedural paths reaching the procedure
- End

Intraprocedural Analysis
Interprocedural Analysis
Conclusions
References
Value Contexts [CC08, SOAP13]

Multiple interprocedural paths reaching the procedure
Value Contexts [CC08, SOAP13]

Multiple interprocedural paths reaching the procedure

Start_p

End_p

Uday Khedker
IIT Bombay

Talk Title:
PSPA Research

Topic:
Some Meanderings

Intraprocedural Analysis

Interprocedural Analysis

Conclusions

References
Value Contexts [CC08, SOAP13]

Multiple interprocedural paths reaching the procedure

Start$_p$

End$_p$
Multiple interprocedural paths reaching the procedure
Value Contexts \([\text{CC08, SOAP13}]\)

Data flow values

\[\text{Start}_p\]

\[\text{End}_p\]
Value Contexts [CC08, SOAP13]

Data flow values

\[ x \rightarrow \text{Start}_p \]

\[ y \rightarrow \text{Start}_p \]

\[ x' \rightarrow \text{End}_p \]

\[ y' \rightarrow \text{End}_p \]
Value Contexts [CC08, SOAP13]
Value Contexts [CC08, SOAP13]
Value Contexts [CC08, SOAP13]

Data flow values

Start<sub>p</sub>

End<sub>p</sub>

References
Value Contexts [CC08, SOAP13]

Data flow values

Contexts

\( x \)
\( y \)
\( y' \)
\( y' \)
\( y' \)
\( z \)
\( x' \)
\( y' \)
\( y' \)
\( y' \)
\( z' \)

Start\(_p\)

End\(_p\)
Value Contexts [CC08, SOAP13]
Value Contexts [CC08, SOAP13]

Data flow values

Contexts

Start_p

End_p

\( x \)

\( y \)

\( y' \)

\( y' \)

\( y' \)

\( z \)

\( x' \)

\( y' \)

\( y' \)

\( y' \)

\( z' \)
Value Contexts [CC08, SOAP13]

Call sites
Data flow values
Contexts

Start_p
End_p

\[ c_0 \rightarrow \sigma_0 \rightarrow x \]
\[ c_1 \rightarrow \sigma_1 \rightarrow y \]
\[ c_2 \rightarrow \sigma_1 \rightarrow y' \]
\[ c_3 \rightarrow \sigma_1 \rightarrow y' \]
\[ c_4 \rightarrow \sigma_2 \rightarrow z \]
Value Contexts [CC08, SOAP13]

Contexts of the callers

(a,σ_i) → c_0 → (a,σ_j) → c_1 → (b,σ_k) → c_2 → (c,σ_l) → c_3 → (d,σ_m) → c_4 → z

Call sites

Data flow values

Contexts

σ_0 → σ_1 → σ_1 → σ_1 → σ_2

Start_p

End_p

x

y

y'

y''

z'

x'

References

Conclusions

Interprocedural Analysis

Intraprocedural Analysis

Topic:
Some Meanderings

Talk Title:
PSPA Research

Uday Khedker
IIT Bombay
value contexts [CC08, SOAP13]

Contexts of the callers

(a, σ_i)

(x, σ_0)

(a, σ_j)

(y, σ_1)

(b, σ_k)

(y, σ_1)

(c, σ_i)

(y, σ_1)

(d, σ_m)

(z, σ_2)

Call sites

Data flow values

Context-sensitive call graph

Context-sensitive call graph

Start_p

End_p

Value Contexts [CC08, SOAP13]
Value Contexts [CC08, SOAP13]

Contexts of the callers:

- \((a, \sigma_i)\)
- \((a, \sigma_j)\)
- \((b, \sigma_k)\)
- \((c, \sigma_l)\)
- \((d, \sigma_m)\)

Call sites:

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

Data flow values:

- \(X\)
- \(Y\)
- \(Y\)
- \(Y\)
- \(Z\)

Context-sensitive call graph:

- \((a, \sigma_i) \rightarrow (p, \sigma_0)\)
- \((a, \sigma_j) \rightarrow c_1\)
- \((b, \sigma_k) \rightarrow (p, \sigma_1)\)
- \((c, \sigma_l) \rightarrow c_3\)
Value Contexts [CC08, SOAP13]

Contexts of the callers

(a,σ_i) → c_0 → (p,σ_0)

(a,σ_j) → c_1

(b,σ_k) → c_2 → (p,σ_1)

(c,σ_l) → c_3

(d,σ_m) → c_4 → (p,σ_2)

Context-sensitive call graph

Call sites

Data flow values

X

Y

Y

Y

Z

Contexts

σ_0

σ_1

σ_1

σ_1

σ_2

Start_p

End_p

x'

y'

y'

y'

z'
Value Contexts [CC08, SOAP13]

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%
Empirical Observations About Value Contexts

• Reaching definitions analysis in GCC 4.2.0 (CC-2008)
  Analysis of Towers of Hanoi
    o Time brought down from $3.973 \times 10^6$ ms to 2.37 ms
    o 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
    o Average number of contexts per procedure lies in the range 4-25
    o 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%

• And yet, it is insufficient for scaling flow- and context-sensitive points-to analysis to more than 35 kLoC
Top-down Analysis With Bypassing

• Procedures $q$ and $r$ do not access $a$

• Can we avoid propagating the points-to pair $(a, b)$ through procedure $q$ (and hence through $r$)?

• How do we know which pairs should bypass a call?
  Compute the bypassing set for each procedure during the analysis
Research Explorations in Interprocedural Analysis

- Broad categories of interprocedural analysis
- Scaling top-down analysis using value contexts and bypassing
- **A unified model of context-sensitive methods**
- Improving bottom-up analysis by eliminating control flow
- Precise virtual call resolution with demand-driven analysis
- Improving call graphs using callee contexts
# Context-Sensitive Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Notion of context</th>
<th>Definition of context</th>
<th>Inlining strategy</th>
<th>Call graph traversal</th>
<th>Call-return matching</th>
<th>Restrictions on frameworks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call strings</td>
<td>Explicit</td>
<td>Call strings</td>
<td>Context-dep. top-down</td>
<td>Context</td>
<td></td>
<td>Finite $L$</td>
</tr>
<tr>
<td>VBTCS</td>
<td>Explicit</td>
<td>Call strings and data flow values</td>
<td>Context-dep. top-down</td>
<td>Context</td>
<td></td>
<td>Finite lattice</td>
</tr>
<tr>
<td>Value contexts</td>
<td>Explicit</td>
<td>Data flow values</td>
<td>Context-dep. top-down</td>
<td>Context transition diagram</td>
<td></td>
<td>Finite $L$</td>
</tr>
<tr>
<td>IFDS</td>
<td>Implicit</td>
<td>Not specified</td>
<td>Context-dep. top-down</td>
<td>By the algorithm</td>
<td></td>
<td>Distributive ffs $L$ with $\subseteq$</td>
</tr>
<tr>
<td>IDE</td>
<td>Implicit</td>
<td>Not specified</td>
<td>Context-dep. top-down</td>
<td>By the algorithm</td>
<td></td>
<td>$L$ of distributive maps</td>
</tr>
<tr>
<td>Object sensitivity</td>
<td>Explicit</td>
<td>Strings of object allocation sites</td>
<td>Context-dep. top-down</td>
<td></td>
<td>Inlining summaries</td>
<td>Finite lattice</td>
</tr>
<tr>
<td>Functional</td>
<td>None</td>
<td>None</td>
<td>Context-ind. bottom-up</td>
<td></td>
<td></td>
<td>Reducing $\sqcap$ and $\circ$ of ffs</td>
</tr>
</tbody>
</table>
Formalizing Context-Sensitive Analyses [CSUR21]

