Static Analysis of Programs:
A Heap-Centric View

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

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

5 April 2008
Part 1

Introduction
Copyright

A large part of these slides are based on the following material:


(2) is not yet published and the copyrights are reserved by the authors. The copyright for (1) is held by ACM and that for (3) is held by CRC Press USA.

These slides may be used only for academic purposes.
Outline

• Motivation: The need of heap data flow analysis
• Formulating data flow analysis
• Mathematical foundations of data flow analysis
  ▶ The set of data flow values
  ▶ The set of flow functions
  ▶ Solutions of data flow analyses
  ▶ Algorithms for performing data flow analysis
  ▶ Complexity of data flow analysis
• Data flow analysis for heap references
• Conclusions
Part 2

Motivation
Motivation

- Heap memory allocation
- Limitations of garbage collection
- Optimization to improve garbage collection
Heap allocation provides the flexibility of

- **Variable Sizes.** Data structures can grow or shrink as desired at runtime.
  
  (Not bound to the declarations in program.)

- **Variable Lifetimes.** Data structures can be created and destroyed as desired at runtime.
  
  (Not bound to the activations of procedures.)
Managing Heap Memory

Decision 1: When to Allocate?

- **Explicit.** Specified in the programs. (e.g. Imperative/OO languages)
- **Implicit.** Decided by the language processors. (e.g. Declarative Languages)
Managing Heap Memory

Decision 1: When to Allocate?

- **Explicit.** Specified in the programs. (eg. Imperative/OO languages)
- **Implicit.** Decided by the language processors. (eg. Declarative Languages)

Decision 2: When to Deallocate?

- **Explicit.** Manual Memory Management (eg. C/C++)
- **Implicit.** Automatic Memory Management aka Garbage Collection (eg. Java/Declarative languages)
State of Art in Manual Deallocation

- Memory leaks
  10% to 20% of last development effort goes in plugging leaks

- Tool assisted manual plugging

  *Purify, Electric Fence, RootCause, GlowCode, yakTest, Leak Tracer, BDW Garbage Collector, mtrace, memwatch, dmalloc etc.*

- All leak detectors
  - are dynamic (and hence specific to execution instances)
  - generate massive reports to be perused by programmers
  - usually do not locate last use but only allocation escaping a call

  ⇒ At which program point should a leak be “plugged”? 

Garbage Collection $\equiv$ Automatic Deallocation

- Retain active data structure. Deallocate inactive data structure.

- What is an Active Data Structure?
Garbage Collection \equiv Automatic Deallocation

- Retain active data structure.
  Deallocate inactive data structure.

- What is an Active Data Structure?

  \textbf{If} an object does not have an access path, (i.e. it is unreachable) \textbf{then} its memory can be reclaimed.
Garbage Collection ≡ Automatic Deallocation

- Retain active data structure. Deallocate inactive data structure.

- What is an Active Data Structure?

  If an object does not have an access path, (i.e. it is unreachable)
  then its memory can be reclaimed.

What if an object has an access path, but is not accessed after the
given program point?
What is Garbage?

1. $w = x$  // $x$ points to $m_a$
2. if ($x.data < \text{max}$)
3.     $x = x.rptr$
4. $y = x.lptr$
5. $z = \text{New class of } z$
6. $y = y.lptr$
7. $z.sum = x.data + y.data$

Garbage
What is Garbage?

1. \( w = x \) \hspace{1cm} // \text{x points to } m_a
2. if \( (x.data < \text{max}) \)
3. \( x = x.rptr \)
4. \( y = x.lptr \)
5. \( z = \text{New class of } z \)
6. \( y = y.lptr \)
7. \( z.sum = x.data + y.data \)
What is Garbage?

1. \( w = x \) // \( x \) points to \( m_a \)
2. if \((x.data < \text{max})\)
3. \( x = x.rptr \)
4. \( y = x.lptr \)
5. \( z = \text{New class of } z \)
6. \( y = y.lptr \)
7. \( z.sum = x.data + y.data \)
What is Garbage?

All white nodes are unused and should be considered garbage.

1. \( w = x \)  // \( x \) points to \( m_a \)
2. if \((x.data < \text{max})\)
3. \( x = x.rptr \)
4. \( y = x.lptr \)
5. \( z = \text{New class of } z \)
6. \( y = y.lptr \)
7. \( z.sum = x.data + y.data \)
Is Reachable Same as Live?

From www.memorymanagement.org/glossary

**live** (also known as alive, active): Memory(2) or an object is live if the program will read from it in future. *The term is often used more broadly to mean reachable.*

It is not possible, in general, for garbage collectors to determine exactly which objects are still live. Instead, they use some approximation to detect objects that are provably dead, *such as those that are not reachable.*

Similar terms: reachable. Opposites: dead. See also: undead.
Is Reachable Same as Live?

- Not really. Most of us know that.

  Even with the state of art of garbage collection, 24% to 76% unused memory remains unclaimed

- Yet we have no way of distinguishing.

  Over a dozen reported approaches (since 1996), no real success.
Cedar Mesa Folk Wisdom

Make the unused memory unreachable by setting references to NULL.
(GC FAQ: http://www.iecc.com/gclist/GC-harder.html)
Cedar Mesa Folk Wisdom

Make the unused memory unreachable by setting references to NULL. (GC FAQ: http://www.iecc.com/gclist/GC-harder.html)

Stack

Heap

Uday Khedker
IIT Bombay
Cedar Mesa Folk Wisdom

- Most promising, simplest to understand, yet the hardest to implement.

- Which references should be set to NULL?
  - Most approaches rely on feedback from profiling.
  - No systematic and clean solution.
Distinguishing Between Reachable and Live

The state of art

- Eliminating objects reachable from root variables which are not live.
- Implemented in current Sun JVMs.
- Uses liveness data flow analysis of root variables (stack data).
- What about liveness of heap data?
Liveness of Stack Data

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

If changed to `while`
Liveness of Stack Data

```
while (x.data < max)
  x = x.rptr
  y = x.lptr
  z = New class_of_z
  y = y.lptr
  z.sum = x.data + y.data
```

No variable is used beyond this program point
Liveness of Stack Data

```
while (x.data < max) {
    x = x.rptr
    y = x.lptr
    z = New class of z
    y = y.lptr
    z.sum = x.data + y.data
}
```

Current values of x, y, and z are used beyond this program point.

[Diagram of stack data with live and dead values highlighted]

Uday Khedker
IIT Bombay
Liveness of Stack Data

- Current values of x, y, and z are used beyond this program point
- The value of y is different before and after the assignment to y
Liveness of Stack Data

- The current values of \( x \) and \( y \) are used beyond this program point
- The current value of \( z \) is not used beyond this program point

\[
\begin{align*}
  w &= x \\
  \text{while } (x\text{.data} < \max) \\
  x &= x\text{.rptr} \\
  y &= x\text{.lptr} \\
  z &= \text{New class of } z \\
  y &= y\text{.lptr} \\
  z\text{.sum} &= x\text{.data} + y\text{.data}
\end{align*}
\]
The current values of \( x \) is used beyond this program point.

Current values of \( y \) and \( z \) are not used beyond this program point.
Liveness of Stack Data

- Nothing is known as of now
- Some information will be available in the next iteration point
Liveness of Stack Data

- Current value of x is used beyond this program point
- However its value is different before and after the assignment

```
w = x

while (x.data < max)
  x = x.rptr

y = x.lptr

z = New_class_of_z

y = y.lptr

z.sum = x.data + y.data
```
Liveness of Stack Data

- Current value of x is used beyond this program point
- There are two control flow paths beyond this program point
Liveness of Stack Data

Current value of x is used beyond this program point
Liveness of Stack Data

Current value of \( x \) is used beyond this program point
Liveness of Stack Data

\[ w = x \]
\[ \text{while } (x.\text{data} < \text{max}) \]
\[ x = x.\text{rptr} \]
\[ y = x.\text{lptr} \]
\[ z = \text{New class of } z \]
\[ y = y.\text{lptr} \]
\[ z.\text{sum} = x.\text{data} + y.\text{data} \]

End of iteration #1
Liveness of Stack Data

\[
\begin{align*}
  w &= x \\
  \text{while (} x.\text{data } < \text{ max} \text{)} \\
  x &= x.\text{rptr} \\
  y &= x.\text{lptr} \\
  z &= \text{New class of } z \\
  y &= y.\text{lptr} \\
  \text{z.sum } &= x.\text{data } + y.\text{data}
\end{align*}
\]
Applying Cedar Mesa Folk Wisdom to Heap Data

Liveness Analysis of Heap Data
If the while loop is not executed even once.

```
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
```
Applying Cedar Mesa Folk Wisdom to Heap Data

Liveness Analysis of Heap Data
If the while loop is executed once.

1. \( w = x \) \hspace{1cm} // 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. \( z.sum = x.data + y.data \)
Applying Cedar Mesa Folk Wisdom to Heap Data

Liveness Analysis of Heap Data

If the \texttt{while} loop is executed twice.

1. \texttt{w = x} \hspace{1cm} // \texttt{x} points to \texttt{ma}
2. \texttt{while (x.data < max)}
3. \texttt{x = x.rptr}
4. \texttt{y = x.lptr}
5. \texttt{z = New \ class\_of\_z}
6. \texttt{y = y.lptr}
7. \texttt{z.sum = x.data + y.data}
The Moral of the Story

- Mappings between access expressions and l-values keep changing.

- This is a *rule* for heap data. For stack and static data, it is an *exception*!

- Static analysis of programs has made significant progress for stack and static data.

What about heap data?

- Given two access expressions at a program point, do they have the same l-value?
- Given the same access expression at two program points, does it have the same l-value?
Our Solution

```
y = z = null
w = x
w = null
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
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
y = y.lptr
y.lptr = y.rptr = null
z.sum = x.data + y.data
x = y = z = null
```
**Our Solution**

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

**While loop is not executed even once**

[Diagram of Stack and Heap with labeled nodes and pointers]
Our Solution

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

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

While loop is not executed even once
Our Solution

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

While loop is not executed even once

Stack

Heap
Our Solution

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

While loop is not executed even once

Stack

Heap
Our Solution

y = z = null
1 w = x
   w = null
2 while (x.data < max)
   { x.lptr = null
   x = x.rptr
   } x.rptr = x.lptr.rptr = null
   x.lptr.lptr.lptr = null
   x.lptr.lptr.rptr = null
3 y = x.lptr
   x.lptr = y.rptr = null
   y.lptr.lptr = y.lptr.rptr = null
4 z = New class_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 = z = null

While loop is not executed even once
Our Solution

\begin{align*}
y = z &= \text{null} \\
w &= x \\
w &= \text{null} \\
2 \text{ while (x.data < max)} \{ x.lptr &= \text{null} \\
3 \quad x &= x.rptr \} \\
x.rptr &= x.lptr.rptr = \text{null} \\
x.lptr.lptr.lptr &= \text{null} \\
x.lptr.lptr.rptr &= \text{null} \\
4 \quad y &= x.lptr \\
x.lptr &= y.rptr = \text{null} \\
y.lptr.lptr &= y.lptr.rptr = \text{null} \\
5 \quad z &= \text{New class of } z \\
z.lptr &= z.rptr = \text{null} \\
6 \quad y &= y.lptr \\
y.lptr &= y.rptr = \text{null} \\
7 \quad z.sum &= x.data + y.data \\
x &= y = z = \text{null}
\end{align*}

While loop is not executed even once

---

Uday Khedker
IIT Bombay
**Our Solution**

1. \( y = z = \text{null} \)
2. \( w = x \)
   \( w = \text{null} \)
3. while \( (x\text{.data} < \text{max}) \)
   \[
   \begin{align*}
   &\{ \quad x\text{.lptr} = \text{null} \\
   &\quad x = x\text{.rptr} \\
   &x\text{.rptr} = x\text{.lptr}\text{.rptr} = \text{null} \\
   &x\text{.lptr}\text{.lptr}\text{.lptr} = \text{null} \\
   &x\text{.lptr}\text{.lptr}\text{.rptr} = \text{null}
   \end{align*}
   \]
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. \( z\text{.sum} = x\text{.data} + y\text{.data} \\
   x = y = z = \text{null} \)

**While loop is executed once**
Our Solution

```
y = z = null
1  w = x
   w = null
2  while (x.data < max)
   { x.lptr = null
     x = x.rptr   }
   x.rptr = x.lptr.rptr = null
   x.lptr.lptr.lptr = null
   x.lptr.lptr.rptr = null
3  y = x.lptr
   x.lptr = y.rptr = null
   y.lptr.lptr = y.lptr.rptr = null
4  z = New class_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 = z = null
```

While loop is executed twice
Some Observations

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

Node \( i \) is live but link \( a \rightarrow i \) is nullified
Some Observations

New access expressions are created. Can they cause exceptions?

1. \( y = z = \text{null} \)
2. \( w = x \)
3. \( w = \text{null} \)
4. \( \text{while (} x.\text{data < max}) \{ x.\text{lptr} = \text{null} \text{; } x = x.\text{rptr} \} \)
5. \( x.\text{rptr} = x.\text{lptr}.\text{rptr} = \text{null} \)
6. \( x.\text{lptr}.\text{lptr}.\text{lptr} = \text{null} \)
7. \( x.\text{lptr}.\text{lptr}.\text{rptr} = \text{null} \)
8. \( y = x.\text{lptr} \)
9. \( x.\text{lptr} = y.\text{rptr} = \text{null} \)
10. \( y.\text{lptr}.\text{lptr} = y.\text{lptr}.\text{rptr} = \text{null} \)
11. \( z = \text{New class of } z \)
12. \( z.\text{lptr} = z.\text{rptr} = \text{null} \)
13. \( y = y.\text{lptr} \)
14. \( y.\text{lptr} = y.\text{rptr} = \text{null} \)
15. \( z.\text{sum} = x.\text{data} + y.\text{data} \)
16. \( x = y = z = \text{null} \)
Goals of This Tutorial

- Main Goal: Explain how to get the modified program
- Side goals:
**Goals of This Tutorial**

- **Main Goal:** Explain how to get the modified program
- **Side goals:**
  - Explain the intuitions
  - Relate them to mathematical rigour
  - Highlight the frontiers

---

Uday Khedker
IIT Bombay
Part 3

Formulating Data Flow Analysis
Formulating Data Flow Analysis

- Live variables analysis and available expressions analysis
- Local and global data flow properties
- Common form of data flow equations
Live Variables Analysis

A variable $v$ is live at a program point $p$, if some path from $p$ to program exit contains an r-value occurrence of $v$ which is not preceded by an l-value occurrence of $v$. 

\[
v = a \ast b
\]

\[
a = v + 2
\]

\[
v = v + 2
\]
Live Variables Analysis

A variable $v$ is live at a program point $p$, if some path from $p$ to program exit contains an r-value occurrence of $v$ which is not preceded by an l-value occurrence of $v$. 

$v$ is live at $p$

\[ v = a \times b \]

\[ a = v + 2 \]

\[ v = a + 2 \]

\[ v = v + 2 \]
Live Variables Analysis

A variable $v$ is live at a program point $p$, if some path from $p$ to program exit contains an r-value occurrence of $v$ which is not preceded by an l-value occurrence of $v$. 

$v$ is live at $p$

$v$ is not live at $p$
A variable $v$ is live at a program point $p$, if some path from $p$ to program exit contains an r-value occurrence of $v$ which is not preceded by an l-value occurrence of $v$. 

$v$ is live at $p$

$v$ is not live at $p$

$v$ is live at $p$
Live Variables Analysis

A variable $v$ is live at a program point $p$, if some path from $p$ to program exit contains an r-value occurrence of $v$ which is not preceded by an l-value occurrence of $v$. 

Path based specification

$v$ is live at $p$

$v$ is not live at $p$

$v$ is live at $p$
Defining Data Flow Analysis for Live Variables Analysis

\[ L_{In}^i = L_{Gen}^i \cup (L_{Out}^i - L_{Kill}^i) \]

\[ L_{Gen}^k, L_{Kill}^k, L_{Out}^k = L_{In}^i \cup L_{In}^j \]
Defining Data Flow Analysis for Live Variables Analysis

Basic Blocks $\equiv$ Single statements or Maximal groups of sequentially executed statements
Defining Data Flow Analysis for Live Variables Analysis

Basic Blocks \equiv \text{Single statements or Maximal groups of sequentially executed statements}

Control Transfer
Defining Data Flow Analysis for Live Variables Analysis

\[ \text{LGen}_k, \text{LKill}_k \]

\[ \text{LGen}_i, \text{LKill}_i \]

\[ \text{LGen}_j, \text{LKill}_j \]
Defining Data Flow Analysis for Live Variables Analysis

Local Data Flow Properties

\[ LGen_k, \ LKill_k \]

\[ LGen_i, \ LKill_i \]

\[ LGen_j, \ LKill_j \]
Local Data Flow Properties for Live Variables Analysis

\[ LGen_b = \{ v \mid \text{variable } v \text{ is used in basic block } b \text{ and is not preceded by a definition of } v \} \]

\[ LKill_b = \{ v \mid \text{basic block } b \text{ contains a definition of } v \} \]
Local Data Flow Properties for Live Variables Analysis

LGenₜ = \{ v \mid \text{variable } v \text{ is used in basic block } b \text{ and is not preceded by a definition of } v \}\
LKillₜ = \{ v \mid \text{basic block } b \text{ contains a definition of } v \}
Local Data Flow Properties for Live Variables Analysis

r-value occurrence
Value is only read, e.g. \( x, y, z \) in
\[ x.\text{sum} = y.\text{data} + z.\text{data} \]

l-value occurrence
Value is modified e.g. \( y \) in
\[ y = x.lptr \]

\[ \text{LGen}_b = \{ v \mid \text{variable } v \text{ is used in basic block } b \text{ and is not preceded by a definition of } v \} \]

\[ \text{LKill}_b = \{ v \mid \text{basic block } b \text{ contains a definition of } v \} \]
Local Data Flow Properties for Live Variables Analysis

**r-value occurrence**
Value is only read, e.g. \( x, y, z \) in
\[
x.\text{sum} = y.\text{data} + z.\text{data}
\]

**l-value occurrence**
Value is modified e.g. \( y \) in
\[
y = x.lptr
\]

\[
LGen_b = \{ v \mid \text{variable } v \text{ is } \text{used} \text{ in basic block } b \text{ and is not } \text{preceded} \text{ by a definition of } v \}
\]

\[
LKill_b = \{ v \mid \text{basic block } b \text{ contains a definition of } v \}
\]

within \( b \)
Local Data Flow Properties for Live Variables Analysis

\[ L\text{Gen}_b = \{ v \mid \text{variable } v \text{ is used in basic block } b \text{ and is not preceded by a definition of } v \} \]

\[ L\text{Kill}_b = \{ v \mid \text{basic block } b \text{ contains a definition of } v \} \]
Defining Data Flow Analysis for Live Variables Analysis

\[ \text{LIn}_k = \text{LGen}_k \cup (\text{LOut}_k - \text{LKill}_k) \]

\[ \text{LOut}_k = \text{LIn}_i \cup \text{LIn}_j \]
Defining Data Flow Analysis for Live Variables Analysis

Global Data Flow Properties

\[ \text{LIn}_k = \text{LGen}_k \cup (\text{LOut}_k - \text{LKill}_k) \]

\[ \text{LOut}_k = \text{LIn}_i \cup \text{LIn}_j \]

\[ \text{LIn}_i \]

\[ \text{LIn}_j \]
Defining Data Flow Analysis for Live Variables Analysis

Global Data Flow Properties

LIn_k = LGen_k ∪ (LOut_k − LKill_k)

LOut_k = LIn_i ∪ LIn_j

LIn_i

LIn_j
Data Flow Equations For Live Variables Analysis

\[
\begin{align*}
\text{LIn}_b & = \text{LGen}_b \cup (\text{LOut}_b - \text{LKill}_b) \\
\text{LOut}_b & = \begin{cases} 
\emptyset & \text{if } b \text{ is the exit node} \\
\bigcup_{s \in \text{succ}(b)} \text{LIn}_s & \text{otherwise}
\end{cases}
\end{align*}
\]
Data Flow Equations For Live Variables Analysis

\[ \text{LIn}_b = \text{LGen}_b \cup (\text{LOut}_b - \text{LKill}_b) \]

\[ \text{LOut}_b = \begin{cases} \emptyset & \text{if } b \text{ is the exit node} \\ \bigcup_{s \in \text{succ}(b)} \text{LIn}_s & \text{otherwise} \end{cases} \]

Alternatively,

\[ \text{LIn}_b = f_b(\text{LOut}_b), \quad \text{where} \]

\[ f_b(X) = \text{LGen}_b \cup (X - \text{LKill}_b) \]
Data Flow Equations For Live Variables Analysis

\[ \text{LIn}_b = \text{LGen}_b \cup (\text{LOut}_b - \text{LKill}_b) \]

\[ \text{LOut}_b = \begin{cases} 
\emptyset & \text{if } b \text{ is the exit node} \\
\bigcup_{s \in \text{succ}(b)} \text{LIn}_s & \text{otherwise} 
\end{cases} \]

Alternatively,

\[ \text{LIn}_b = f_b(\text{LOut}_b), \quad \text{where} \]

\[ f_b(X) = \text{LGen}_b \cup (X - \text{LKill}_b) \]

\text{LIn}_b \text{ and } \text{LOut}_b \text{ are sets of variables.}
Performing Live Variables Analysis

\[ \text{LGen} = \{x\}, \quad \text{LKill} = \{w\} \]

\[ w = x \]

\[ \text{LGen} = \{x\}, \quad \text{LKill} = \emptyset \]

\[ \text{while} \ (x.\text{data} < \text{max}) \]

\[ \text{LGen} = \{x\}, \quad \text{LKill} = \{y\} \]

\[ y = x.\text{lptr} \]

\[ \text{LGen} = \emptyset, \quad \text{LKill} = \{z\} \]

\[ z = \text{New class of z} \]

\[ \text{LGen} = \{y\}, \quad \text{LKill} = \{y\} \]

\[ y = y.\text{lptr} \]

\[ \text{LGen} = \{x, y, z\}, \quad \text{LKill} = \emptyset \]

\[ z.\text{sum} = x.\text{data} + y.\text{data} \]
Performing Live Variables Analysis

\[ \text{LGen} = \{x\}, \text{LKill} = \{w\} \]

\[ w = x \]

\[ \text{LGen} = \{x\}, \text{LKill} = \emptyset \]

\[ \text{while } (x.\text{data} < \text{max}) \]

\[ \text{LGen} = \{x\}, \text{LKill} = \{\} \]

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

\[ \text{LGen} = \{\}, \text{LKill} = \{y\} \]

\[ y = x.\text{lptr} \]

\[ \text{LGen} = \{\}, \text{LKill} = \{\} \]

\[ z = \text{New class of } z \]

\[ \text{LGen} = \{y\}, \text{LKill} = \{y\} \]

\[ y = y.\text{lptr} \]

\[ \text{LGen} = \{x, y, z\}, \text{LKill} = \emptyset \]

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

LGen and LKill need not be mutually exclusive
Performing Live Variables Analysis

LGen = \{x\}, LKill = \{w\}

w = x

LGen = \{x\}, LKill = \{\emptyset\}
while (x.data < max)

LGen = \{x\}, LKill = \{y\}
y = x.lptr

LGen = \{\emptyset\}, LKill = \{z\}
z = New class_of_z

LGen = \{y\}, LKill = \{y\}
y = y.lptr

LGen = \{x, y, z\}, LKill = \{\emptyset\}
z.sum = x.data + y.data

z is an r-value occurrence and not an l-value occurrence
Performing Live Variables Analysis

\[ \text{LGen} = \{x\}, \ \text{LKill} = \{w\} \]
\[ w = x \]

\[ \text{LGen} = \{x\}, \ \text{LKill} = \emptyset \]
\[ \text{while} \ (x\text{.data} < \text{max}) \]

\[ \text{LGen} = \{x\}, \ \text{LKill} = \{y\} \]
\[ y = x\text{.lptr} \]

\[ \text{LGen} = \emptyset, \ \text{LKill} = \{z\} \]
\[ z = \text{New class of z} \]

\[ \text{LGen} = \{y\}, \ \text{LKill} = \{y\} \]
\[ y = y\text{.lptr} \]

\[ \text{LGen} = \{x, y, z\}, \ \text{LKill} = \emptyset \]
\[ z\text{.sum} = x\text{.data} + y\text{.data} \]

\[ x, y, z \text{ are considered to be used based purely on local use even if the value of z is not use later. A different analysis called faint variables analysis improves on this.} \]
Performing Live Variables Analysis

\[
LGen = \{x\}, \quadLKil = \{w\}
\]

\[
w = x
\]

\[
LGen = \{x\}, \quadLKil = \emptyset
\]

while \((x.\text{data} < \text{max})\)

\[
LGen = \{x\}, \quadLKil = \emptyset
\]

\[
x = x.rptr
\]

\[
LGen = \{y\}, \quadLKil = \emptyset
\]

\[
y = x.lptr
\]

\[
LGen = \emptyset, \quadLKil = \{z\}
\]

\[
z = \text{New class of } z
\]

\[
LGen = \emptyset, \quadLKil = \{y\}
\]

\[
y = y.lptr
\]

\[
LGen = \{x, y, z\}, \quadLKil = \emptyset
\]

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

Initialization
Performing Live Variables Analysis

\[ \text{LGen} = \{ x \}, \quad \text{LKill} = \{ w \} \]

\[ w = x \]

\[ \text{LGen} = \{ x \}, \quad \text{LKill} = \emptyset \]

\[ \text{while} \ (x.\text{data} < \text{max}) \]

\[ \text{LGen} = \{ x \}, \quad \text{LKill} = \emptyset \]

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

\[ \text{LGen} = \emptyset, \quad \text{LKill} = \{ z \} \]

\[ z = \text{New class of z} \]

\[ \text{LGen} = \{ y \}, \quad \text{LKill} = \{ y \} \]

\[ y = y.\text{lptr} \]

\[ \text{LGen} = \{ x, y, z \}, \quad \text{LKill} = \emptyset \]

\[ z.\text{sum} = x.\text{data} + y.\text{data} \]
Performing Live Variables Analysis

\[
\begin{align*}
LGen &= \{x\}, \quad LKill = \{w\} \\
&\quad w = x \\
LGen &= \{x\}, \quad LKill = \emptyset \\
&\quad \text{while } (x.\text{data} < \text{max}) \\
LGen &= \{x\}, \quad LKill = \emptyset \\
LGen &= \emptyset, \quad LKill = \{z\} \\
&\quad z = \text{New class of } z \\
LGen &= \{y\}, \quad LKill = \{y\} \\
&\quad y = y.\text{lptr} \\
LGen &= \{x, y, z\}, \quad LKill = \emptyset \\
&\quad z.\text{sum} = x.\text{data} + y.\text{data} \\
LGen &= \{x, y\}, \quad LKill = \emptyset \\
LGen &= \{x, y\}, \quad LKill = \{x\} \\
&\quad x = x.\text{rptr} \\
\end{align*}
\]
Performing Live Variables Analysis

\[ \text{LGen} = \{x\}, \ \text{LKill} = \{w\} \]

\[ w = x \]

\[ \text{LGen} = \{x\}, \ \text{LKill} = \emptyset \]

\[ \text{while} \ (x\.\text{data} < \text{max}) \]

\[ \text{LGen} = \{x\}, \ \text{LKill} = \{y, z\} \]

\[ y = x\.\text{lptr} \]

\[ z = \text{New class of z} \]

\[ y = y\.\text{lptr} \]

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

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

\[ \text{LGen} = \{x\}, \ \text{LKill} = \{x\} \]
Available Expressions Analysis

An expression $e$ is available at a program point $p$, if every path from program entry to $p$ contains an evaluation of $e$ which is not followed by a definition of any operand of $e$. 

\[
\begin{align*}
&\text{Entry} \\
&a \times b \\
&\quad a \times b \\
&\quad a \times b \\
&p \\
&\quad ( ) \\
&\text{Exit}
\end{align*}
\]
Available Expressions Analysis

An expression $e$ is available at a program point $p$, if every path from program entry to $p$ contains an evaluation of $e$ which is not followed by a definition of any operand of $e$.

$a \times b$ is available at $p$
Available Expressions Analysis

An expression $e$ is available at a program point $p$, if every path from program entry to $p$ contains an evaluation of $e$ which is not followed by a definition of any operand of $e$. 

$a \times b$ is available at $p$

$a \times b$ is not available at $p$
Available Expressions Analysis

An expression $e$ is available at a program point $p$, if every path from program entry to $p$ contains an evaluation of $e$ which is not followed by a definition of any operand of $e$.