- Context-sensitive information at a program point is a pair \((\sigma, x)\) where 
  - \(\sigma\) is the context and \(x\) is the data flow value 
    - A separate value is computed for each context reaching a procedure 
    - We leave the context \(\sigma\) and the data flow value \(x\) undefined 
      - \(\sigma\) is defined by the method of performing analysis 
      - \(x\) is defined by the analysis to be performed 
- Examples of contexts 
  - A call chain (i.e. a sequence of unfinished calls) reaching a procedure 
  - A data flow reaching the start of a procedure
• Context-sensitive information at a program point is a pair \((\sigma, x)\)

• We model context-sensitive methods by defining two structures
  
  \(\text{○ An abstract value structure} \) models ways of computing \(x\)

  \[ \mathcal{V} = (M, \nu_0, \theta_n, \eta^\text{Call}_n, \eta^\text{Ret}_n, \text{Project}_Q) \]

  \(\text{○ An abstract context structure} \) models ways of computing \(\sigma\)

  \[ \mathcal{A} = (A, \alpha_0, N_{\text{context}}) \]
A Unified Model of Context-Sensitive Data Flow Analysis
[CSUR21]

Abstract value structure: \( \mathcal{V} = (M, v_0, \theta_n, \eta_n^{\text{Call}}, \eta_n^{\text{Ret}}, \text{Project}_Q) \)

Abstract context structure: \( \mathcal{A} = (\mathcal{A}, \alpha_0, \text{Ncontext}_n) \)
A Unified Model of Context-Sensitive Data Flow Analysis [CSUR21]

Abstract value structure: \( \mathcal{V} = (\mathbb{M}, v_0, \theta_n, \eta_n^{\text{Call}}, \eta_n^{\text{Ret}}, \text{Project}_Q) \)

Abstract context structure: \( \mathcal{A} = (\mathbb{A}, \alpha_0, \text{Ncontext}_n) \)
A Unified Model of Context-Sensitive Data Flow Analysis
[CSUR21]

Abstract value structure: \( \mathcal{V} = (\mathbb{M}, v_0, \theta_n, \eta_n^{Call}, \eta_n^{Ret}, \text{Project}_Q) \)

Abstract context structure: \( \mathcal{A} = (\mathbb{A}, \alpha_0, \text{Ncontext}_n) \)
A Unified Model of Context-Sensitive Data Flow Analysis [CSUR21]

Abstract value structure: $\mathcal{V} = (\mathbb{M}, v_0, \theta_n, \eta_n^{\text{Call}}, \eta_n^{\text{Ret}}, \text{Project}_Q)$

Abstract context structure: $\mathcal{A} = (\mathbb{A}, \alpha_0, \text{Ncontext}_n)$
Abstract value structure: \( \mathcal{V} = (\mathbb{M}, v_0, \theta_n, \eta_n^{\text{Call}}, \eta_n^{\text{Ret}}, \text{Project}_Q) \)

- Set of abstract values \( \mathbb{M} \)
  - May be different from the underlying data flow values in \( \mathbb{L} \)
  - Function \( \text{Project}_{Q \in \text{Proc}} : \mathbb{M} \rightarrow \mathbb{L} \) gives values in \( \mathbb{L} \) from those in \( \mathbb{M} \)

- Initial abstract value \( v_0 \in \mathbb{M} \)
  - (holds at \( n: \text{Start}_{\text{main}} \))

Abstract context structure: \( \mathcal{A} = (\mathbb{A}, \alpha_0, \text{Ncontext}_n) \)
A Unified Model of Context-Sensitive Data Flow Analysis [CSUR21]

Abstract value structure: \( \mathcal{V} = (\mathbb{M}, v_0, \vartheta_n, \eta_n^{\text{Call}}, \eta_n^{\text{Ret}}, \text{Project}_Q) \)

- Set of abstract values \( \mathbb{M} \)
  May be different from the underlying data flow values in \( \mathbb{L} \)
  Function \( \text{Project}_{Q_{\text{proc}}} : \mathbb{M} \rightarrow \mathbb{L} \)
gives values in \( \mathbb{L} \) from those in \( \mathbb{M} \)

- Initial abstract value \( v_0 \in \mathbb{M} \)
  (holds at \( n: \text{Start}_{\text{main}} \))

Abstract context structure: \( \mathcal{A} = (\mathbb{A}, \alpha_0, \text{Ncontext}_n) \)
A Unified Model of Context-Sensitive Data Flow Analysis
[CSUR21]

Abstract value structure: \( \mathcal{V} = (\mathbb{M}, v_0, \theta_n, \eta_n^{\text{Call}}, \eta_n^{\text{Ret}}, \text{Project}_Q) \)

Abstract context structure: \( \mathcal{A} = (\mathbb{A}, \alpha_0, \text{Ncontext}_n) \)
A Unified Model of Context-Sensitive Data Flow Analysis
[CSUR21]

Abstract value structure: $\mathcal{V} = (\mathbb{M}, v_0, \theta_n, \eta_n^{\text{Call}}, \eta_n^{\text{Ret}}, \text{Project}_Q)$

Abstract context structure: $\mathcal{A} = (\mathbb{A}, \alpha_0, \text{Ncontext}_n)$

Context transition function associated with call node $n$
$\text{Ncontext}_{n \in \mathbb{N}} : \mathbb{A} \rightarrow \mathbb{A}$
A Unified Model of Context-Sensitive Data Flow Analysis [CSUR21]

Abstract value structure: \( \mathcal{V} = (\mathbb{M}, v_0, \theta_n, \eta_n^{\text{Call}}, \eta_n^{\text{Ret}}, \text{Project}_Q) \)

Abstract context structure: \( \mathcal{A} = (\mathbb{A}, \alpha_0, \text{Ncontext}_n) \)
Abstract value structure: $V = (M, v_0, \theta_n, \eta_n^{Call}, \eta_n^{Ret}, \text{Project}_Q)$

Abstract context structure: $A = (\mathbb{A}, \alpha_0, \text{Ncontext}_n)$

Interprocedural abstract flow function
$\eta_n^{Call} : M \rightarrow M$
(defined only for call nodes)
Abstract value structure: \( \mathcal{V} = (\mathbb{M}, v_0, \theta_n, \eta_n^{\text{Call}}, \eta_n^{\text{Ret}}, \text{Project}_Q) \)

Abstract context structure: \( \mathcal{A} = (\mathbb{A}, \alpha_0, \text{Ncontext}_n) \)
A Unified Model of Context-Sensitive Data Flow Analysis [CSUR21]

Abstract value structure: \( \mathcal{V} = (\mathbb{M}, v_0, \theta_n, \eta_n^{\text{Call}}, \eta_n^{\text{Ret}}, \text{Project}_Q) \)

Abstract context structure: \( \mathcal{A} = (\mathbb{A}, \alpha_0, \text{Ncontext}_n) \)

Intraprocedural abstract flow function
\( \theta_{n \in \mathbb{N}} : \mathbb{M} \rightarrow \mathbb{M} \)
A Unified Model of Context-Sensitive Data Flow Analysis
[CSUR21]

Abstract value structure: \( \mathcal{V} = (\mathbb{M}, v_0, \theta_n, \eta_n^{Call}, \eta_n^{Ret}, \text{Project}_Q) \)

Abstract context structure: \( \mathcal{A} = (A, \alpha_0, N_{\text{context}}) \)
A Unified Model of Context-Sensitive Data Flow Analysis
[CSUR21]