- $a \times b$ is available at $p$
- $a \times b$ is not available at $p$
- $a \times b$ is not available at $p$
Local Data Flow Properties for Available Expressions Analysis

\[ \text{AGen}_b = \{ e \mid \text{expression } e \text{ is evaluated in basic block } b \text{ and this evaluation is not followed by a definition of any operand of } e \} \]

\[ \text{AKill}_b = \{ e \mid \text{basic block } b \text{ contains a definition of an operand of } e \} \]
Data Flow Equations For Available Expressions Analysis

\[
\begin{align*}
\text{AIn}_b &= \begin{cases} 
\emptyset & \text{if } b \text{ is the entry node} \\
\bigcap_{p \in \text{pred}(b)} \text{AOut}_p & \text{otherwise}
\end{cases} \\
\text{AOut}_b &= \text{AGen}_b \cup (\text{AIn}_b - \text{AKill}_b)
\end{align*}
\]
Data Flow Equations For Available Expressions Analysis

\[
A\text{In}_b = \begin{cases} 
\emptyset & \text{if } b \text{ is the entry node} \\
\bigcap_{p \in \text{pred}(b)} A\text{Out}_p & \text{otherwise}
\end{cases}
\]

\[
A\text{Out}_b = A\text{Gen}_b \cup (A\text{In}_b - A\text{Kill}_b)
\]

Alternatively,

\[
A\text{Out}_b = f_b(A\text{In}_b), \quad \text{where}
\]

\[
f_b(X) = A\text{Gen}_b \cup (X - A\text{Kill}_b)
\]
Data Flow Equations For Available Expressions Analysis

\[
\begin{align*}
\text{AIn}_b &= \begin{cases} 
\emptyset & \text{if } b \text{ is the entry node} \\
\bigcap_{p \in \text{pred}(b)} \text{AOut}_p & \text{otherwise}
\end{cases} \\
\text{AOOut}_b &= \text{AGen}_b \cup (\text{AIn}_b - \text{AKill}_b)
\end{align*}
\]

Alternatively,

\[
\text{AOOut}_b = f_b(\text{AIn}_b), \quad \text{where}
\]

\[
f_b(X) = \text{AGen}_b \cup (X - \text{AKill}_b)
\]

\text{AIn}_b \text{ and } \text{AOOut}_b \text{ are sets of expressions.}
An Example of Available Expressions Analysis

Let $e_1 \equiv a \times b$, $e_2 \equiv b \times c$, $e_3 \equiv c \times d$, $e_4 \equiv d \times e$

<table>
<thead>
<tr>
<th>Node</th>
<th>Computed</th>
<th>Killed</th>
<th>Available</th>
<th>Redund.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>${e_1, e_2}$</td>
<td>1100</td>
<td>$\emptyset$</td>
<td>0000</td>
</tr>
<tr>
<td>2</td>
<td>${e_3}$</td>
<td>0010</td>
<td>$\emptyset$</td>
<td>0000</td>
</tr>
<tr>
<td>3</td>
<td>$\emptyset$</td>
<td>0000</td>
<td>${e_2, e_3}$</td>
<td>0110</td>
</tr>
<tr>
<td>4</td>
<td>$\emptyset$</td>
<td>0000</td>
<td>${e_3, e_4}$</td>
<td>0011</td>
</tr>
<tr>
<td>5</td>
<td>${e_1, e_4}$</td>
<td>1001</td>
<td>$\emptyset$</td>
<td>0000</td>
</tr>
<tr>
<td>6</td>
<td>${e_4}$</td>
<td>0001</td>
<td>$\emptyset$</td>
<td>0000</td>
</tr>
</tbody>
</table>
An Example of Available Expressions Analysis

Let $e_1 \equiv a \times b$, $e_2 \equiv b \times c$, $e_3 \equiv c \times d$, $e_4 \equiv d \times e$

<table>
<thead>
<tr>
<th>Node</th>
<th>Computed</th>
<th>Killed</th>
<th>Available</th>
<th>Redund.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>${e_1, e_2}$ 1100</td>
<td>$\emptyset$</td>
<td>0000</td>
<td>$\emptyset$ 0000</td>
</tr>
<tr>
<td>2</td>
<td>${e_3}$ 0010</td>
<td>$\emptyset$</td>
<td>0000</td>
<td>${e_1}$ 1000</td>
</tr>
<tr>
<td>3</td>
<td>$\emptyset$ 0000</td>
<td>${e_2, e_3}$ 0110</td>
<td>${e_1, e_3}$ 1010</td>
<td>$\emptyset$ 0000</td>
</tr>
<tr>
<td>4</td>
<td>$\emptyset$ 0000</td>
<td>${e_3, e_4}$ 0011</td>
<td>${e_1, e_3}$ 1010</td>
<td>$\emptyset$ 0000</td>
</tr>
<tr>
<td>5</td>
<td>${e_1, e_4}$ 1001</td>
<td>$\emptyset$</td>
<td>0000</td>
<td>${e_1}$ 1000</td>
</tr>
<tr>
<td>6</td>
<td>${e_4}$ 0001</td>
<td>$\emptyset$</td>
<td>0000</td>
<td>${e_1, e_4}$ 1001</td>
</tr>
</tbody>
</table>
An Example of Available Expressions Analysis

Let $e_1 \equiv a \ast b$, $e_2 \equiv b \ast c$, $e_3 \equiv c \ast d$, $e_4 \equiv d \ast e$

<table>
<thead>
<tr>
<th>Node</th>
<th>Computed</th>
<th>Killed</th>
<th>Available</th>
<th>Redund.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>${e_1, e_2}$</td>
<td>1100</td>
<td>$\emptyset$</td>
<td>0000</td>
</tr>
<tr>
<td>2</td>
<td>${e_3}$</td>
<td>0010</td>
<td>$\emptyset$</td>
<td>0000</td>
</tr>
<tr>
<td>3</td>
<td>$\emptyset$</td>
<td>0000</td>
<td>${e_2, e_3}$</td>
<td>0110</td>
</tr>
<tr>
<td>4</td>
<td>$\emptyset$</td>
<td>0000</td>
<td>${e_3, e_4}$</td>
<td>0011</td>
</tr>
<tr>
<td>5</td>
<td>${e_1, e_4}$</td>
<td>1001</td>
<td>$\emptyset$</td>
<td>0000</td>
</tr>
<tr>
<td>6</td>
<td>${e_4}$</td>
<td>0001</td>
<td>$\emptyset$</td>
<td>0000</td>
</tr>
</tbody>
</table>
### An Example of Available Expressions Analysis

Let $e_1 \equiv a \times b$, $e_2 \equiv b \times c$, $e_3 \equiv c \times d$, $e_4 \equiv d \times e$

<table>
<thead>
<tr>
<th>Node</th>
<th>Computed</th>
<th>Killed</th>
<th>Available</th>
<th>Redund.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>${e_1, e_2}$</td>
<td>1100</td>
<td>$\emptyset$</td>
<td>0000</td>
</tr>
<tr>
<td>2</td>
<td>${e_3}$</td>
<td>0010</td>
<td>$\emptyset$</td>
<td>0000</td>
</tr>
<tr>
<td>3</td>
<td>$\emptyset$</td>
<td>0000</td>
<td>${e_2, e_3}$</td>
<td>0110</td>
</tr>
<tr>
<td>4</td>
<td>$\emptyset$</td>
<td>0000</td>
<td>${e_3, e_4}$</td>
<td>0011</td>
</tr>
<tr>
<td>5</td>
<td>${e_1, e_4}$</td>
<td>1001</td>
<td>$\emptyset$</td>
<td>0000</td>
</tr>
<tr>
<td>6</td>
<td>${e_4}$</td>
<td>0001</td>
<td>$\emptyset$</td>
<td>0000</td>
</tr>
</tbody>
</table>
An Example of Available Expressions Analysis

Let $e_1 \equiv a \ast b$, $e_2 \equiv b \ast c$, $e_3 \equiv c \ast d$, $e_4 \equiv d \ast e$

<table>
<thead>
<tr>
<th>Node</th>
<th>Computed</th>
<th>Killed</th>
<th>Available</th>
<th>Redund.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>${e_1, e_2}$</td>
<td>1100</td>
<td>$\emptyset$</td>
<td>0000</td>
</tr>
<tr>
<td>2</td>
<td>${e_3}$</td>
<td>0010</td>
<td>$\emptyset$</td>
<td>0000</td>
</tr>
<tr>
<td>3</td>
<td>$\emptyset$</td>
<td>0000</td>
<td>${e_2, e_3}$</td>
<td>0110</td>
</tr>
<tr>
<td>4</td>
<td>$\emptyset$</td>
<td>0000</td>
<td>${e_3, e_4}$</td>
<td>0011</td>
</tr>
<tr>
<td>5</td>
<td>${e_1, e_4}$</td>
<td>1001</td>
<td>$\emptyset$</td>
<td>0000</td>
</tr>
<tr>
<td>6</td>
<td>${e_4}$</td>
<td>0001</td>
<td>$\emptyset$</td>
<td>0000</td>
</tr>
</tbody>
</table>

Final Result

\[
\begin{array}{c}
0000 \\
1 \\
1100 \\
1000 \\
1010 \\
3 \\
1000 \\
1000 \\
5 \\
1001 \\
1001 \\
1001
\end{array}
\]
Available Expressions Analysis.

- Used for common subexpression elimination.
Using Data Flow Information

Available Expressions Analysis.

- Used for common subexpression elimination.
  - If an expression is available at the entry of a block $b$ and
Available Expressions Analysis.

- Used for common subexpression elimination.
  - If an expression is available at the entry of a block $b$ and
  - a computation of the expression exists in $b$ such that
Using Data Flow Information

Available Expressions Analysis.

- Used for common subexpression elimination.
  - If an expression is available at the entry of a block $b$ and
  - a computation of the expression exists in $b$ such that
  - it is not preceded by a definition of any of its operands
Using Data Flow Information

Available Expressions Analysis.

- Used for common subexpression elimination.
  - If an expression is available at the entry of a block \( b \) and
  - a computation of the expression exists in \( b \) such that
  - it is not preceded by a definition of any of its operands

Then the expression is redundant.
Using Data Flow Information

Available Expressions Analysis.

- Used for common subexpression elimination.
  - If an expression is available at the entry of a block $b$ and
  - a computation of the expression exists in $b$ such that
  - it is not preceded by a definition of any of its operands

  Then the expression is redundant.

- Expression must be *upwards exposed* or *locally anticipable*.
Available Expressions Analysis.

- Used for common subexpression elimination.
  - If an expression is available at the entry of a block $b$ and
  - a computation of the expression exists in $b$ such that
  - it is not preceded by a definition of any of its operands

Then the expression is redundant.

- Expression must be *upwards exposed* or *locally anticipable*.
- Expressions in $\text{Gen}_b$ are *downwards exposed*. 
Using Data Flow Information

Live Variables Analysis.

- Used for register allocation. If variable $x$ is live in a basic block $b$, it is a potential candidate for register allocation.
Using Data Flow Information

Live Variables Analysis.

- Used for register allocation. If variable $x$ is live in a basic block $b$, it is a potential candidate for register allocation.

- Used for dead code elimination. If variable $x$ is not live after an assignment $x = \ldots$, then the assignment is redundant and can be deleted as dead code.
Reaching Definitions Analysis

- A definition $d_x : x = y$ reaches a program point $u$ if it appears (without a redefinition of $x$) on some path from program entry to $u$

- Application: Copy Propagation
  A use of a variable $x$ at a program point $u$ can be replaced by $y$ if $d_x : x = y$ is the only definition which reaches $p$ and $y$ is not modified between the point of $d_x$ and $p$. 
Defining Data Flow Analysis for Reaching Definitions Analysis

Let $d_v$ be a definition of variable $v$

$$\text{Gen}_b = \{ d_v \mid \text{variable } v \text{ is defined in basic block } b \text{ and this definition is not followed (within } b \text{) by a definition of } v \}$$

$$\text{Kill}_b = \{ d_v \mid \text{basic block } b \text{ contains a definition of } v \}$$
Data Flow Equations for Reaching Definitions Analysis

\[
\begin{align*}
\text{IN}_b &= \begin{cases} 
\{ \} & \text{if } b \text{ is the entry node} \\
\bigcup_{p \in \text{pred}(b)} \text{OUT}_p & \text{otherwise}
\end{cases} \\
\text{OUT}_b &= \text{Gen}_b \cup (\text{IN}_b - \text{Kill}_b)
\end{align*}
\]

IN$_b$ and OUT$_b$ are sets of definitions.
Very Busy Expressions Analysis

An expression $e$ is very busy at a program point $u$, if every path from $u$ to the program exit contains an evaluation of $e$ which is not preceded by a redefinition of any operand of $e$.

Application: Safety of Code Motion
Safety of Code Motion

- Expression \( a/b \) is not very busy at the exit of 1.
- Moving \( a/b \) to the exit of 1 is unsafe.
Defining Data Flow Analysis for Very Busy Expressions Analysis

\[ \text{Gen}_b = \{ e \mid \text{expression } e \text{ is evaluated in basic block } b \text{ and this evaluation is not preceded (within } b \text{) by a definition of any operand of } e \} \]

\[ \text{Kill}_b = \{ e \mid \text{basic block } b \text{ contains a definition of an operand of } e \} \]
Data Flow Equations for Very Busy Expressions Analysis

\[
\begin{align*}
\text{IN}_b &= \text{Gen}_b \cup (\text{OUT}_b - \text{Kill}_b) \\
\text{OUT}_b &= \begin{cases} 
\emptyset & \text{if } b \text{ is the exit node} \\
\bigcap_{a \in \text{succ}(b)} \text{IN}_s & \text{otherwise}
\end{cases}
\end{align*}
\]

\text{IN}_b \text{ and } \text{OUT}_b \text{ are sets of expressions}
Common Form of Data Flow Equations

\[ X_i = f(Y_i) \]

\[ Y_i = \bigcap X_j \]
Common Form of Data Flow Equations

So far we have seen sets (or bit vectors). Could be entities other than sets.

\[ X_i = f(Y_i) \]
\[ Y_i = \bigcap X_j \]
Common Form of Data Flow Equations

So far we have seen sets (or bit vectors). Could be entities other than sets.

\[ X_i = f(Y_i) \]
\[ Y_i = \bigcap X_j \]

Data Flow Information

Flow Function
Common Form of Data Flow Equations

So far we have seen sets (or bit vectors). Could be entities other than sets.

Data Flow Information

\[ X_i = f(Y_i) \]

\[ Y_i = \bigcap X_j \]

Flow Function

Confluence

So far we have seen \( \cup \) and \( \cap \). Could be other operations.
# A Taxonomy of Bit Vector Data Flow Frameworks

<table>
<thead>
<tr>
<th></th>
<th>Confluence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Union</td>
</tr>
<tr>
<td>Forward</td>
<td>Reaching Definitions</td>
</tr>
<tr>
<td>Backward</td>
<td>Live Variables</td>
</tr>
<tr>
<td>Bidirectional (limited)</td>
<td></td>
</tr>
</tbody>
</table>
# A Taxonomy of Bit Vector Data Flow Frameworks

## Any Path

<table>
<thead>
<tr>
<th></th>
<th>Confluence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Union</td>
</tr>
<tr>
<td>Forward</td>
<td>Reaching Definitions</td>
</tr>
<tr>
<td>Backward</td>
<td>Live Variables</td>
</tr>
<tr>
<td>Bidirectional (limited)</td>
<td></td>
</tr>
</tbody>
</table>
A Taxonomy of Bit Vector Data Flow Frameworks

<table>
<thead>
<tr>
<th></th>
<th>Confluence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Union</td>
</tr>
<tr>
<td>Forward</td>
<td>Reaching Definitions</td>
</tr>
<tr>
<td>Backward</td>
<td>Live Variables</td>
</tr>
<tr>
<td>Bidirectional</td>
<td></td>
</tr>
</tbody>
</table>

Any Path

All Paths
# A Taxonomy of Bit Vector Data Flow Frameworks

<table>
<thead>
<tr>
<th></th>
<th>Confluence</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Union</td>
<td>Intersection</td>
</tr>
<tr>
<td>Forward</td>
<td>Reaching Definitions</td>
<td>Available Expressions</td>
</tr>
<tr>
<td>Backward</td>
<td>Live Variables</td>
<td>Anticipable Expressions</td>
</tr>
<tr>
<td>Bidirectional (limited)</td>
<td>Partial Redundancy Elimination (Original M-R Formulation)</td>
<td></td>
</tr>
</tbody>
</table>
## A Taxonomy of Bit Vector Data Flow Frameworks

<table>
<thead>
<tr>
<th></th>
<th>Confluence</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Union</td>
<td>Intersection</td>
<td></td>
</tr>
<tr>
<td>Forward</td>
<td>Reaching Definitions</td>
<td>Available Expressions</td>
<td></td>
</tr>
<tr>
<td>Backward</td>
<td>Live Variables</td>
<td>Anticipable Expressions</td>
<td></td>
</tr>
<tr>
<td>Bidirectional (limited)</td>
<td>Partial Redundancy Elimination (Original M-R Formulation)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Any Path**

**All Paths**
### A Taxonomy of Bit Vector Data Flow Frameworks

<table>
<thead>
<tr>
<th>Method</th>
<th>Confluence</th>
<th>Union</th>
<th>Intersection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>Reaching Definitions</td>
<td>Available Expressions</td>
<td></td>
</tr>
<tr>
<td>Backward</td>
<td>Live Variables</td>
<td>Anticipable Expressions</td>
<td></td>
</tr>
<tr>
<td>Bidirectional</td>
<td></td>
<td>Partial Redundancy Elimination</td>
<td>(Original M-R Formulation)</td>
</tr>
<tr>
<td>(limited)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Any Path**
- **All Paths**
An Introduction to Constant Propagation

\[ a = 1 \]
\[ b = 2 \]
\[ c = a + b \]

\[ c = a + b \]
\[ d = a \times b \]

\[ d = c - 1 \]
\[ a = 2 \]
\[ b = 1 \]
\[ c = a + b \]
An Introduction to Constant Propagation

\[ a = 1 \]
\[ b = 2 \]
\[ c = a + b \]
\[ d = a \times b \]
\[ d = c - 1 \]
\[ a = 2 \]
\[ b = 1 \]
\[ c = a + b \]

Execution Sequence
\[ \langle a, b, c, d \rangle \]
\[ \langle ?, ?, ?, ? \rangle \]
An Introduction to Constant Propagation

\[
\begin{align*}
 n_1 & : \\
 a &= 1 \\
 b &= 2 \\
 c &= a + b \\

 n_2 & : \\
 c &= a + b \\
 d &= a \times b \\

 n_3 & : \\
 d &= c - 1 \\
 a &= 2 \\
 b &= 1 \\
 c &= a + b
\end{align*}
\]

Execution Sequence
\[
\langle a, b, c, d \rangle \\
\langle ?, ?, ?, ? \rangle \\
\langle 1, 2, 3, ? \rangle
\]
An Introduction to Constant Propagation

Execution Sequence

\[ \langle a, b, c, d \rangle \]

\[ \langle ?, ?, ?, ? \rangle \]

\[ \langle 1, 2, 3, ? \rangle \]

\[ \langle 1, 2, 3, 2 \rangle \]
An Introduction to Constant Propagation

\[
\begin{align*}
\begin{array}{l}
a = 1 \\
b = 2 \\
c = a + b
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{l}
c = a + b \\
d = a \times b
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{l}
d = c - 1 \\
a = 2 \\
b = 1 \\
c = a + b
\end{array}
\end{align*}
\]

\[
\begin{align*}
\langle a, b, c, d \rangle \\
\langle 1, 2, 3, 2 \rangle \\
\langle 2, 1, 3, 2 \rangle
\end{align*}
\]
An Introduction to Constant Propagation

\[ a = 1 \]
\[ b = 2 \]
\[ c = a + b \]

\[ c = a + b \]
\[ d = a \times b \]

\[ d = c - 1 \]
\[ a = 2 \]
\[ b = 1 \]
\[ c = a + b \]

\[ \langle a, b, c, d \rangle \]
\[ \langle ?, ?, ?, ? \rangle \]
\[ \langle 1, 2, 3, ? \rangle \]
\[ \langle 1, 2, 3, 2 \rangle \]
\[ \langle 2, 1, 3, 2 \rangle \]
An Introduction to Constant Propagation

Execution Sequence

\[ \langle a, b, c, d \rangle \]
\[ \langle ?, ?, ?, ? \rangle \]
\[ \langle 1, 2, 3, ? \rangle \]
\[ \langle 1, 2, 3, 2 \rangle \]
\[ \langle 2, 1, 3, 2 \rangle \]
\[ \langle 2, 1, 3, 2 \rangle \]
\[ \langle 2, 1, 3, 2 \rangle \]

n₁

\[ a = 1 \]
\[ b = 2 \]
\[ c = a + b \]

n₂

\[ c = a + b \]
\[ d = a \times b \]

n₃

\[ d = c - 1 \]
\[ a = 2 \]
\[ b = 1 \]
\[ c = a + b \]
An Introduction to Constant Propagation

\[
\begin{align*}
\text{n}_1 & \quad a &= 1 \\
& \quad b &= 2 \\
& \quad c &= a + b \\
\text{n}_2 & \quad c &= a + b \\
& \quad d &= a \times b \\
\text{n}_3 & \quad d &= c - 1 \\
& \quad a &= 2 \\
& \quad b &= 1 \\
& \quad c &= a + b \\
\end{align*}
\]

Execution Sequence
\[
\langle a, b, c, d \rangle \quad \langle ?, ?, ?, ? \rangle \quad \langle 1, 2, 3, ? \rangle \quad \langle ?,?,?,? \rangle 
\]

IN

OUT

\[
\langle 2, 1, 3, 2 \rangle \quad \langle 2, 1, 3, 2 \rangle \quad \langle 2, 1, 3, 2 \rangle \quad \langle 2, 1, 3, 2 \rangle 
\]
An Introduction to Constant Propagation

\[
\begin{align*}
\text{n}_1 & : \quad a = 1 \\
& \quad b = 2 \\
& \quad c = a + b \\
\text{n}_2 & : \quad c = a + b \\
& \quad d = a \times b \\
\text{n}_3 & : \quad d = c - 1 \\
& \quad a = 2 \\
& \quad b = 1 \\
& \quad c = a + b
\end{align*}
\]

Execution Sequence

\[
\langle a, b, c, d \rangle \\
\langle ?, ?, ?, ? \rangle \\
\langle 1, 2, 3, ? \rangle \\
\langle 1, 2, 3, 2 \rangle \\
\langle 2, 1, 3, 2 \rangle \\
\langle 2, 1, 3, 2 \rangle \\
\langle 2, 1, 3, 2 \rangle \\
\langle 2, 1, 3, 2 \rangle \\
\ldots
\]
An Introduction to Constant Propagation

Summary Values
\[ \langle ?, ?, ?, ? \rangle \]

Execution Sequence
\[ \langle a, b, c, d \rangle \]
\[ \downarrow \]
\[ \langle ?, ?, ?, ? \rangle \]

\[ n_1 \]
\[ \langle 1, 2, 3, ? \rangle \] IN
\[ n_2 \]
\[ \langle 1, 2, 3, 2 \rangle \] OUT
\[ n_3 \]
\[ \langle 2, 1, 3, 2 \rangle \]
\[ n_2 \]
\[ \langle 2, 1, 3, 2 \rangle \]
\[ n_3 \]
\[ \langle 2, 1, 3, 2 \rangle \]
\[ n_2 \]
\[ \langle 2, 1, 3, 2 \rangle \]
\[ n_3 \]
\[ \langle 2, 1, 3, 2 \rangle \]
\[ \ldots \]

Uday Khedker
IIT Bombay
An Introduction to Constant Propagation

Summary Values

\[ \langle ?, ?, ?, ? \rangle \]
\[ \langle 1, 2, 3, ? \rangle \]

Execution Sequence

\[ \langle a, b, c, d \rangle \]
\[ \langle ?, ?, ?, ? \rangle \]
\[ \langle 1, 2, 3, ? \rangle \]
\[ \langle 1, 2, 3, 2 \rangle \]
\[ \langle 2, 1, 3, 2 \rangle \]
\[ \langle 2, 1, 3, 2 \rangle \]
\[ \langle 2, 1, 3, 2 \rangle \]
\[ \langle 2, 1, 3, 2 \rangle \]

\[ \vdots \]
An Introduction to Constant Propagation

Summary Values

\[ \langle a, b, c, d \rangle \]
\[ \langle 1, 2, 3, ? \rangle \]
\[ \langle ?, ?, ?, ? \rangle \]

Execution Sequence

\[ n_1 \]
\[ n_2 \]
\[ n_3 \]

\[ \langle 1, 2, 3, ? \rangle \]
\[ \langle 1, 2, 3, 2 \rangle \]
\[ \langle 2, 1, 3, 2 \rangle \]
\[ \langle 2, 1, 3, 2 \rangle \]
\[ \langle 2, 1, 3, 2 \rangle \]
\[ \langle 2, 1, 3, 2 \rangle \]

\[ \ldots \]
An Introduction to Constant Propagation

Summary Values

\( n_1 \)
\[
\begin{align*}
  a &= 1 \\
  b &= 2 \\
  c &= a + b
\end{align*}
\]
\( \langle ?, ?, ?, ? \rangle \)
\( \langle 1, 2, 3, ? \rangle \)

\( n_2 \)
\[
\begin{align*}
  c &= a + b \\
  d &= a \times b
\end{align*}
\]
\( \langle \times, \times, 3, 2 \rangle \)
\( \langle \times, \times, 3, 2 \rangle \)

\( n_3 \)
\[
\begin{align*}
  d &= c - 1 \\
  a &= 2 \\
  b &= 1 \\
  c &= a + b
\end{align*}
\]
\( \langle ?, ?, ?, ? \rangle \)

Execution Sequence

\[ \langle a, b, c, d \rangle \]
\[ \langle ?, ?, ?, ? \rangle \]
\[ \langle 1, 2, 3, ? \rangle \]
\[ \langle 1, 2, 3, 2 \rangle \]
\[ \langle 2, 1, 3, 2 \rangle \]
\[ \langle 2, 1, 3, 2 \rangle \]
\[ \langle 2, 1, 3, 2 \rangle \]
\[ \langle 2, 1, 3, 2 \rangle \]
\[ \ldots \]
\[ \langle 2, 1, 3, 2 \rangle \]

\[ \langle 2, 1, 3, 2 \rangle \]

\[ \langle 2, 1, 3, 2 \rangle \]

\[ \langle 2, 1, 3, 2 \rangle \]

\[ \langle 2, 1, 3, 2 \rangle \]

\[ \langle 2, 1, 3, 2 \rangle \]

\[ \langle 2, 1, 3, 2 \rangle \]
An Introduction to Constant Propagation

Summary Values

\[ a = 1 \]
\[ b = 2 \]
\[ c = a + b \]

\[ \langle 1, 2, 3, ? \rangle \]

\[ c = a + b \]
\[ d = a \times b \]

\[ \langle \times, \times, 3, 2 \rangle \]

\[ d = c - 1 \]
\[ a = 2 \]
\[ b = 1 \]
\[ c = a + b \]

\[ \langle \times, \times, 3, 2 \rangle \]

Execution Sequence

\[ \langle a, b, c, d \rangle \]

\[ \langle 1, 2, 3, ? \rangle \]

\[ \langle 1, 2, 3, 2 \rangle \]

\[ \langle 2, 1, 3, 2 \rangle \]

\[ \langle 2, 1, 3, 2 \rangle \]

\[ \langle 2, 1, 3, 2 \rangle \]

\[ \langle 2, 1, 3, 2 \rangle \]

\[ \langle 2, 1, 3, 2 \rangle \]

\[ \ldots \]
An Introduction to Constant Propagation

Summary Values

\[ n_1 \]
\[
\begin{aligned}
a &= 1 \\
b &= 2 \\
c &= a + b \\
\end{aligned}
\]
\[
\langle ?, ?, ?, ? \rangle
\]
\[
\langle 1, 2, 3, ? \rangle
\]

\[ n_2 \]
\[
\begin{aligned}
c &= a + b \\
d &= a \times b \\
\end{aligned}
\]
\[
\langle \times, \times, 3, 2 \rangle
\]

\[ n_3 \]
\[
\begin{aligned}
d &= c - 1 \\
a &= 2 \\
b &= 1 \\
c &= a + b \\
\end{aligned}
\]
\[
\langle 2, 1, 3, 2 \rangle
\]

\[
\langle 2, 1, 3, 2 \rangle
\]

\[
\langle 2, 1, 3, 2 \rangle
\]

\[
\langle 2, 1, 3, 2 \rangle
\]

\[
\langle 2, 1, 3, 2 \rangle
\]

Execution Sequence

\[ n_1 \]
\[
\langle a, b, c, d \rangle
\]
\[
\langle ?, ?, ?, ? \rangle
\]
\[
\langle 1, 2, 3, ? \rangle
\]
\[
\langle 1, 2, 3, 2 \rangle
\]
\[
\langle 2, 1, 3, 2 \rangle
\]

\[ n_2 \]
\[
\langle 1, 2, 3, ? \rangle
\]
\[
\langle 1, 2, 3, 2 \rangle
\]
\[
\langle 2, 1, 3, 2 \rangle
\]

\[ n_3 \]
\[
\langle 1, 2, 3, ? \rangle
\]
\[
\langle 2, 1, 3, 2 \rangle
\]
\[
\langle 2, 1, 3, 2 \rangle
\]
\[
\langle 2, 1, 3, 2 \rangle
\]

\[ \ldots \]
Issues

- Data Flow Values
- Desired Solutions
- Acceptable Operations
- Practical Algorithms
Issues

- Representation
- Lattice
  - Partial Order, Top, Bottom
- Data Flow Values
- Desired Solutions
- Acceptable Operations
- Practical Algorithms
Issues

Representation

Lattice
Partial Order, Top, Bottom

Merge
Commutativity, Associativity, Idempotence

Data Flow Values

Desired Solutions

Acceptable Operations

Practical Algorithms

Uday Khedker
IIT Bombay
Issues

- Representation
- Lattice
  - Partial Order, Top, Bottom
- Merge
  - Commutativity, Associativity, Idempotence
- Flow Functions
  - Monotonicity, Distributivity, k-Boundedness, Separability
Issues

- Representation
- Lattice
  - Partial Order, Top, Bottom
- Merge
  - Commutativity, Associativity, Idempotence
- Flow Functions
  - Monotonicity, Distributivity, k-Boundedness, Separability
- Data Flow Values
- Desired Solutions
- Practical Algorithms
- Existence
- Safety
- Precision

Data Flow Analysis

Algorithm

Uday Khedker
IIT Bombay
Issues

Data Flow Values
- Representation
- Lattice: Partial Order, Top, Bottom

Desired Solutions
- Existence
- Safety
- Precision

Acceptable Operations
- Merge: Commutativity, Associativity, Idempotence

Practical Algorithms
- Complexity
- Convergence

Flow Functions
- Initialisation
- Flow Functions: Monotonicity, Distributivity, k-Boundedness, Separability