Abstract value structure: \( \mathcal{V} = (\mathbb{M}, v_0, \theta_n, \eta^n_{\text{Call}}, \eta^n_{\text{Ret}}, \text{Project}_Q) \)

\[
\begin{align*}
\mathcal{A} = & (\mathbb{A}, \alpha_0, \text{Ncontext}_n) \\
(\alpha_0, v_0) \quad \text{S}_{\text{main}} \\
(\alpha_0, v_0) \quad \text{C}_i \\
(\alpha_0, v_4) \quad \text{R}_i \\
\text{E}_{\text{main}} \\

\text{Interprocedural abstract flow function} \\
\eta^n_{\text{Ret}} : \mathbb{M} \times \mathbb{M} \rightarrow \mathbb{M} \\
\text{(defined only for return nodes)}
\end{align*}
\]

\( v_4 = \eta^n_{\text{Ret}}(v_3, v_0) \)
A Unified Model of Context-Sensitive Data Flow Analysis
[CSUR21]

Abstract value structure: \( \mathcal{V} = (\mathbb{M}, v_0, \theta_n, \eta_n^{\text{Call}}, \eta_n^{\text{Ret}}, \text{Project}_Q) \)

Abstract context structure: \( \mathcal{A} = (\mathbb{A}, \alpha_0, \text{Ncontext}_n) \)

- Abstract values
- Projection function for underlying data flow values
- Abstract flow functions
- Abstract contexts
- Context transition function
A Unified Model of Context-Sensitive Data Flow Analysis
[CSUR21]

Abstract value structure: \( \mathcal{V} = (\mathbb{M}, v_0, \theta_n, \eta_{\text{Call}}^n, \eta_{\text{Ret}}^n, \text{Project}_Q) \)

Abstract context structure: \( \mathcal{A} = (\mathbb{A}, \alpha_0, N_{\text{context}}) \)
A Unified Model of Context-Sensitive Data Flow Analysis [CSUR21]

Abstract value structure: \( \mathcal{V} = (\overline{M}, v_0, \theta_n, \eta_n^{\text{Call}}, \eta_n^{\text{Ret}}, \text{Project}_Q) \)

Abstract context structure: \( \mathcal{A} = (\overline{A}, \alpha_0, N_{\text{context}}) \)
A Unified Model of Context-Sensitive Data Flow Analysis
[CSUR21]

Abstract value structure: \( \mathcal{V} = (\mathbb{M}, v_0, \theta_n, \eta_n^{Call}, \eta_n^{Ret}, \text{Project}_Q) \)

Abstract context structure: \( \mathcal{A} = (\mathbb{A}, \alpha_0, \text{Ncontext}_n) \)
A Unified Model of Context-Sensitive Data Flow Analysis [CSUR21]

Abstract value structure: \( \mathcal{V} = (\mathbb{M}, v_0, \theta_n, \eta_n^{Call}, \eta_n^{Ret}, \text{Project}_Q) \)

Abstract context structure: \( \mathcal{A} = (\mathbb{A}, \alpha_0, \text{Ncontext}_n) \)
A Unified Model of Context-Sensitive Data Flow Analysis [CSUR21]

Abstract value structure: $\mathcal{V} = (\mathcal{M}, v_0, \theta_n, \eta_n^{Call}, \eta_n^{Ret}, \text{Project}_Q)$

Abstract context structure: $\mathcal{A} = (A, \alpha_0, N\text{context}_n)$

- Abstract values
- Projection function for underlying data flow values
- Abstract flow functions
- Abstract contexts
- Context transition function
Abstract value structure: \( \mathcal{V} = (M, v_0, \theta_n, \eta_n^{Call}, \eta_n^{Ret}, \text{Project}_Q) \)

Abstract context structure: \( \mathcal{A} = (\mathbb{A}, \alpha_0, N_{\text{context}_n}) \)
A Unified Model of Context-Sensitive Data Flow Analysis
[CSUR21]

Abstract value structure: $\mathcal{V} = (\mathcal{M}, v_0, \theta_n, \eta_n^{\text{Call}}, \eta_n^{\text{Ret}}, \text{Project}_Q)$

- This model can be instantiated to a large class of methods by defining $\mathcal{V}$ and $\mathcal{A}$ for a method
- Soundness and precision of the given method can be argued using the model

Abstract context structure: $\mathcal{A} = (\mathcal{A}, \alpha_0, \text{Ncontext}_n)$
## Abstract Context Structure for Context-Sensitive Methods

[CSUR21]

<table>
<thead>
<tr>
<th>Method</th>
<th>$A$</th>
<th>$\alpha_0$</th>
<th>$N_{context}(\alpha)$</th>
</tr>
</thead>
</table>

**Talk Title:** 
PSPA Research
## Abstract Context Structure for Context-Sensitive Methods

[CSUR21]

<table>
<thead>
<tr>
<th>Method</th>
<th>$\mathcal{A}$</th>
<th>$\alpha_0$</th>
<th>$N_{\text{context}}(\alpha)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full call strings</td>
<td>$\sum$</td>
<td>$\epsilon$</td>
<td>$\alpha \cdot n$</td>
</tr>
<tr>
<td>VBTCS</td>
<td>$\sum$</td>
<td>$\epsilon$</td>
<td>$R(\alpha \cdot n, \ln A_n(\alpha))$</td>
</tr>
<tr>
<td>VASCO</td>
<td>$\llbracket$</td>
<td>$B\rrbracket$</td>
<td>$\ln A_n(\alpha)$</td>
</tr>
<tr>
<td>Restricted contexts</td>
<td>${\epsilon}$</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
</tr>
<tr>
<td>IFDS method</td>
<td>$\llbracket$</td>
<td>$B\rrbracket$</td>
<td>$\ln A_n(\alpha)$</td>
</tr>
<tr>
<td>IDE method</td>
<td>${\epsilon}$</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
</tr>
<tr>
<td>Functional method</td>
<td>${\epsilon}$</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
</tr>
<tr>
<td>$k$-limited call strings</td>
<td>$\sum_k$</td>
<td>$\epsilon$</td>
<td>$\text{suffix}_k(\alpha \cdot n)$</td>
</tr>
<tr>
<td>Context-insensitive</td>
<td>${\epsilon}$</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
</tr>
</tbody>
</table>
Abstract Value Structure for Context-Sensitive Methods
[CSUR21]

<table>
<thead>
<tr>
<th>Method</th>
<th>M</th>
<th>$v_0$</th>
<th>$\theta_n(v)$</th>
<th>$\eta^\text{Call}_n(v)$</th>
<th>$\eta^\text{Ret}_n(v, w)$</th>
<th>Project$_Q(v)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full call strings</td>
<td>\mathbb{L}</td>
<td>$Bl$</td>
<td>$f_n(v)$</td>
<td>$v$</td>
<td>$v$</td>
<td>$v$</td>
</tr>
<tr>
<td>VBTCS</td>
<td>\mathbb{L}</td>
<td>$Bl$</td>
<td>$f_n(v)$</td>
<td>$v$</td>
<td>$v$</td>
<td>$v$</td>
</tr>
<tr>
<td>VASCO</td>
<td>\mathbb{L}</td>
<td>$Bl$</td>
<td>$f_n(v)$</td>
<td>$v$</td>
<td>$v$</td>
<td>$v$</td>
</tr>
<tr>
<td>IFDS method</td>
<td>\mathbb{L}</td>
<td>$Bl$</td>
<td>$f_n(v)$</td>
<td>$v$</td>
<td>$v$</td>
<td>$v$</td>
</tr>
<tr>
<td>IDE method</td>
<td>\mathbb{L} $\rightarrow$ \mathbb{L}</td>
<td>$id$</td>
<td>$f_n \circ v$</td>
<td>$id$</td>
<td>$v \circ w$</td>
<td>$v (Bl_Q)$</td>
</tr>
<tr>
<td>Functional method</td>
<td>\mathbb{L} $\rightarrow$ \mathbb{L}</td>
<td>$id$</td>
<td>$f_n \circ v$</td>
<td>$id$</td>
<td>$v \circ w$</td>
<td>$v (Bl_Q)$</td>
</tr>
<tr>
<td>$k$-limited call strings</td>
<td>\mathbb{L}</td>
<td>$Bl$</td>
<td>$f_n(v)$</td>
<td>$v$</td>
<td>$v$</td>
<td>$v$</td>
</tr>
<tr>
<td>Context-insensitive</td>
<td>\mathbb{L}</td>
<td>$Bl$</td>
<td>$f_n(v)$</td>
<td>$v$</td>
<td>$v$</td>
<td>$v$</td>
</tr>
</tbody>
</table>
Research Explorations in Interprocedural Analysis

• Broad categories of interprocedural analysis
• Scaling top-down analysis using value contexts and bypassing
• A unified model of context-sensitive methods
• Improving bottom-up analysis by eliminating control flow
• Precise virtual call resolution with demand-driven analysis
• Improving call graphs using callee contexts
Our Approach [SAS16, TOPLAS20]

Improve the scalability of *exhaustive* pointer analysis without losing precision

- Construct sound and precise but compact statement level summaries
- Combine them naively and optimize for scalability without compromising soundness or precision

![Graph showing the trade-off between precision and scalability with Our Approach highlighting no compromise in precision](image-url)
A flow-sensitive analysis requires control flow to be recorded between memory updates that share data dependence.

**Data dependence exists ⇒**
Can be eliminated and the
Control flow between the updates becomes redundant

1. \( x = &a; \)
2. \( y = x; \)

\[ x = &a \| y = &a \]

**Data dependence does not exist ⇒**
Redundant memory updates can be eliminated
Control flow between the updates is redundant

1. \( x = &a; \)
2. \( y = &b; \)
3. \( x = &b; \)

\[ y = &b \| x = &b \]

**Data dependence is unknown ⇒**
More information is required (available in callers)
Control flow between the updates is required

1. \( y = &b; \)
2. \( *x = &a; \)
3. \( z = y; \)
Generalized Points-to Updates (GPUs) [SAS16, TOPLAS20]

- The direction in a GPU is to distinguish between what is being defined to what is being read
- For pointer analysis, case $i = 0$ does not exist
- Classical points-to update is a special case of generalized points-to update with $i = 1$ and $j = 0
All variables are global

Red nodes are known named locations
Classical Points-to Updates: 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 Updates: 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

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

Information from callers
Classical Points-to Updates: 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 Updates: 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 Updates: 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

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

Information from callers
Classical Points-to Updates: 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 Updates: 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 Updates: A High Level Abstraction of Memory for Points-to Analysis

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

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

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 Updates: A High Level Abstraction of Memory for Points-to Analysis

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 Updates: A High Level Abstraction of Memory for Points-to Analysis

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 Updates: A High Level Abstraction of Memory for Points-to Analysis

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 Updates: A High Level Abstraction of Memory for Points-to Analysis

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 Updates: A High Level Abstraction of Memory for Points-to Analysis

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
}
```
Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis

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
}
```
Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis

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

This abstraction does not introduce any imprecision over the classical points-to graph
How GPGs Handle the 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

- Scale to 158 kLoC when implemented using LTO framework in GCC 4.7.2
GPGs Across Optimizations [SAS16, TOPLAS20]

<table>
<thead>
<tr>
<th>CFG of proc f</th>
<th>Initial GPG of proc f</th>
<th>After Call Inlining</th>
<th>After Strength Reduction</th>
<th>After Dead GPU Elimination</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = &amp; a;$</td>
<td>$x \xrightarrow{1</td>
<td>0} a$</td>
<td>$x \xrightarrow{1</td>
<td>0} a$</td>
</tr>
<tr>
<td>$g();$</td>
<td>$g();$</td>
<td>$y \xrightarrow{1</td>
<td>1} x$</td>
<td>$y \xrightarrow{1</td>
</tr>
<tr>
<td>$x = &amp; b;$</td>
<td>$x \xrightarrow{1</td>
<td>0} b$</td>
<td>$x \xrightarrow{1</td>
<td>0} b$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CFG of proc g</th>
<th>After Empty GPB Elimination</th>
<th>After Coalescing</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y = x;$</td>
<td>$y \xrightarrow{1</td>
<td>0} a$</td>
</tr>
<tr>
<td></td>
<td>$x \xrightarrow{1</td>
<td>0} b$</td>
</tr>
</tbody>
</table>
GPGs Across Optimizations [SAS16, TOPLAS20]

All of the above GPGs represent sound, precise, and semantically equivalent summaries of procedure f for points-to analysis.
• Broad categories of interprocedural analysis
• Scaling top-down analysis using value contexts and bypassing
• A unified model of context-sensitive methods
• Improving bottom-up analysis by eliminating control flow
• Precise virtual call resolution with demand-driven analysis
• Improving call graphs using callee contexts
Precise Virtual Call Resolution With Demand-Driven Analysis [SCP20]

- Virtual calls are made through pointers to objects
- The resolution of virtual calls needs the class of the objects rather than the objects thus a full blown points-to analysis is an overkill hampering scalability. We need a points-to-class analysis that uses type based abstraction
- Any data abstraction (such as type based abstraction) conflates many objects together and introduces imprecision
- We show how this imprecision can be mitigated in a demand-driven method
Exhaustive Analysis and Data Abstraction

\[ p = \&z \]

\[ y = \text{new} A \]

\[ *p = \text{new} A \]

\[ z \rightarrow f = \text{new} B \]

\[ y \rightarrow f = \text{new} C \]

\[ z \rightarrow f \rightarrow \text{vfun()} \]
Exhaustive Analysis and Data Abstraction

\[ p = \&z \]

\[ y = \text{new}A \]

\[ *p = \text{new}A \]

\[ z \rightarrow f = \text{new}B \]

\[ y \rightarrow f = \text{new}C \]

\[ z \rightarrow f \rightarrow \text{vfun()} \]
Exhaustive Analysis and Data Abstraction

\[ p = \&z \]

\[ y = \text{new} A \]

\[ \ast p = \text{new} A \]

\[ z \rightarrow f = \text{new} B \]

\[ y \rightarrow f = \text{new} C \]

\[ z \rightarrow f \rightarrow \text{vfun()} \]
Exhaustive Analysis and Data Abstraction

\[ p = \& z \]

\[ y = \text{new}A \]

\[ *p = \text{new}A \]

\[ z \rightarrow f = \text{new}B \]

\[ y \rightarrow f = \text{new}C \]

\[ z \rightarrow f \rightarrow \text{vfun()} \]
Exhaustive Analysis and Data Abstraction

\[ p = &z \]
\[ y = \text{new}A \]
\[ *p = \text{new}A \]
\[ z \to f = \text{new}B \]
\[ y \to f = \text{new}C \]
\[ z \to f \to \text{vfun}() \]
Exhaustive Analysis and Data Abstraction

\[ p = \& z \]

\[ y = \text{newA} \]

\[ *p = \text{newA} \]

\[ z \rightarrow f = \text{newB} \]

\[ y \rightarrow f = \text{newC} \]

\[ z \rightarrow f \rightarrow \text{vfun()} \]
Exhaustive Analysis and Data Abstraction

\[ p = \&z \]
\[ y = \text{new} A \]
\[ *p = \text{new} A \]
\[ z \rightarrow f = \text{new} B \]
\[ y \rightarrow f = \text{new} C \]
\[ z \rightarrow f \rightarrow \text{vfun()} \]
Exhaustive Analysis and Data Abstraction