Uday Khedker
IIT Bombay
Part 4

Data Flow Values
The Set of Data Flow Values

- Properties of the data flow values
- The notion of approximations
- Combining data flow values
The Set of Data Flow Values

- Can be looked upon as a partially ordered set
  - For available expressions analysis,
    - The powerset of the universal set of expressions
    - Partial order is the subset relation

Set View of the Lattice
The Set of Data Flow Values

- Can be looked upon as a partially ordered set
  For available expressions analysis,
  - The powerset of the universal set of expressions
  - Partial order is the subset relation

\[
\begin{align*}
\{e_1, e_2, e_3\} & \rightarrow \{e_1, e_2\} \rightarrow \{e_1\} \\
\{e_1, e_3\} & \rightarrow \{e_1\} \\
\{e_2, e_3\} & \rightarrow \{e_2\} \\
\end{align*}
\]

Set View of the Lattice
The Set of Data Flow Values

- Can be looked upon as a partially ordered set
  For available expressions analysis,
  - The powerset of the universal set of expressions
  - Partial order is the subset relation

Set View of the Lattice

Bit Vector View
An Aside on Lattices

Partially ordered sets

Partial order $\subseteq$ is reflexive, transitive, and antisymmetric
An Aside on Lattices

Partially ordered sets

Partial order \( \subseteq \) is reflexive, transitive, and antisymmetric

A lower bound of \( x, y \) is \( u \) s.t. \( u \subseteq x \) and \( u \subseteq y \)

An upper bound of \( x, y \) is \( u \) s.t. \( x \subseteq u \) and \( y \subseteq u \)
An Aside on Lattices

Partially ordered sets

\( \sqsubseteq \) is reflexive, transitive, and antisymmetric

Lattices

Every non-empty finite subset has a greatest lower bound (glb) and a least upper bound (lub)
Partially Ordered Sets

Set \{1, 2, 3, 4, 9\} with \sqsubseteq relation as ”divides” (i.e. \(a \sqsubseteq b \text{ iff } a \text{ divides } b\))
Set \{1, 2, 3, 4, 9\} with \subseteq relation as ”divides” (i.e. \(a \subseteq b\) iff \(a\) divides \(b\))
Set \( \{1, 2, 3, 4, 9\} \) with \( \sqsubseteq \) relation as "divides" (i.e. \( a \sqsubseteq b \) iff \( a \) divides \( b \))

Subsets \( \{4, 9\} \) and \( \{2, 3\} \) do not have an upper bound in the set.
Set \(\{1, 2, 3, 4, 9, 36\}\) with \(\sqsubseteq\) relation as ”divides” (i.e. \(a \sqsubseteq b\) iff \(a\) divides \(b\))
Complete Lattice

- Lattice: Every non-empty finite subset has a glb and a lub.
  Example: Lattice of integers under $\leq$ relation. All finite subsets have a glb and a lub. Infinite subsets do not have a glb or a lub.
Complete Lattice

- Lattice: Every non-empty finite subset has a glb and a lub.
  Example: Lattice of integers under $\leq$ relation. All finite subsets have a glb and a lub. Infinite subsets do not have a glb or a lub.

- Complete Lattice: Even empty and finite subsets have a glb and a lub.
  - Every finite lattice is complete.
  
  Example: Lattice of integers under $\leq$ relation with $\infty$ as $\top$ and $-\infty$ as $\bot$. Even infinite subsets have a glb and lub.
Complete Lattice

- **Lattice**: Every non-empty finite subset has a glb and a lub.
  Example: Lattice of integers under $\leq$ relation. All finite subsets have a glb and a lub. Infinite subsets do not have a glb or a lub.

- **Complete Lattice**: Even empty and finite subsets have a glb and a lub.
  - Every finite lattice is complete.
    Example: Lattice of integers under $\leq$ relation with $\infty$ as $\top$ and $-\infty$ as $\bot$. Even infinite subsets have a glb and lub.

- **Our discussion is restricted to complete lattices**.
  - Each lattice is finite, or
    - glb and lub exists for each subset even if the lattice is infinite.
Ascending and Descending Chains

- **Strictly ascending chain.** $x \sqsubseteq y \sqsubseteq \cdots \sqsubseteq z$

- **Strictly descending chain.** $x \sqsupseteq y \sqsupseteq \cdots \sqsupseteq z$

- If all strictly ascending and descending chains in $L$ are finite, then
  - $L$ has finite height, and
  - $L$ is complete.

- A complete lattice need not have finite height (i.e. strict chains may not be finite).

Example:
Lattice of integers under $\leq$ relation with $\infty$ as $\top$ and $-\infty$ as $\bot$.

We require all descending chains to be finite.
Operations on Lattices

- Meet (⊔) and Join (⊓)

Diagram:

```
    36
   /|
  /  |
/    |
1    2
   |
   |
   3  
   |
   9
```

Uday Khedker
IIT Bombay
Operations on Lattices

- Meet (⊓) and Join (⊔)
  - $x \sqcap y$ computes the glb of $x$ and $y$.
  - $z = x \sqcap y \Rightarrow z \sqsubseteq x \land z \sqsubseteq y$
Operations on Lattices

- Meet (\(\sqcap\)) and Join (\(\sqcup\))
  - \(x \sqcap y\) computes the glb of \(x\) and \(y\).
    \[z = x \sqcap y \Rightarrow z \sqsubseteq x \land z \sqsubseteq y\]
  - \(x \sqcup y\) computes the lub of \(x\) and \(y\).
    \[z = x \sqcup y \Rightarrow z \sqsupseteq x \land z \sqsupseteq y\]
Operations on Lattices

- Meet (\(\cap\)) and Join (\(\cup\))
  - \(x \cap y\) computes the glb of \(x\) and \(y\).
    \[z = x \cap y \Rightarrow z \subseteq x \land z \subseteq y\]
  - \(x \cup y\) computes the lub of \(x\) and \(y\).
    \[z = x \cup y \Rightarrow z \supseteq x \land z \supseteq y\]
  - \(\cap\) and \(\cup\) are commutative, associative, and idempotent.
Operations on Lattices

- **Meet (\(\cap\)) and Join (\(\sqcup\))**
  - \(x \cap y\) computes the glb of \(x\) and \(y\).
    \[z = x \cap y \Rightarrow z \subseteq x \land z \subseteq y\]
  - \(x \sqcup y\) computes the lub of \(x\) and \(y\).
    \[z = x \sqcup y \Rightarrow z \supseteq x \land z \supseteq y\]
  - \(\cap\) and \(\sqcup\) are commutative, associative, and idempotent.

- **Top (\(\top\)) and Bottom (\(\bot\)) elements**
  
  \[
  \forall x \in L, \ x \cap \top = x \\
  \forall x \in L, \ x \sqcup \top = \top \\
  \forall x \in L, \ x \cap \bot = \bot \\
  \forall x \in L, \ x \sqcup \bot = x
  \]
Operations on Lattices

- **Meet (\(\sqcap\)) and Join (\(\sqcup\))**
  - \(x \sqcap y\) computes the glb of \(x\) and \(y\).
    
    \[z = x \sqcap y \Rightarrow z \subseteq x \land z \subseteq y\]
  - \(x \sqcup y\) computes the lub of \(x\) and \(y\).
    
    \[z = x \sqcup y \Rightarrow z \supseteq x \land z \supseteq y\]
  - \(\sqcap\) and \(\sqcup\) are commutative, associative, and idempotent.

- **Top (\(\top\)) and Bottom (\(\bot\)) elements**
  
  \[
  \forall x \in L, \ x \sqcap \top = x \\
  \forall x \in L, \ x \sqcup \top = \top \\
  \forall x \in L, \ x \sqcap \bot = \bot \\
  \forall x \in L, \ x \sqcup \bot = x \\
  \]

Greatest common divisor (or highest common factor) in the lattice

\[x \sqcap y = gcd'(x, y)\]
Operations on Lattices

- **Meet** ($\sqcap$) and **Join** ($\sqcup$)
  - $x \sqcap y$ computes the glb of $x$ and $y$.
    
    $z = x \sqcap y \Rightarrow z \subseteq x \land z \subseteq y$
  
  - $x \sqcup y$ computes the lub of $x$ and $y$.
    
    $z = x \sqcup y \Rightarrow z \supseteq x \land z \supseteq y$
  
  - $\sqcap$ and $\sqcup$ are commutative, associative, and idempotent.

- **Top** ($\top$) and **Bottom** ($\bot$) elements

  $\forall x \in L, x \sqcap \top = x$

  $\forall x \in L, x \sqcup \top = \top$

  $\forall x \in L, x \sqcap \bot = \bot$

  $\forall x \in L, x \sqcup \bot = x$

- **Greatest common divisor (or highest common factor)** in the lattice

  - $x \sqcap y = gcd'(x, y)$
  - $x \sqcup y = lcm'(x, y)$

- **Lowest common multiple** in the lattice
Cartesian Product of Lattice

\[ \langle L_N, \subseteq_N, \cap_N, \cup_N \rangle \times \langle L_A, \subseteq_A, \cap_A, \cup_A \rangle = \langle L_A \times L_N, \subseteq_{A \times N}, \cap_{A \times N}, \cup_{A \times N} \rangle \]
Cartesian Product of Lattice

\[ \langle L, \sqsubseteq_N, \sqcap_N, \sqcup_N \rangle \times \langle L, \sqsubseteq_A, \sqcap_A, \sqcup_A \rangle = \langle 1, a \rangle, \langle 2, a \rangle, \langle 3, a \rangle, \langle 4, a \rangle \]
Cartesian Product of Lattice

\[ \langle L_N, \sqsubseteq_N, \sqcap_N, \sqcup_N \rangle \times \langle L_A, \sqsubseteq_A, \sqcap_A, \sqcup_A \rangle = \langle 1, a \rangle \langle 2, a \rangle \langle 3, a \rangle \langle 4, a \rangle \langle 1, b \rangle \langle 2, b \rangle \langle 3, b \rangle \langle 4, b \rangle \]
Cartesian Product of Lattice

\[
\langle L, \sqsubseteq, \sqcap, \sqcup \rangle \times \langle L, \sqsubseteq, \sqcap, \sqcup \rangle = \langle 1, a \rangle \langle 2, a \rangle \langle 3, a \rangle \langle 4, a \rangle \langle 2, b \rangle \langle 3, b \rangle \langle 4, b \rangle
\]
Cartesian Product of Lattice

\[
\langle L_N, \sqsubseteq_N, \sqcap_N, \sqcup_N \rangle 
\times 
\langle L_A, \sqsubseteq_A, \sqcap_A, \sqcup_A \rangle 
= 
\langle 1, a \rangle 
\langle 2, a \rangle 
\langle 3, a \rangle 
\langle 1, b \rangle 
\langle 2, b \rangle 
\langle 3, b \rangle 
\langle 4, b \rangle
\]
Cartesian Product of Lattice

\[ \langle L_N, \sqsubseteq_N, \sqcap_N, \sqcup_N \rangle \times \langle L_A, \sqsubseteq_A, \sqcap_A, \sqcup_A \rangle = \langle 1, a \rangle \langle 2, a \rangle \langle 3, a \rangle \langle 4, a \rangle \langle 1, b \rangle \langle 2, b \rangle \langle 3, b \rangle \langle 4, b \rangle \]

\[ \langle 1, a \rangle \quad \langle 2, a \rangle \quad \langle 3, a \rangle \quad \langle 1, b \rangle \]

\[ \langle 2, b \rangle \quad \langle 3, b \rangle \quad \langle 4, b \rangle \]
Cartesian Product of Lattice

\[ \langle L_N, \sqsubseteq_N, \sqcap_N, \sqcup_N \rangle \times \langle L_A, \sqsubseteq_A, \sqcap_A, \sqcup_A \rangle = \langle 1, a \rangle \langle 2, a \rangle \langle 3, a \rangle \langle 4, a \rangle \langle 1, b \rangle \langle 2, b \rangle \langle 3, b \rangle \langle 4, b \rangle \]
Cartesian Product of Lattice

\[ \langle L_N, \sqsubseteq_N, \bigwedge_N, \bigvee_N \rangle \times \langle L_A, \sqsubseteq_A, \bigwedge_A, \bigvee_A \rangle = \langle L_C, \sqsubseteq_C, \bigwedge_C, \bigvee_C \rangle \]
Cartesian Product of Lattice

\[ \langle L_N, \sqsubseteq_N, \cap N, \cup N \rangle \times \langle L_A, \sqsubseteq A, \cap A, \cup A \rangle = \langle L_C, \sqsubseteq C, \cap C, \cup C \rangle \]

\[ \langle x_1, y_1 \rangle \sqsubseteq C \langle x_2, y_2 \rangle \iff x_1 \sqsubseteq_N x_2 \land y_1 \sqsubseteq_A y_2 \]

\[ \langle x_1, y_1 \rangle \cap C \langle x_2, y_2 \rangle = \langle x_1 \cap_N x_2, y_1 \cap_A y_2 \rangle \]

\[ \langle x_1, y_1 \rangle \cup C \langle x_2, y_2 \rangle = \langle x_1 \cup_N x_2, y_1 \cup_A y_2 \rangle \]
The Concept of Approximation

- $x$ approximates $y$ iff $x$ can be used in place of $y$ without causing any problems.

- Validity of approximation is context specific
  - $x$ may be approximated by $y$ in one context and by $z$ in another
    - Earnings: Rs. 1050 can be safely approximated by Rs. 1000.
    - Expenses: Rs. 1050 can be safely approximated by Rs. 1100.
Two Important Objectives in Data Flow Analysis

- The discovered data flow information should be
  - *Exhaustive*. No optimization opportunity should be missed.
  - *Safe*. Optimizations which do not preserve semantics should not be enabled.
Two Important Objectives in Data Flow Analysis

• The discovered data flow information should be
  
  ▶ **Exhaustive.** No optimization opportunity should be missed.
  
  ▶ **Safe.** Optimizations which do not preserve semantics should not be enabled.