- Type abstraction introduces spurious alias \((y, z)\)
- We spuriously conclude that vfun could be called for class C too
Need For Speculation in Demand-Driven Method

\[ p = \& z \]

\[ *p = \text{new}A \]

\[ z \rightarrow \text{vfun()} \]
Need For Speculation in Demand-Driven Method

\[ p = \& z \]
\[ *p = \text{new} A \]
\[ z \rightarrow \text{vfun()} \]
Need For Speculation in Demand-Driven Method

\[
p = \&z \\
*p = \text{new}A \\
z \rightarrow \text{vfun()}\]
Need For Speculation in Demand-Driven Method

\[ p = \&z \]
\[ *p = \text{new} A \]
\[ z \rightarrow \text{vfun()} \]
Need For Speculation in Demand-Driven Method

\[ p = \&z \]

\[ *p = \text{new}A \]

\[ z \rightarrow \text{vfun()} \]
To ensure soundness, we need to speculate demands at indirect assignment statements.
Need For Speculation in Demand-Driven Method

\[ p = \& z \]

Is \( *p \) aliased to \( z \)?

\[ *p = \text{new}A \]

\[ z \rightarrow \text{vfun()} \]

Alias speculation

To ensure soundness, we need to speculate demands at indirect assignment statements.
Need For Speculation in Demand-Driven Method

To ensure soundness, we need to speculate demands at indirect assignment statements.

 Alias speculation

\[
\begin{align*}
p &= &\& z \\
p \rightarrow z & \quad p \\
*p &= &\text{new} A \\
? & \quad z \\
z \rightarrow \text{vfun}() 
\end{align*}
\]
Need For Speculation in Demand-Driven Method

To ensure soundness, we need to speculate demands at indirect assignment statements.
Need For Speculation in Demand-Driven Method

To ensure soundness, we need to speculate demands at indirect assignment statements.

Alias speculation  
Pointer speculation

Is \( *p \) aliased to \( z \)?

\[ p = &z \]

\[ \text{Is } *p \text{ aliased to } z? \]

\[ *p = \text{new } A \]

\[ z \rightarrow A \]

\[ z \rightarrow \text{vfun()} \]
Need For Speculation in Demand-Driven Method

To ensure soundness, we need to speculate demands at indirect assignment statements

Alias pair \((p, &z)\) can be found by raising demand for \&z

Alias speculation

Pointer speculation
Need For Speculation in Demand-Driven Method

![Diagram]

To ensure soundness, we need to speculate demands at indirect assignment statements.
Need For Speculation in Demand-Driven Method

To ensure soundness, we need to speculate demands at indirect assignment statements.

Alias speculation

Pointer speculation

Is \( \ast p \) aliased to \( z \)?

\[
p = \& z \\
p \rightarrow z \quad p \quad z, \& z \quad p \rightarrow z
\]

\[
\ast p = \text{new} A \\
\ast p \rightarrow z \quad p \rightarrow z
\]

\[
z \rightarrow A \quad z \quad z, \& z \quad z \rightarrow A
\]

Alias pair \((p, \&z)\) can be found by raising demand for \( \&z \)

\[
z \rightarrow \text{vfun}() \\
z \rightarrow A
\]
Need For Speculation in Demand-Driven Method

\[ p = &z \]

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFSA</td>
<td>Exhaustive flow-sensitive points-to analysis</td>
</tr>
<tr>
<td>DFSA</td>
<td>Demand-driven flow-sensitive points-to analysis</td>
</tr>
<tr>
<td>ADFSA</td>
<td>Demand-driven flow-sensitive points-to analysis using alias speculation</td>
</tr>
<tr>
<td>PDFSA</td>
<td>Demand-driven flow-sensitive points-to analysis using pointer speculation</td>
</tr>
</tbody>
</table>

Alias speculation  Pointer speculation

To ensure soundness, we need to speculate demands at indirect assignment statements.
ADFSA and PDFSA [SCP20]

\[
p = &z \\
y = \text{new} A \\
*p = \text{new} A \\
z \rightarrow f = \text{new} B \\
y \rightarrow f = \text{new} C \\
z \rightarrow f \rightarrow \text{vfun()}\
\]
ADFSA and PDFSA [SCP20]

Alias speculation (ADFSA)

\[ p = \& z \]
\[ y = \text{new} A \]
\[ \ast p = \text{new} A \]
\[ z \rightarrow f = \text{new} B \]
\[ y \rightarrow f = \text{new} C \]
\[ z, z \rightarrow f \]
\[ z \rightarrow f \rightarrow \text{vfun()} \]
ADFSA and PDFSA [SCP20]

**Alias speculation (ADFSA)**

```
p = &z
```

```
y = new A
```

```
*p = new A
```

```
z → f = new B
```

```
z, y
```

```
y → f = new C
```

```
z, z → f
```

```
z → f → vfun()
```

**Demand Propagation**
**ADFSA and PDFSA** [SCP20]

**Alias speculation (ADFSA)**

```
p = &z
```

```
y = newA
```

```
*p = newA
```

```
z, y
```

```
z \rightarrow f = newB
```

```
z, y
```

```
y \rightarrow f = newC
```

```
z, z \rightarrow f
```

```
z \rightarrow f \rightarrow vfun()
```

**Demand Propagation**
ADFSA and PDFSA [SCP20]

**Demand Propagation**

```
p = &z

y = new A

*p = new A

z, y

z → f = new B

z, y

y → f = new C

z, z → f

z → f → vfun()
```

**Alias speculation (ADFSA)**

```
p = &z

y = new A

*p = new A

z, y

z → f = new B

z, y

y → f = new C

z, z → f

z → f → vfun()
```
ADFSA and PDFSA [SCP20]

Alias speculation (ADFSA)

\[
p = \&\!z
\]

\[
p
\]

\[
y = \text{new} A
\]

\[
p
\]

\[
*p = \text{new} A
\]

\[
z, y
\]

\[
z \rightarrow f = \text{new} B
\]

\[
z, y
\]

\[
y \rightarrow f = \text{new} C
\]

\[
z, z \rightarrow f
\]

\[
z \rightarrow f \rightarrow \text{vfun}(\)

Demand Propagation
ADFSA and PDFSA [SCP20]

Points-to Propagation

Alias speculation (ADFSA)

\[ p = \& z \]

\[ p \rightarrow z \]

\[ p \]

\[ y = \text{new} A \]

\[ \]

\[ *p = \text{new} A \]

\[ z, y \]

\[ z \rightarrow f = \text{new} B \]

\[ z, y \]

\[ y \rightarrow f = \text{new} C \]

\[ z, z \rightarrow f \]

\[ z \rightarrow f \rightarrow \text{vfun()} \]
ADFSA and PDFSA [SCP20]

Points-to Propagation

```
p = &z
  →
p
  →
  →
y = new A
  →
y
  →
  →
*p = new A
  →
z, y
  →
z → f = new B
  →
z, y
  →
y → f = new C
  →
z, z → f
  →
z → f → vfun()
```

Alias speculation (ADFSA)
ADFSA and PDFSA [SCP20]