• Conservative approximations of these objectives are allowed
Two Important Objectives in Data Flow Analysis

- The discovered data flow information should be
  - **Exhaustive**. No optimization opportunity should be missed.
  - **Safe**. Optimizations which do not preserve semantics should not be enabled.
- Conservative approximations of these objectives are allowed
- The intended use of data flow information (≡ context) determines validity of approximations
Context Determines the Validity of Approximations

May prohibit correct optimization
May enable wrong optimization

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Application</th>
<th>Safe Approximation</th>
<th>Exhaustive Approximation</th>
</tr>
</thead>
</table>
Context Determines the Validity of Approximations

May prohibit correct optimization
May enable wrong optimization

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Application</th>
<th>Safe Approximation</th>
<th>Exhaustive Approximation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Live variables</td>
<td>Dead code elimination</td>
<td>A dead variable is erroneously considered live</td>
<td>A live variable is erroneously considered dead</td>
</tr>
</tbody>
</table>
## Context Determines the Validity of Approximations

May prohibit correct optimization

May enable wrong optimization

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Application</th>
<th>Safe Approximation</th>
<th>Exhaustive Approximation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Live variables</td>
<td>Dead code elimination</td>
<td>A dead variable is erroneously considered live</td>
<td>A live variable is erroneously considered dead</td>
</tr>
<tr>
<td>Available expressions</td>
<td>Common subexpression elimination</td>
<td>An available expression is erroneously considered non-available</td>
<td>A non-available expression is erroneously considered available</td>
</tr>
</tbody>
</table>
Context Determines the Validity of Approximations

May prohibit correct optimization

May enable wrong optimization

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Application</th>
<th>Safe Approximation</th>
<th>Exhaustive Approximation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Live variables</td>
<td>Dead code elimination</td>
<td>A dead variable is erroneously considered live</td>
<td>A live variable is erroneously considered dead</td>
</tr>
<tr>
<td>Available expressions</td>
<td>Common subexpression elimination</td>
<td>An available expression is erroneously considered non-available</td>
<td>A non-available expression is erroneously considered available</td>
</tr>
</tbody>
</table>

Spurious Inclusion

Spurious Exclusion
Partial Order Captures Approximation

- $\sqsubseteq$ captures valid approximations for safety

$$x \sqsubseteq y \Rightarrow x \text{ is weaker than } y$$

- The data flow information represented by $x$ can be safely used in place of the data flow information represented by $y$
- It may be imprecise, though.
Partial Order Captures Approximation

• $\sqsubseteq$ captures valid approximations for safety
  \[ x \sqsubseteq y \implies x \text{ is weaker than } y \]
  ▶ The data flow information represented by $x$ can be safely used in place of the data flow information represented by $y$
  ▶ It may be imprecise, though.

• $\sqsupseteq$ captures valid approximations for exhaustiveness
  \[ x \sqsupseteq y \implies x \text{ is stronger than } y \]
  ▶ The data flow information represented by $x$ contains every value contained in the data flow information represented by $y$
  ▶ It may be unsafe, though.
Partial Order Captures Approximation

- $\sqsubseteq$ captures valid approximations for safety

  $x \sqsubseteq y \implies x$ is weaker than $y$

  - The data flow information represented by $x$ can be safely used in place of the data flow information represented by $y$
  - It may be imprecise, though.

- $\sqsupseteq$ captures valid approximations for exhaustiveness

  $x \sqsupseteq y \implies x$ is stronger than $y$

  - The data flow information represented by $x$ contains every value contained in the data flow information represented by $y$
  - It may be unsafe, though.

*We want most exhaustive information which is also safe.*
Most Approximate Values in a Complete Lattice

- **Top.** \( \forall x \in L, \ x \sqsubseteq \top \). The most exhaustive value.

- **Bottom.** \( \forall x \in L, \ \bot \sqsubseteq x \). The safest value.
Most Approximate Values in a Complete Lattice

- **Top.** ∀x ∈ L, x ⊑ ⊤. The most exhaustive value.
  - Using ⊤ in place of any data flow value will never miss out (or rule out) any possible value.

- **Bottom.** ∀x ∈ L, ⊥ ⊑ x. The safest value.
Most Approximate Values in a Complete Lattice

- **Top.** $\forall x \in L, x \sqsubseteq \top$. The most exhaustive value.
  
  - Using $\top$ in place of any data flow value will never miss out (or rule out) any possible value.
  
  - The consequences may be semantically unsafe, or incorrect.

- **Bottom.** $\forall x \in L, \bot \sqsubseteq x$. The safest value.
Most Approximate Values in a Complete Lattice

- **Top.** \( \forall x \in L, \ x \sqsubseteq \top \). The most exhaustive value.
  
  - Using \( \top \) in place of any data flow value will never miss out (or rule out) any possible value.
  
  - The consequences may be semantically unsafe, or incorrect.

- **Bottom.** \( \forall x \in L, \ \bot \sqsubseteq x \). The safest value.
  
  - Using \( \bot \) in place of any data flow value will never be unsafe, or incorrect.
Most Approximate Values in a Complete Lattice

- **Top.** \( \forall x \in L, \ x \sqsubseteq \top \). The most exhaustive value.
  - Using \( \top \) in place of any data flow value will never miss out (or rule out) any possible value.
  - The consequences may be semantically *unsafe*, or *incorrect*.

- **Bottom.** \( \forall x \in L, \ \bot \sqsubseteq x \). The safest value.
  - Using \( \bot \) in place of any data flow value will never be *unsafe*, or *incorrect*.
  - The consequences may be *undefined* or *useless* because this replacement might miss out valid values.

Uday Khedker
Most Approximate Values in a Complete Lattice

- **Top.** $\forall x \in L$, $x \sqsubseteq \top$. The most exhaustive value.
  - Using $\top$ in place of any data flow value will never miss out (or rule out) any possible value.
  - The consequences may be semantically *unsafe*, or *incorrect*.

- **Bottom.** $\forall x \in L$, $\bot \sqsubseteq x$. The safest value.
  - Using $\bot$ in place of any data flow value will never be *unsafe*, or *incorrect*.
  - The consequences may be *undefined* or *useless* because this replacement might miss out valid values.

*Appropriate orientation chosen by design.*
Setting Up Lattices

Available Expressions Analysis

\[
\begin{align*}
\{e_1, e_2, e_3\} & \quad \{e_1, e_2\} \quad \{e_1, e_3\} \quad \{e_2, e_3\} \\
\{e_1\} & \quad \{e_2\} \quad \{e_3\} \\
\emptyset &
\end{align*}
\]

\[\sqsubseteq \text{ is } \subseteq\]

\[\sqcap \text{ is } \cap\]

Live Variables Analysis

\[
\begin{align*}
\{v_1\} & \quad \{v_2\} \quad \{v_3\} \\
\{v_1, v_2\} & \quad \{v_1, v_3\} \quad \{v_2, v_3\} \\
\{v_1, v_2, v_3\} &
\end{align*}
\]

\[\sqsubseteq \text{ is } \supseteq\]

\[\sqcap \text{ is } \cup\]
Partial Order Relation

- Reflexive: \( x \sqsubseteq x \)
- Transitive: \( x \sqsubseteq y, y \sqsubseteq z \Rightarrow x \sqsubseteq z \)
- Antisymmetric: \( x \sqsubseteq y, y \sqsubseteq x \Leftrightarrow x = y \)
Partial Order Relation

**Reflexive**

\[ x \sqsubseteq x \]

\( x \) can be safely used in place of \( x \)

**Transitive**

\[ x \sqsubseteq y, y \sqsubseteq z \quad \Rightarrow \quad x \sqsubseteq z \]

If \( x \) can be safely used in place of \( y \) and \( y \) can be safely used in place of \( z \), then \( x \) can be safely used in place of \( z \)

**Antisymmetric**

\[ x \sqsubseteq y, y \sqsubseteq x \quad \iff \quad x = y \]

If \( x \) can be safely used in place of \( y \) and \( y \) can be safely used in place of \( x \), then \( x \) must be same as \( y \)
Merging Information

• $x \sqcap y$ computes the greatest lower bound of $x$ and $y$ i.e. largest $z$ such that $z \sqsubseteq x$ and $z \sqsubseteq y$

The largest safe approximation of combining data flow information $x$ and $y$
Merging Information

- $x \sqcap y$ computes the \textit{greatest lower bound} of $x$ and $y$ i.e. largest $z$ such that $z \sqsubseteq x$ and $z \sqsubseteq y$

  The largest safe approximation of combining data flow information $x$ and $y$

- Commutative  $x \sqcap y = y \sqcap x$

- Associative  $x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z$

- Idempotent  $x \sqcap x = x$
Merging Information

- \( x \sqcap y \) computes the *greatest lower bound* of \( x \) and \( y \) i.e. largest \( z \) such that \( z \sqsubseteq x \) and \( z \sqsubseteq y \)

The largest safe approximation of combining data flow information \( x \) and \( y \)

- **Commutative** \( x \sqcap y = y \sqcap x \) The order in which the data flow information is merged, does not matter

- **Associative** \( x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z \) Allow n-ary merging without any restriction on the order

- **Idempotent** \( x \sqcap x = x \) No loss of information if \( x \) is merged with itself
Merging Information

- $x \sqcap y$ computes the greatest lower bound of $x$ and $y$ i.e. largest $z$ such that $z \sqsubseteq x$ and $z \sqsubseteq y$

The largest safe approximation of combining data flow information $x$ and $y$

- **Commutative** $x \sqcap y = y \sqcap x$ The order in which the data flow information is merged, does not matter

- **Associative** $x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z$ Allow n-ary merging without any restriction on the order

- **Idempotent** $x \sqcap x = x$ No loss of information if $x$ is merged with itself

- $x \sqcap \top = x$ (ensures exhaustiveness)

- $x \sqcap \bot = \bot$ (ensures safety)
More on Lattices in Data Flow Analysis

\[ L = \text{Lattice for all expressions} \]
\[ \hat{L} = \text{Lattice for a single expression} \]

(Expressions \( e \) is available)

\[ 1 \text{ or } \{ e \} \]
\[ 0 \text{ or } \emptyset \]

(Expressions \( e \) is not available)
More on Lattices in Data Flow Analysis

$L = \text{Lattice for all expressions}$

$\hat{L} = \text{Lattice for a single expression}$

<table>
<thead>
<tr>
<th>111</th>
<th>110</th>
<th>101</th>
<th>011</th>
</tr>
</thead>
<tbody>
<tr>
<td>110</td>
<td>101</td>
<td>011</td>
<td>001</td>
</tr>
<tr>
<td>100</td>
<td>010</td>
<td>001</td>
<td></td>
</tr>
</tbody>
</table>

(Expression e is available)

1 or \(\{e\}\)

0 or \(\emptyset\)

(Expression e is not available)

Cartesian products if sets are used, vectors (or tuples) if bit are used.

- \(L = \hat{L} \times \hat{L} \times \hat{L}\) and \(x = \langle \hat{x}_1, \hat{x}_2, \hat{x}_3 \rangle \in L\) where \(\hat{x}_i \in \hat{L}\)
- \(\subseteq = \hat{\subseteq} \times \hat{\subseteq} \times \hat{\subseteq}\) and \(\square = \hat{\square} \times \hat{\square} \times \hat{\square}\)
- \(\top = \hat{\top} \times \hat{\top} \times \hat{\top}\) and \(\bot = \hat{\bot} \times \hat{\bot} \times \hat{\bot}\)
Component Lattice for Data Flow Information Represented By Bit Vectors

\[
\begin{array}{c}
\hat{\top} \\
1 \\
\cap \\
0 \\
\hat{\bot}
\end{array}
\quad
\begin{array}{c}
\hat{\top} \\
0 \\
\cup \\
1 \\
\hat{\bot}
\end{array}
\]

\(\cap\) is \(\cap\) or Boolean AND

\(\cup\) is \(\cup\) or Boolean OR
Component Lattice for Integer Constant Propagation

(\text{undef or ud})

\begin{center}
\begin{tikzpicture}

\node (a) at (0,0) {$-\infty$};
\node (b) at (1,0) {$\cdots$};
\node (c) at (2,0) {$-2$};
\node (d) at (3,0) {$-1$};
\node (e) at (4,0) {$0$};
\node (f) at (5,0) {$1$};
\node (g) at (6,0) {$2$};
\node (h) at (7,0) {$\cdots$};
\node (i) at (8,0) {$\infty$};
\node (j) at (4,1) {$\text{nonconst or nc}$};
\node (k) at (4,-1) {$\text{null or ud}$};

\draw (a) -- (b);
\draw (b) -- (c);
\draw (c) -- (d);
\draw (d) -- (e);
\draw (e) -- (f);
\draw (f) -- (g);
\draw (g) -- (h);
\draw (h) -- (i);
\draw (i) -- (j);
\draw (j) -- (k);
\draw (k) -- (a);
\end{tikzpicture}
\end{center}

- Overall lattice $L$ is the product of $\hat{L}$ for all variables.
- $\sqcap$ and $\hat{\sqcap}$ get defined by $\sqsubseteq$ and $\hat{\sqsubseteq}$.

<table>
<thead>
<tr>
<th>$\hat{\sqcap}$</th>
<th>$\langle a, ud \rangle$</th>
<th>$\langle a, nc \rangle$</th>
<th>$\langle a, c_1 \rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle a, ud \rangle$</td>
<td>$\langle a, ud \rangle$</td>
<td>$\langle a, nc \rangle$</td>
<td>$\langle a, c_1 \rangle$</td>
</tr>
<tr>
<td>$\langle a, nc \rangle$</td>
<td>$\langle a, nc \rangle$</td>
<td>$\langle a, nc \rangle$</td>
<td>$\langle a, nc \rangle$</td>
</tr>
<tr>
<td>$\langle a, c_2 \rangle$</td>
<td>$\langle a, c_2 \rangle$</td>
<td>$\langle a, nc \rangle$</td>
<td>If $c_1 = c_2$ then $\langle a, c_1 \rangle$ else $\langle a, nc \rangle$</td>
</tr>
</tbody>
</table>
Part 5

Flow Functions
Flow Functions

- Computing data flow values from local effects
- Properties of flow functions
The Set of Flow Functions

- $F$ is the set of functions $f : L \mapsto L$ such that
  - $F$ contains an identity function
    To model “empty” statements, i.e. statements which do not influence the data flow information
  - $F$ is closed under composition
    Cumulative effect of statements should generate data flow information from the same set.

- Properties of $f$
  - Monotonicity and Distributivity
  - Loop Closure Boundedness
Flow Functions in Bit Vector Data Flow Frameworks

- Bit Vector Frameworks: Available Expressions Analysis, Reaching Definitions Analysis, Live variable Analysis, Very Busy Expressions Analysis, Partial Redundancy Elimination etc.
  - All functions can be defined in terms of constant Gen and Kill
    \[ f(x) = \text{Gen} \cup (x - \text{Kill}) \]
  - Lattices are powersets with partial orders as \( \subseteq \) or \( \supseteq \) relations
  - Information is merged using \( \cap \) or \( \cup \)
Flow Functions in Bit Vector Data Flow Frameworks

- Bit Vector Frameworks: Available Expressions Analysis, Reaching Definitions Analysis Live variable Analysis, Very Busy Expressions Analysis, Partial Redundancy Elimination etc.
  - All functions can be defined in terms of constant Gen and Kill
    \[ f(x) = \text{Gen} \cup (x - \text{Kill}) \]
  - Lattices are powersets with partial orders as \( \subseteq \) or \( \supseteq \) relations
  - Information is merged using \( \cap \) or \( \cup \)

- Flow functions in Faint Variables Analysis, Pointer Analyses, Constant Propagation, Possibly Uninitialized Variables cannot be expressed using constant Gen and Kill.

Local context alone is not sufficient to describe the effect of statements fully.
Monotonicity of Flow Functions

- Partial order is preserved: If $x$ can be safely used in place of $y$ then $f(x)$ can be safely used in place of $f(y)$. 

\[ x \rightarrow f(x) \quad y \rightarrow f(y) \]
Monotonicity of Flow Functions

- Partial order is preserved: If $x$ can be safely used in place of $y$ then $f(x)$ can be safely used in place of $f(y)$
Monotonicity of Flow Functions

- Partial order is preserved: If $x$ can be safely used in place of $y$ then $f(x)$ can be safely used in place of $f(y)$

\[ \forall x, y \in L, x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y) \]
Monotonicity of Flow Functions

- Partial order is preserved: If \( x \) can be safely used in place of \( y \) then \( f(x) \) can be safely used in place of \( f(y) \)

\[
\forall x, y \in L, x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)
\]

- Alternative definition

\[
\forall x, y \in L, f(x \sqcap y) \sqsubseteq f(x) \sqcap f(y)
\]
Monotonicity of Flow Functions

• Partial order is preserved: If \( x \) can be safely used in place of \( y \) then \( f(x) \) can be safely used in place of \( f(y) \)

\[
\forall x, y \in L, x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)
\]

• Alternative definition

\[
\forall x, y \in L, f(x \cap y) \sqsubseteq f(x) \cap f(y)
\]

• Merging at intermediate points in shared segments of paths is safe (However, it may lead to imprecision).
Distributivity of Flow Functions

- Merging distributes over function application

\[ f(x) \sqcap f(y) \]
Distributivity of Flow Functions

- Merging distributes over function application

\[ f(x \sqcap y) = f(x) \sqcap f(y) \]
Distributivity of Flow Functions

- Merging distributes over function application

\[ \forall x, y \in L, x \sqsubseteq y \Rightarrow f(x \sqcap y) = f(x) \sqcap f(y) \]
Distributivity of Flow Functions

- Merging distributes over function application

\[ \forall x, y \in L, x \sqsubseteq y \Rightarrow f(x \cap y) = f(x) \cap f(y) \]

- Merging at intermediate points in shared segments of paths does not lead to imprecision.
Monotonicity and Distributivity
Monotonicity and Distributivity
Monotonicity and Distributivity
Monotonicity and Distributivity

\[ \top \quad \bot \quad L \]

\[ \downarrow \quad \downarrow \quad L \]

\[ \quad \rightarrow \quad \rightarrow \]

\[ L \quad L \]
Monotonicity and Distributivity
Monotonicity and Distributivity

Monotonic and Distributive
Monotonicity and Distributivity

Monotonic but not Distributive

$L$
Distributivity of Bit Vector Frameworks

\[ f(x) = \text{Gen} \cup (x - \text{Kill}) \]
\[ f(y) = \text{Gen} \cup (y - \text{Kill}) \]

\[ f(x \cup y) = \text{Gen} \cup ((x \cup y) - \text{Kill}) \]
\[ = \text{Gen} \cup ((x - \text{Kill}) \cup (y - \text{Kill})) \]
\[ = (\text{Gen} \cup (x - \text{Kill}) \cup \text{Gen} \cup (y - \text{Kill})) \]
\[ = f(x) \cup f(y) \]

\[ f(x \cap y) = \text{Gen} \cup ((x \cap y) - \text{Kill}) \]
\[ = \text{Gen} \cup ((x - \text{Kill}) \cap (y - \text{Kill})) \]
\[ = (\text{Gen} \cup (x - \text{Kill}) \cap \text{Gen} \cup (y - \text{Kill})) \]
\[ = f(x) \cap f(y) \]
Non-Distributivity of Constant Propagation

\[ a = 1 \]
\[ b = 2 \]
\[ c = a + b \]

\[ c = a + b \]
\[ d = a \times b \]

\[ d = c - 1 \]
\[ a = 2 \]
\[ b = 1 \]
\[ c = a + b \]
Non-Distributivity of Constant Propagation

- $x = \langle 1, 2, 3, ? \rangle$ (Along $Out_{n_1} \rightarrow In_{n_2}$)

Diagram:

- $n_1$:
  - $a = 1$
  - $b = 2$
  - $c = a + b$
  - $a = 1, b = 2$

- $n_2$:
  - $c = a + b$
  - $d = a \times b$

- $n_3$:
  - $d = c - 1$
  - $a = 2$
  - $b = 1$
  - $c = a + b$
Non-Distributivity of Constant Propagation

- \( x = \langle 1, 2, 3, ? \rangle \) (Along \( Out_{n_1} \rightarrow In_{n_2} \))
- \( y = \langle 2, 1, 3, 2 \rangle \) (Along \( Out_{n_3} \rightarrow In_{n_2} \))
Non-Distributivity of Constant Propagation

- \( x = \langle 1, 2, 3, ? \rangle \) (Along \( Out_{n_1} \rightarrow In_{n_2} \))
- \( y = \langle 2, 1, 3, 2 \rangle \) (Along \( Out_{n_3} \rightarrow In_{n_2} \))
- Function application before merging
  \[
  f(x) \sqcap f(y) = f(\langle 1, 2, 3, ? \rangle) \sqcap f(\langle 2, 1, 3, 2 \rangle) \\
  = \langle 1, 2, 3, 2 \rangle \sqcap \langle 2, 1, 3, 2 \rangle \\
  = \langle \hat{\perp}, \hat{\perp}, 3, 2 \rangle
  \]
Non-Distributivity of Constant Propagation

- $x = \langle 1, 2, 3, ? \rangle$ (Along $Out_{n_1} \rightarrow Ln_{n_2}$)
- $y = \langle 2, 1, 3, 2 \rangle$ (Along $Out_{n_3} \rightarrow Ln_{n_2}$)
- Function application before merging
  
  
  $$f(x) \cap f(y) = f(\langle 1, 2, 3, ? \rangle) \cap f(\langle 2, 1, 3, 2 \rangle)$$
  
  $$= \langle 1, 2, 3, 2 \rangle \cap \langle 2, 1, 3, 2 \rangle$$
  
  $$= \langle \perp, \perp, 3, 2 \rangle$$

- Function application after merging
  
  $$f(x \cap y) = f(\langle 1, 2, 3, ? \rangle \cap \langle 2, 1, 3, 2 \rangle)$$
  
  $$= f(\langle \perp, \perp, 3, 2 \rangle)$$
  
  $$= \langle \perp, \perp, \perp, \perp \rangle$$
Non-Distributivity of Constant Propagation

- $x = \langle 1, 2, 3, ? \rangle$ (Along $Out_{n_1} \rightarrow In_{n_2}$)
- $y = \langle 2, 1, 3, 2 \rangle$ (Along $Out_{n_3} \rightarrow In_{n_2}$)
- Function application before merging

$$f(x) \sqcap f(y) = f(\langle 1, 2, 3, ? \rangle) \sqcap f(\langle 2, 1, 3, 2 \rangle)$$
$$= \langle 1, 2, 3, 2 \rangle \sqcap \langle 2, 1, 3, 2 \rangle$$
$$= \langle \perp, \perp, 3, 2 \rangle$$

- Function application after merging

$$f(x \sqcap y) = f(\langle 1, 2, 3, ? \rangle \sqcap \langle 2, 1, 3, 2 \rangle)$$
$$= f(\langle \perp, \perp, 3, 2 \rangle)$$
$$= \langle \perp, \perp, \perp, \perp \rangle$$

- $f(x \sqcap y) \sqcup f(x) \sqcap f(y)$
Why is Constant Propagation Non-Distributive?

\[ a = 1 \\
\quad b = 2 \]

\[ a = 2 \\
\quad b = 1 \]

\[ c = a + b \]
Why is Constant Propagation Non-Distributive?

Possible combinations due to merging

\[ a = 1 \quad a = 2 \quad b = 1 \quad b = 2 \]
Why is Constant Propagation Non-Distributive?

Possible combinations due to merging

\[
\begin{aligned}
    a &= 1 \\
    b &= 2 \\
    c &= a + b
\end{aligned}
\quad
\begin{aligned}
    a &= 2 \\
    b &= 1 \\
    c &= a + b
\end{aligned}
\]

\[
\begin{aligned}
    a &= 1 \\
    b &= 2 \\
    c &= a + b
\end{aligned}
\]

\[
\begin{aligned}
    a &= 1 \\
    b &= 1 \\
    c &= a + b
\end{aligned}
\]

• Correct combination.

Uday Khedker

IIT Bombay
Why is Constant Propagation Non-Distributive?

Possible combinations due to merging

\[ a = 1 \quad a = 2 \quad b = 1 \quad b = 2 \]

\[ c = a + b = 3 \]

- Correct combination.
Why is Constant Propagation Non-Distributive?

Possible combinations due to merging

\[ a = 1 \quad a = 2 \quad b = 1 \quad b = 2 \]

\[ c = a + b = 2 \]

- Wrong combination.
- Mutually exclusive information.
- No execution path along which this information holds.

Uday Khedker
Why is Constant Propagation Non-Distributive?

Possible combinations due to merging

\[
\begin{align*}
    a &= 1 \\
    b &= 2 \\
    c &= a + b = 3
\end{align*}
\]

\[
\begin{align*}
    a &= 2 \\
    b &= 1 \\
    c &= a + b = 3
\end{align*}
\]

Wrong combination.
Mutually exclusive information.
No execution path along which this information holds.
Part 6

Solutions of Data Flow Analysis
Solutions of Data Flow Analysis

- Characterizing solutions
  - Desirable solutions and computable solutions
- Existence and computability
Solutions of Data Flow Analysis

- An assignment \( A \) associates data flow values with program points. 
  \[ A \sqsubseteq B \text{ if for all program points } p, \ A(p) \sqsubseteq B(p) \]

- Performing data flow analysis

  **Given**
  
  - A set of flow functions, a lattice, and merge operation
  - A program flow graph with a mapping from nodes to flow functions

  **Find out**
  
  - An assignment \( A \) which is as exhaustive as possible and is safe
Meet Over Paths (MoP) Assignment

- The largest safe approximation of the information reaching a program point along all information flow paths.

\[
MoP(p) = \bigcap_{\rho \in \text{Paths}(p)} f_{\rho}(\text{BoundaryInfo})
\]

- \( f_{\rho} \) represents the compositions of flow functions along \( \rho \).
- \( \text{BoundaryInfo} \) refers to the relevant information from the calling context.
- All execution paths are considered potentially executable by ignoring the results of conditionals.
Meet Over Paths (MoP) Assignment

- The largest safe approximation of the information reaching a program point along all information flow paths.

\[ MoP(p) = \bigsqcap_{\rho \in Paths(p)} f_\rho(BoundaryInfo) \]

- \( f_\rho \) represents the compositions of flow functions along \( \rho \).
- \( BoundaryInfo \) refers to the relevant information from the calling context.
- All execution paths are considered potentially executable by ignoring the results of conditionals.

- Any \( Info(p) \subseteq MoP(p) \) is safe.
Existence of an MoP Assignment

\[ MoP(p) = \prod_{\rho \in \text{Paths}(p)} f_{\rho}(\text{BoundaryInfo}) \]

- If all paths reaching \( p \) are acyclic, then existence of solution trivially follows from the definition of the function space.
- If cyclic paths also reach \( p \), then there are an infinite number of unbounded paths.  
  \( \Rightarrow \) Need to define loop closures.
Loop Closures of Flow Functions

Paths Terminating at $p_2$ | Data Flow Value
--- | ---
$p_1, p_2$ | $x$
$p_1, p_2, p_3, p_2$ | $f(x)$
$p_1, p_2, p_3, p_2, p_3, p_2$ | $f(f(x)) = f^2(x)$
$p_1, p_2, p_3, p_2, p_3, p_2, p_3, p_2$ | $f(f(f(x))) = f^3(x)$
... | ...
### Loop Closures of Flow Functions

<table>
<thead>
<tr>
<th>Paths Terminating at $p_2$</th>
<th>Data Flow Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1, p_2$</td>
<td>$x$</td>
</tr>
<tr>
<td>$p_1, p_2, p_3, p_2$</td>
<td>$f(x)$</td>
</tr>
<tr>
<td>$p_1, p_2, p_3, p_2, p_3, p_2$</td>
<td>$f(f(x)) = f^2(x)$</td>
</tr>
<tr>
<td>$p_1, p_2, p_3, p_2, p_3, p_2$</td>
<td>$f(f(f(x))) = f^3(x)$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- For static analysis we need to summarize the value at $p_2$ by a value which is safe after any iteration.

\[
f^*(x) = x \sqcap f(x) \sqcap f^2(x) \sqcap f^3(x) \sqcap f^4(x) \sqcap \ldots\]
Loop Closures of Flow Functions

- For static analysis we need to summarize the value at $p_2$ by a value which is safe after any iteration.

$$f^*(x) = x \sqcap f(x) \sqcap f^2(x) \sqcap f^3(x) \sqcap f^4(x) \sqcap \ldots$$

- $f^*$ is called the loop closure of $f$. 

<table>
<thead>
<tr>
<th>Paths Terminating at $p_2$</th>
<th>Data Flow Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1, p_2$</td>
<td>$x$</td>
</tr>
<tr>
<td>$p_1, p_2, p_3, p_2$</td>
<td>$f(x)$</td>
</tr>
<tr>
<td>$p_1, p_2, p_3, p_2, p_3, p_2$</td>
<td>$f(f(x)) = f^2(x)$</td>
</tr>
<tr>
<td>$p_1, p_2, p_3, p_2, p_3, p_2$</td>
<td>$f(f(f(x))) = f^3(x)$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Bounded Loop Closures

- The loop closure of \( f \) is bounded if there exists a \( k \) such that

\[
\forall x \in L, \quad f^*(x) = x \cap f(x) \cap f^2(x) \cap \ldots \cap f^{k-1}(x)
\]
Bounded Loop Closures

- The loop closure of $f$ is bounded if there exists a $k$ such that

$$\forall x \in L, \ f^*(x) = x \sqcap f(x) \sqcap f^2(x) \sqcap \ldots \sqcap f^{k-1}(x)$$

- If all descending chains are finite, then loop closures are bounded.