**Alias speculation (ADFSA)**

```
p = &z
```

```
p → z
```

```
p, y
```

```
y = new A
```

```
*p = new A
```

```
z, y
```

```
z → f = new B
```

```
z, y
```

```
y → f = new C
```

```
z, z → f
```

```
z → f → vfun()
```
**ADFSA and PDFSA [SCP20]**

### Points-to Propagation

1. **p → z**
   - `p = &z`

2. **p → z, y → A**
   - `y = newA`
   - `*p = newA`
   - `z, y`
   - `z → f = newB`
   - `y → f = newC`
   - `z, z → f`
   - `z → f → vfun()`

### Alias speculation (ADFSA)

- `z, z → f`
- `z → f = newC`
**ADFSA and PDFSA [SCP20]**

**Alias speculation (ADFSA)**

- $p \rightarrow z$
- $p$
- $y = \text{new } A$
- $p, y$
- $*p = \text{new } A$
- $z \rightarrow f = \text{new } B$
- $z, y$
- $y \rightarrow f = \text{new } C$
- $z, z \rightarrow f$
- $z \rightarrow f \rightarrow \text{vfun()}$

**Points-to Propagation**

- $p \rightarrow z \rightarrow A, y \rightarrow A$
- $z, y$
- $z \rightarrow f = \text{new } B$
- $z, y$
- $y \rightarrow f = \text{new } C$
- $z, z \rightarrow f$
- $z \rightarrow f \rightarrow \text{vfun()}$
**ADFSA and PDFSA [SCP20]**

**Alias speculation (ADFSA)**

- $p = \&z$
- $p \rightarrow z$
- $y = \text{new}A$
- $p, y \rightarrow p$
- $\ast p = \text{new}A$
- $z \rightarrow f = \text{new}B$
- $z, y \rightarrow z$
- $y \rightarrow f = \text{new}C$
- $z, z \rightarrow f$
- $z \rightarrow f \rightarrow \text{vfun()}$

**Points-to Propagation**

- $p \rightarrow z \rightarrow A, y \rightarrow A$
- $z, y \rightarrow p \rightarrow z \rightarrow A$
- $y \rightarrow f \rightarrow B, y \rightarrow A$
- $z, y \rightarrow f$
**ADFSA and PDFSA [SCP20]**

**Points-to Propagation**

1. \( p \rightarrow z \rightarrow A, y \rightarrow A \)
2. \( p \rightarrow z \rightarrow A \stackrel{f}{\rightarrow} B, y \rightarrow A \)
3. \( p \rightarrow z \rightarrow A \stackrel{f}{\rightarrow} B, y \rightarrow A \stackrel{f}{\rightarrow} C \)

**Alias speculation (ADFSA)**

1. \( p = \& z \)
2. \( y = \text{new} A \)
3. \( *p = \text{new} A \)
4. \( z \rightarrow f = \text{new} B \)
5. \( y \rightarrow f = \text{new} C \)
6. \( z \rightarrow f \rightarrow \text{vfun}() \)
ADFSA and PDFSA [SCP20]

Alias speculation (ADFSA)

\[ p \rightarrow z \]

\[ p \]

\[ y = \text{new} A \]

\[ p, y \]

\[ *p = \text{new} A \]

\[ z, y \]

\[ z \rightarrow f = \text{new} B \]

\[ z, y \]

\[ y \rightarrow f = \text{new} C \]

\[ z, z \rightarrow f \]

\[ z \rightarrow f \rightarrow \text{vfun()} \]

\[ p = \& z \]
Alias speculation (ADFSA)

\[ p = \& z \]

\[ p \rightarrow z \]

\[ y = \text{new} A \]

\[ p, y \]

In general, demand and exhaustive methods achieve *identical* precision

\[ z \rightarrow f = \text{new} B \]

\[ z, y \]

\[ z, z \rightarrow f \]

\[ z \rightarrow f \rightarrow \text{vfun()} \]
ADFSA and PDFSA [SCP20]

Alias speculation (ADFSA)

\[ p \rightarrow z \]
\[ y \rightarrow A \]
\[ z \rightarrow f \rightarrow vfun() \]

Pointer speculation (PDFSA)

\[ p = &z \]
\[ y = \text{new} A \]
\[ *p = \text{new} A \]
\[ z \rightarrow f = \text{new} B \]
\[ y \rightarrow f = \text{new} C \]
\[ z, &z \]

Demand Propagation
**ADFSA and PDFSA [SCP20]**

**Alias speculation (ADFSA)**

```
p \rightarrow z
```

```
\text{p = \&z}
```

```
p
```

```
y = \text{newA}
```

```
p, y
```

```
p \rightarrow z, y \rightarrow A
```

```
z, y
```

```
*\text{p = newA}
```

```
p \rightarrow z \rightarrow A, y \rightarrow A
```

```
z \rightarrow f = \text{newB}
```

```
z, y
```

```
z, \&z
```

```
p \rightarrow z \rightarrow A \rightarrow B, y \rightarrow A
```

```
z, y
```

```
z, \&z
```

```
y \rightarrow f = \text{newC}
```

```
z, z \rightarrow f
```

```
z, \&z
```

```
z \rightarrow f \rightarrow \text{vfun()}
```

**Pointer speculation (PDFSA)**

```
p \rightarrow z
```

```
f \rightarrow B
```

```
p \rightarrow z
```

```
f \rightarrow C
```

```
y \rightarrow A
```

**Demand Propagation**
**ADFSA and PDFSA [SCP20]**

**Alias speculation (ADFSA)**

- \( p \rightarrow z \)
- \( p \rightarrow z, y \rightarrow A \)
- \( p \rightarrow z \rightarrow A, y \rightarrow A \)
- \( p \rightarrow z \rightarrow A \rightarrow B, y \rightarrow A \)
- \( p \rightarrow z \rightarrow A \rightarrow B, y \rightarrow A \rightarrow C \)

**Pointer speculation (PDFSA)**

- \( p = &z \)
- \( y = \text{new} A \)
- \( *p = \text{new} A \)
- \( z \rightarrow f = \text{new} B \)
- \( y \rightarrow f = \text{new} C \)
- \( z \rightarrow f \rightarrow \text{vfun()} \)

**Demand Propagation**
**Talk Title:** PSPA Research

**Topic:** Some Meanderings

**Intraprocedural Analysis**

**Interprocedural Analysis**

**Conclusions**

**References**

---

**ADFSA and PDFSA [SCP20]**

**Alias speculation (ADFSA)**

- $p \rightarrow z$
  - $p$
  - $y = \text{new}A$
  - $p, y$
  - $z, &z$
  - $\ast p = \text{new}A$
  - $z, y$
  - $z, &z$
  - $z \rightarrow f = \text{new}B$
  - $z, y$
  - $z, &z$
  - $y \rightarrow f = \text{new}C$
  - $z, z \rightarrow f$
  - $z, &z$
  - $z \rightarrow f \rightarrow \text{vfun}()$

**Pointer speculation (PDFSA)**

- $p = \& z$
  - $p$

---

**Demand Propagation**
**ADFSA and PDFSA [SCP20]**

**Alias speculation (ADFSA)**

- $p \rightarrow z$
- $p \rightarrow z, y \rightarrow A$
- $p \rightarrow z \rightarrow A, y \rightarrow A$
- $p \rightarrow z \rightarrow A \rightarrow B, y \rightarrow A$
- $p \rightarrow z \rightarrow A \rightarrow B, y \rightarrow A \rightarrow C$

**Pointer speculation (PDFSA)**

- $p = \& z$
- $p, y \rightarrow z, \& z$
- $*p = \text{new}A$
- $z, y \rightarrow z, \& z$
- $z \rightarrow f = \text{new}B$
- $z, y \rightarrow z, \& z$
- $y \rightarrow f = \text{new}C$
- $z, z \rightarrow f \rightarrow \text{vfun}()$

**Demand Propagation**
ADFSA and PDFSA [SCP20]

Alias speculation (ADFSA)

\[ p = \&z \]
\[ p \rightarrow z \]
\[ y = \text{new}A \]
\[ p \rightarrow z, y \rightarrow A \]
\[ \ast p = \text{new}A \]
\[ p \rightarrow z \rightarrow A, y \rightarrow A \]
\[ z \rightarrow f = \text{new}B \]
\[ p \rightarrow z \rightarrow A \rightarrow B, y \rightarrow A \]
\[ y \rightarrow f = \text{new}C \]
\[ p \rightarrow z \rightarrow A \rightarrow B, y \rightarrow A \rightarrow C \]
\[ z, z \rightarrow f \rightarrow \text{vfun()} \]

Pointer speculation (PDFSA)

\[ p \rightarrow z \]
\[ p \]
\[ z, \&z \]
\[ y = \text{new}A \]
\[ p, y \rightarrow z, \&z \]
\[ \ast p = \text{new}A \]
\[ z, y \rightarrow z, \&z \]
\[ z \rightarrow f = \text{new}B \]
\[ z, y \rightarrow z, \&z \]
\[ y \rightarrow f = \text{new}C \]
\[ z, z \rightarrow f \rightarrow \text{vfun()} \]
ADFSA and PDFSA [SCP20]

**Alias speculation (ADFSA)**
- \( p = \& z \)
- \( p \rightarrow z \)
- \( y = \text{new} A \)
- \( p \rightarrow z, y \rightarrow A \)
- \( *p = \text{new} A \)
- \( p \rightarrow z \rightarrow A, y \rightarrow A \)
- \( p \rightarrow z \rightarrow A \xrightarrow{f} B, y \rightarrow A \)
- \( p \rightarrow z \rightarrow A \xrightarrow{f} B, y \rightarrow A \xrightarrow{f} C \)

**Pointer speculation (PDFSA)**
- \( p = \& z \)
- \( z, \& z \)
- \( y = \text{new} A \)
- \( p, y \rightarrow z, \& z \)
- \( *p = \text{new} A \)
- \( z, y \rightarrow z, \& z \)
- \( z \rightarrow f = \text{new} B \)
- \( z, y \rightarrow z, \& z \)
- \( y \rightarrow f = \text{new} C \)
- \( z, z \rightarrow f \rightarrow z, \& z \)
- \( z \rightarrow f \rightarrow \text{vfun()} \)

Points-to Propagation
ADFSA and PDFSA [SCP20]

Alias speculation (ADFSA)

\[ p = \&z \]
\[ p \rightarrow z \]
\[ y = \text{new} A \]
\[ p \rightarrow z, y \rightarrow A \]
\[ *p = \text{new} A \]
\[ p \rightarrow z \rightarrow A, y \rightarrow A \]
\[ p \rightarrow z \rightarrow A \rightarrow B, y \rightarrow A \]
\[ p \rightarrow z \rightarrow A \rightarrow B, y \rightarrow A \rightarrow C \]

Pointer speculation (PDFSA)

\[ p \rightarrow z \]
\[ p \rightarrow z, \&z \]
\[ p \rightarrow z \]
\[ y \rightarrow \text{new} A \]
\[ p, y \rightarrow z, \&z \]
\[ *p = \text{new} A \]
\[ p \rightarrow z \rightarrow A, y \rightarrow A \]
\[ p \rightarrow z \rightarrow A \rightarrow B, y \rightarrow A \]
\[ p \rightarrow z \rightarrow A \rightarrow B, y \rightarrow A \rightarrow C \]

Points-to Propagation
ADFSA and PDFSA [SCP20]

Alias speculation (ADFSA)

\[ p = \& z \]
\[ p \rightarrow z \]
\[ y = \text{new} A \]
\[ p \rightarrow z, y \rightarrow A \]
\[ *p = \text{new} A \]
\[ p \rightarrow z \rightarrow A, y \rightarrow A \]
\[ z \rightarrow f = \text{new} B \]
\[ p \rightarrow z \rightarrow A \rightarrow B, y \rightarrow A \]
\[ y \rightarrow f = \text{new} C \]
\[ p \rightarrow z \rightarrow A \rightarrow B, y \rightarrow A \rightarrow C \]
\[ z \rightarrow f \rightarrow \text{vfun()} \]

Pointer speculation (PDFSA)

\[ p \rightarrow z \]
\[ p \]
\[ z, \& z \]
\[ p \rightarrow z \]
\[ p, y \]
\[ z, \& z \]
\[ p \rightarrow z \]
\[ z, y \]
\[ z, \& z \]
\[ p \rightarrow z, z \rightarrow A \]
\[ p \rightarrow z, z \rightarrow A \rightarrow f \]
\[ p \rightarrow z \rightarrow A \rightarrow f \rightarrow B \]

Points-to Propagation
ADFSA and PDFSA [SCP20]

**Alias speculation (ADFSA)**

- $p = \& z$
- $p \rightarrow z$
- $y = \text{new} A$
- $p \rightarrow z, y \rightarrow A$
- $\ast p = \text{new} A$
- $p \rightarrow z \rightarrow A, y \rightarrow A$
- $z \rightarrow f = \text{new} B$
- $p \rightarrow z \rightarrow A \rightarrow B, y \rightarrow A$
- $y \rightarrow f = \text{new} C$
- $p \rightarrow z \rightarrow A \rightarrow B, y \rightarrow A \rightarrow C$

**Pointer speculation (PDFSA)**

- $p \rightarrow z$
- $z, \& z$
- $p \rightarrow z$
- $p \rightarrow z, z \rightarrow A$
- $z, y$
- $z, \& z$
- $p \rightarrow z, z \rightarrow A$
- $z, y$
- $z, \& z$
- $p \rightarrow z, z \rightarrow A$
- $z, z \rightarrow f$
- $z, \& z$
- $p \rightarrow z, z \rightarrow A$
- $z \rightarrow f \rightarrow \text{vfun}()$
ADFSA and PDFSA [SCP20]

Alias speculation (ADFSA)

\[ p = z \]

\[ p \rightarrow z \]

\[ y = \text{new}A \]

\[ p \rightarrow z, y \rightarrow A \]

\[ \star p = \text{new}A \]

\[ p \rightarrow z \rightarrow A, y \rightarrow A \]

\[ z \rightarrow f = \text{new}B \]

\[ p \rightarrow z \rightarrow A \rightarrow B, y \rightarrow A \]

\[ y \rightarrow f = \text{new}C \]

\[ p \rightarrow z \rightarrow A \rightarrow B, y \rightarrow A \rightarrow C \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]

\[ p \rightarrow z \rightarrow A \rightarrow B \]
**ADFSA and PDFSA [SCP20]**

**Alias speculation (ADFSA)**

\[
p = \&z
\]

\[
p \rightarrow z
\]

\[
y = \text{new}A
\]

\[
p \rightarrow z, y \rightarrow A
\]

\[
p \rightarrow z \rightarrow A \rightarrow B, y \rightarrow A
\]

\[
p \rightarrow z \rightarrow A \rightarrow B, y \rightarrow A \rightarrow C
\]

**Pointer speculation (PDFSA)**

\[
p \rightarrow z
\]

\[
* p = \text{new}A
\]

**Demand-driven analysis with pointer speculation is more precise than with alias speculation in this case**

\[
p \rightarrow z \rightarrow A \rightarrow B
\]

\[
\rightarrow B
\]

\[
y \rightarrow f = \text{new}C
\]

\[
z, z \rightarrow f
\]

\[
z \rightarrow f \rightarrow \text{vfun()}
\]

\[
p \rightarrow z \rightarrow A \rightarrow B
\]
Comparing Precision of ADFSA and PDFSA [SCP20]

- ADFSA seeks aliases of the demand raised whereas PDFSA seeks aliases and pointers of the demand raised
Comparing Precision of ADFSA and PDFSA [SCP20]

• ADFSA seeks aliases of the demand raised whereas PDFSA seeks aliases and pointers of the demand raised

• Since a pointer cannot be an object whereas pointee can be, PDFSA can avoid some imprecision that ADFSA cannot
Comparing Precision of ADFSA and PDFSA [SCP20]

- ADFSA seeks aliases of the demand raised whereas PDFSA seeks aliases and pointers of the demand raised

- Since a pointer cannot be an object whereas pointee can be, PDFSA can avoid some imprecision that ADFSA cannot

- PDFSA is at least as precise as ADFSA in each case and more precise in some cases
Comparing Precision of PDFSA and EFSA [SCP20]

- **Conventional wisdom**
  - Demand-driven methods are more efficient versions of exhaustive methods because they compute only the required information
  - Precision of demand-driven method and exhaustive method is identical
Comparing Precision of PDFSA and EFSA [SCP20]

- **Conventional wisdom**
  - Demand-driven methods are more efficient versions of exhaustive methods because they compute only the required information
  - Precision of demand-driven method and exhaustive method is identical

- **Known Fact**
  Demand-driven methods compute less information
Comparing Precision of PDFSA and EFSA [SCP20]

- **Conventional wisdom**
  - Demand-driven methods are more efficient versions of exhaustive methods because they compute only the required information
  - Precision of demand-driven method and exhaustive method is identical

- **Known Fact**
  Demand-driven methods compute less information

- **Self-evident intuition**
  Imprecision caused by abstraction should increase with the amount of data abstracted
Comparing Precision of PDFSA and EFSA [SCP20]

• **Conventional wisdom**
  - Demand-driven methods are more efficient versions of exhaustive methods because they compute only the required information
  - Precision of demand-driven method and exhaustive method is identical

• **Known Fact**
  Demand-driven methods compute less information

• **Self-evident intuition**
  Imprecision caused by abstraction should increase with the amount of data abstracted

• Our work shows how a demand-driven method can be made more precise than the corresponding exhaustive method in some cases
Research Explorations in Interprocedural Analysis

• Broad categories of interprocedural analysis
• Scaling top-down analysis using value contexts and bypassing
• A unified model of context-sensitive methods
• Improving bottom-up analysis by eliminating control flow
• Precise virtual call resolution with demand-driven analysis
• Improving call graphs using callee contexts

Next Topic
Precise Construction of Call Graphs (or Constructing Callee Contexts) [WiP]

• **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? [WiP]

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

```
Start\_main

\text{Start}_P\quad x*y

\text{End}_P

\text{R}_1\quad i = i + 1

\text{End}_Q

\text{End\_main}

fp = a[i]

call * fp
```

```
\text{Start}_Q\quad x*y

\text{End}_Q
```
What Does A Callee Context Mean? [WiP]

Start_{main}

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

\[ \text{Call } p \]

Start_P \ x \ast \ y

C_1 \ call \ \ast \ fp

End_P \ x \ast \ y \ Start_Q

\[ \text{Call } p \]

End_P \ x \ast \ y \ End_Q

Start_{main}

\[ fp = a[i] \]

\[ n_1 \]

\[ n_2 \]

\[ n_3 \]

\[ i = i + 1 \]

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

\[
\begin{align*}
  a &= ["P", "Q"] \\
  fp &= a[i] \\
  call &\ast fp \\
  x &\ast y \\
  i &= i + 1 \\
  if &i < n \\
  End_{main} &
\end{align*}
\]

Invalid execution path!

Is \(x \ast y\) available?
What Does A Callee Context Mean? [WiP]

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

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

$\text{Start}_P$

$\text{Call } *fp$

End$_P$

R$_1$

$\text{End}_Q$

Valid execution path!

Is $x \ast y$ available?

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

$x \ast y$

$fp = a[i]$

$n_1$

$n_2$ $i = i + 1$

$n_3$ if $i < n$

Main

P

Q

Is $x \ast y$ available?
What Does A Callee Context Mean? [WiP]

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

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

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

• **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) [WiP]

• **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) [WiP]

- **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) [WiP]

- **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
Conclusions
Observations

- Relevant pointer information in a program is very small and sparse
- 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!*

- Precision of analysis can be improved by
  - Excluding infeasible control flow paths
  - Interleaving program analyses
Observations

- 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.
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.

- We achieve this without compromising soundness or precision.
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

Original Control Flow

Over-approximated Control Flow

May under-approximate data dependences

May over-approximate data dependences
The Next Holy Grail in Search of Scalability?

- Under-approximated Control Flow
  - May under-approximate data dependences
  - Unsound

- Original Control Flow
  - Sound and Precise

- Over-approximated Control Flow
  - May over-approximate data dependences
  - Sound and Imprecise
The Next Holy Grail in Search of Scalability?

Under-approximated Control Flow

Original Control Flow

Over-approximated Control Flow

Precision

Scalability

High

Low

High

Low

Precision

Scalability

High

Low
The Next Holy Grail in Search of Scalability?

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

**Precision**

**Scalability**

- High
- Low

Precision vs. Scalability

- High Precision, Low Scalability
- Low Precision, High Scalability

- Under-approximated Control Flow
- Original Control Flow
- Over-approximated Control Flow
How to eliminate redundant control flow without under-approximating or over-approximating data dependences? SSA form achieves this only at the intraprocedural level and only for local non-address-taken variables.
How to eliminate redundant control flow without under-approximating or over-approximating data dependences? SSA form achieves this only at the intraprocedural level and only for local non-address-taken variables.
A Spectrum of Possible Ways of Performing Computation

exhaustive computation

computation restricted to usable information

avoiding redundant computation

demand driven computation

Maximum Computation

What should be computed?

Minimum Computation

Early Computation

When should it be computed?

Late Computation

exhaustive computation

computation restricted to usable information

avoiding redundant computation

demand driven 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

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

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

What should be computed?
Maximum Computation

When should it be computed?
Early 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

Uday Khedker
IIT Bombay

Talk Title:
PSPA Research

Topic:
Some Meanderings

Intraprocedural Analysis
Interprocedural Analysis
Conclusions
References
A Spectrum of Possible Ways of Performing Computation

What should be computed?

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

When should it be computed?

- Maximum Computation
- Early Computation
- Late Computation
- Minimum Computation

Do not compute what you don’t need!

Who defines what is needed?

Definition of Analysis

Maximum Computation

Minimum Computation

Early Computation

Late Computation

Uday Khedker

IIT Bombay

Talk Title:
PSPA Research

Topic:
Some Meanderings

Intraprocedural Analysis

Interprocedural Analysis

Conclusions

References
A Spectrum of Possible Ways of Performing Computation

- exhaustive computation
- computation restricted to usable information
- avoiding redundant computation
- splitting into pre and post 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? No One!

Uday Khedker
IIT Bombay

Talk Title: PSPA Research

Topic: Some Meanderings

Intraprocedural Analysis

Interprocedural Analysis

Conclusions

References
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?

These seem orthogonal and may be used together
References
References


References


Thank You!