The lattice may have infinite elements, e.g., in Constant Propagation
Maximum Fixed Point (MFP) Assignment

- Difficulties in computing MoP assignment
Maximum Fixed Point (MFP) Assignment

• Difficulties in computing MoP assignment
  
  ▶ In the presence of cycles there are infinite paths
    
    If all paths need to be traversed ⇒ Undecidability
Maximum Fixed Point (MFP) Assignment

- Difficulties in computing MoP assignment
  - In the presence of cycles there are infinite paths
    - If all paths need to be traversed ⇒ Undecidability
  - Even if a program is acyclic, every conditional multiplies the number of paths by two
    - If all paths need to be traversed ⇒ Intractability
Maximum Fixed Point (MFP) Assignment

- Difficulties in computing MoP assignment
  - In the presence of cycles there are infinite paths
    - If all paths need to be traversed ⇒ Undecidability
  - Even if a program is acyclic, every conditional multiplies the number of paths by two
    - If all paths need to be traversed ⇒ Intractability

- Why not merge information at intermediate points?
  - Merging is safe but may lead to imprecision.
  - Computes fixed point solutions of data flow equations.
Maximum Fixed Point (MFP) Assignment

- Difficulties in computing MoP assignment
  - In the presence of cycles there are infinite paths
    If all paths need to be traversed \(\Rightarrow\) Undecidability
  - Even if a program is acyclic, every conditional multiplies the number of paths by two
    If all paths need to be traversed \(\Rightarrow\) Intractability

- Why not merge information at intermediate points?
  - Merging is safe but may lead to imprecision.
  - Computes fixed point solutions of data flow equations.
Undecidability of Data Flow Analysis

- Reducing MPCP (Modified Post’s Correspondence Problem) to constant propagation
- MPCP is known to be undecidable
- If an algorithm exists for detecting all constants
  ⇒ MPCP would be decidable
- Since MPCP is undecidable
  ⇒ There does not exist an algorithm for detecting all constants
  ⇒ Static analysis is undecidable
Fixed Points Computation: Flow Functions Vs. Equations

• Consider a CFG with $N$ nodes.
  Let $\mathcal{X}$ be a vector $\langle \text{In}_1, \text{Out}_1, \ldots, \text{In}_N, \text{Out}_N \rangle$
Fixed Points Computation: Flow Functions Vs. Equations

- Consider a CFG with \( N \) nodes.
  Let \( \mathcal{X} \) be a vector \( \langle ln_1, out_1, \ldots, ln_N, out_N \rangle \)

- The data flow equations are:

\[
\begin{align*}
ln_1 &= f_{ln_1}(\mathcal{X}) & out_1 &= f_{out_1}(\mathcal{X}) \\
ln_2 &= f_{ln_2}(\mathcal{X}) & out_2 &= f_{out_2}(\mathcal{X}) \\
&\quad \cdots \\
ln_N &= f_{ln_N}(\mathcal{X}) & out_N &= f_{out_N}(\mathcal{X})
\end{align*}
\]

where \( f_p : \mathcal{L} \mapsto L \) and \( \mathcal{L} \) is an \( 2N \)-way product \( L \times L \times \ldots \times L \)
Fixed Points Computation: Flow Functions Vs. Equations

- Consider a CFG with N nodes. Let $\mathcal{X}$ be a vector $\langle \text{ln}_1, \text{out}_1, \ldots, \text{ln}_N, \text{out}_N \rangle$
- The data flow equations are:

\[
\begin{align*}
\text{ln}_1 &= f_{\text{ln}_1}(\mathcal{X}) & \text{out}_1 &= f_{\text{out}_1}(\mathcal{X}) \\
\text{ln}_2 &= f_{\text{ln}_2}(\mathcal{X}) & \text{out}_2 &= f_{\text{out}_2}(\mathcal{X}) \\
& \quad \vdots \\
\text{ln}_N &= f_{\text{ln}_N}(\mathcal{X}) & \text{out}_N &= f_{\text{out}_N}(\mathcal{X})
\end{align*}
\]

where $f_p : \mathcal{L} \mapsto \mathcal{L}$ and $\mathcal{L}$ is an $2N$-way product $\mathcal{L} \times \mathcal{L} \times \ldots \times \mathcal{L}$
- They can be rewritten as $\mathcal{X} = \mathcal{F}(\mathcal{X})$ where $\mathcal{F} : \mathcal{L} \mapsto \mathcal{L}$ is

\[
\mathcal{F}(\mathcal{X}) = \langle f_{\text{ln}_1}(\mathcal{X}), f_{\text{out}_1}(\mathcal{X}), \ldots, f_{\text{ln}_N}(\mathcal{X}), f_{\text{out}_N}(\mathcal{X}) \rangle
\]
Fixed Points Computation: Flow Functions Vs. Equations

• Consider a CFG with N nodes. Let $\mathcal{X}$ be a vector $\langle In_1, Out_1, \ldots, In_N, Out_N \rangle$

• The data flow equations are:

\[
\begin{align*}
In_1 &= f_{In_1}(\mathcal{X}) & Out_1 &= f_{Out_1}(\mathcal{X}) \\
In_2 &= f_{In_2}(\mathcal{X}) & Out_2 &= f_{Out_2}(\mathcal{X}) \\
\vdots \\
In_N &= f_{In_N}(\mathcal{X}) & Out_N &= f_{Out_N}(\mathcal{X})
\end{align*}
\]

where $f_p : L \mapsto L$ and $L$ is an 2N-way product $L \times L \times \ldots \times L$

• They can be rewritten as $\mathcal{X} = \mathcal{F}(\mathcal{X})$ where $\mathcal{F} : L \mapsto L$ is

\[
\mathcal{F}(\mathcal{X}) = \langle f_{In_1}(\mathcal{X}), f_{Out_1}(\mathcal{X}), \ldots, f_{In_N}(\mathcal{X}), f_{Out_N}(\mathcal{X}) \rangle
\]

• We compute the fixed points of equation $\mathcal{X} = \mathcal{F}(\mathcal{X})$
Tarski’s Fixed Point Theorem

Given monotonic \( f : L \mapsto L \) where \( L \) is a complete lattice

Define

- \( p \) is a fixed point of \( f \) :
  \[
  \text{Fix}(f) = \{ p \mid f(p) = p \}
  \]
- \( f \) is reductive at \( p \) :
  \[
  \text{Red}(f) = \{ p \mid f(p) \sqsubseteq p \}
  \]
- \( f \) is extensive at \( p \) :
  \[
  \text{Ext}(f) = \{ p \mid f(p) \sqsupseteq p \}
  \]

Then

- \( \text{LFP}(f) = \bigcap \text{Red}(f) \in \text{Fix}(f) \)
- \( \text{MFP}(f) = \bigcup \text{Ext}(f) \in \text{Fix}(f) \)
Fixed Points of a Function

\[ \top \rightarrow T \rightarrow \bot \]
Fixed Points of a Function

Red\( (f) \)

\( f^n(T) \)

\( T \)
Fixed Points of a Function

- $\text{Red}(f)$
- $\text{Ext}(f)$
- $f^n(\top)$
- $f^n(\bot)$
Fixed Points of a Function

Red\( (f) \)

\( f^n(\top) \)

\( MFP(f) \)

\( LFP(f) \)

\( f^n(\bot) \)

\( \bot \)
Examples of Reductive and Extensive Sets

Finite $L$ \hspace{2cm} Monotonic $f : L \mapsto L$

$$
\begin{align*}
\top & \mapsto \top \\
\downarrow & \downarrow \\
\nu_1 & \mapsto \nu_1 \\
\downarrow & \downarrow \\
\nu_2 & \mapsto \nu_2 \\
\downarrow & \downarrow \\
\nu_3 & \mapsto \nu_3 \\
\downarrow & \downarrow \\
\nu_4 & \mapsto \nu_4 \\
\downarrow & \downarrow \\
\bot & \mapsto \bot
\end{align*}
$$

$\text{Red}(f) = \{\top, \nu_3, \nu_4, \bot\}$

$\text{Ext}(f) = \{\top, \nu_1, \nu_2, \bot\}$

$\text{Fix}(f) = \text{Red}(f) \cap \text{Ext}(f)$

$= \{\top, \bot\}$

$\text{MFP}(f) = \text{lub} (\text{Ext}(f))$

$= \text{lub} (\text{Fix}(f))$

$= \top$

$\text{LFP}(f) = \text{glb} (\text{Red}(f))$

$= \text{glb} (\text{Fix}(f))$

$= \bot$
Existence of MFP: Proof of Tarski’s Fixed Point Theorem

$\text{Ext}(f)$
Existence of MFP: Proof of Tarski’s Fixed Point Theorem

- Claim 1: Let $X \subseteq L$. 
  \[ p \supseteq x, \forall x \in X \Rightarrow p \supseteq \bigcup(X). \]
Existence of MFP: Proof of Tarski’s Fixed Point Theorem

- Claim 1: Let $X \subseteq L$.
  \[ p \supseteq x, \ \forall x \in X \Rightarrow p \supseteq \bigcup(X). \]
- $\forall q \in \text{Ext}(f), \ hi \supseteq q$
Existence of MFP: Proof of Tarski’s Fixed Point Theorem

- Claim 1: Let $X \subseteq L$.
  \[ p \sqsubseteq x, \quad \forall x \in X \Rightarrow p \sqsubseteq \bigsqcup(X). \]
- $\forall q \in \text{Ext}(f), \quad hi \sqsupseteq q$
- $hi \sqsupseteq q$
  \[ \Rightarrow f(hi) \sqsupseteq f(q) \sqsupseteq q \quad \text{(monotonicity)} \]
  \[ \Rightarrow f(hi) \sqsupseteq hi \quad \text{(claim 1, hi is $\bigsqcup$)} \]
Existence of MFP: Proof of Tarski’s Fixed Point Theorem

- Claim 1: Let $X \subseteq L$
  \[ p \supseteq x, \ \forall x \in X \Rightarrow p \supseteq \bigcup(X). \]
- $\forall q \in \text{Ext}(f), \ hi \supseteq q$
- $hi \supseteq q$
  \( \Rightarrow f(hi) \supseteq f(q) \supseteq q \text{ (monotonicity)} \)
  \( \Rightarrow f(hi) \supseteq hi \text{ (claim 1, hi is } \sqcup \text{)} \)
- $f$ is extensive at $hi$ also: $hi \in \text{Ext}(f)$
Existence of MFP: Proof of Tarski’s Fixed Point Theorem

Claim 1: Let $X \subseteq L$.

- $p \sqsupseteq x, \forall x \in X \Rightarrow p \sqsupseteq \bigsqcup(X)$.
- $\forall q \in \text{Ext}(f), \ hi \sqsupseteq q$
- \( hi \sqsupseteq q \)
  \[ \Rightarrow f(hi) \sqsupseteq f(q) \sqsupseteq q \] (monotonicity)
  \[ \Rightarrow f(hi) \sqsupseteq hi \] (claim 1, \( hi \) is \( \bigsqcup \))

- \( f \) is extensive at \( hi \) also: \( hi \in \text{Ext}(f) \)
- \( f(hi) \sqsupseteq hi \Rightarrow f^2(hi) \sqsupseteq f(hi) \)
  \[ \Rightarrow f(hi) \in \text{Ext}(f) \]
  \[ \Rightarrow hi \sqsubset f(hi) \]
  \[ \Rightarrow hi = f(hi) \Rightarrow hi \in \text{Fix}(f) \]
Existence of MFP: Proof of Tarski’s Fixed Point Theorem

Claim 1: Let \( X \subseteq L \).
\[ p \sqsupseteq x, \forall x \in X \Rightarrow p \sqsupseteq \bigcup(X). \]
\[ \forall q \in \text{Ext}(f), \ hi \sqsupseteq q \]
\[ hi \sqsupseteq q \]
\[ \Rightarrow f(hi) \sqsupseteq f(q) \sqsupseteq q \quad \text{(monotonicity)} \]
\[ \Rightarrow f(hi) \sqsupseteq hi \quad \text{(claim 1, hi is } \bigcup) \]
\[ f \text{ is extensive at } hi \text{ also: } hi \in \text{Ext}(f) \]
\[ f(hi) \sqsupseteq hi \Rightarrow f^2(hi) \sqsupseteq f(hi) \]
\[ \Rightarrow f(hi) \in \text{Ext}(f) \]
\[ \Rightarrow hi \sqsubseteq f(hi) \]
\[ \Rightarrow hi = f(hi) \Rightarrow hi \in \text{Fix}(f) \]
\[ \text{Fix}(f) \subseteq \text{Ext}(f) \Rightarrow hi \sqsupseteq p, \ \forall p \in \text{Fix}(f) \]
Computing the Maximum Fixed Point

For monotonic $f : L \mapsto L$, if all descending chains are finite, then

$MFP(f) = f^{k+1}(\top) = f^k(\top)$

such that $f^{j+1}(\top) \neq f^j(\top)$, $j < k$. 
Computing the Maximum Fixed Point

For monotonic $f : L \mapsto L$, if all descending chains are finite, then $MFP(f) = f^{k+1}(\top) = f^k(\top)$ such that $f^{j+1}(\top) \neq f^j(\top)$, $j < k$. 
Computing the Maximum Fixed Point

For monotonic $f : L \mapsto L$, if all descending chains are finite, then $MFP(f) = f^{k+1}(\top) = f^k(\top)$ such that $f^{j+1}(\top) \neq f^j(\top)$, $j < k$. 
Computing the Maximum Fixed Point

For monotonic $f : L \mapsto L$, if all descending chains are finite, then $\text{MFP}(f) = f^{k+1}(\top) = f^k(\top)$ such that $f^{j+1}(\top) \neq f^j(\top)$, $j < k$.

$\top \supseteq f(\top) \supseteq f^2(\top) \supseteq f^3(\top) \supseteq f^4(\top) \supseteq \ldots$
Computing the Maximum Fixed Point

For monotonic $f : L \mapsto L$, if all descending chains are finite, then

$$MFP(f) = f^{k+1}(\top) = f^k(\top)$$

such that $f^{j+1}(\top) \neq f^j(\top)$, $j < k$.

- $\top \sqsupseteq f(\top) \sqsupseteq f^2(\top) \sqsupseteq f^3(\top) \sqsupseteq f^4(\top) \sqsupseteq \ldots$

- Since descending chains are finite, there must exist $f^k(\top)$ such that $f^{k+1}(\top) = f^k(\top)$ and $f^{j+1}(\top) \neq f^j(\top)$, $j < k$. 
Computing the Maximum Fixed Point

For monotonic $f : L \mapsto L$, if all descending chains are finite, then

$$MFP(f) = f^{k+1}(\bot) = f^k(\bot)$$

such that $f^{j+1}(\bot) \neq f^j(\bot), j < k$.

- $\bot \sqsupseteq f(\bot) \sqsupseteq f^2(\bot) \sqsupseteq f^3(\bot) \sqsupseteq f^4(\bot) \sqsupseteq \ldots$

- Since descending chains are finite, there must exist $f^k(\bot)$ such that $f^{k+1}(\bot) = f^k(\bot)$ and $f^{j+1}(\bot) \neq f^j(\bot), j < k$.

- If $p$ is a fixed point of $f$ then $f^k(\bot) \sqsupseteq p$.

Proof strategy: Induction on $i$ for $f^i(\bot)$
Computing the Maximum Fixed Point

For monotonic $f : L \mapsto L$, if all descending chains are finite, then

$$MFP(f) = f^{k+1}(\top) = f^k(\top)$$

such that $f^{j+1}(\top) \neq f^j(\top)$, $j < k$.

- $\top \supseteq f(\top) \supseteq f^2(\top) \supseteq f^3(\top) \supseteq f^4(\top) \supseteq \ldots$

- Since descending chains are finite, there must exist $f^k(\top)$ such that $f^{k+1}(\top) = f^k(\top)$ and $f^{j+1}(\top) \neq f^j(\top)$, $j < k$.

- If $p$ is a fixed point of $f$ then $f^k(\top) \supseteq p$.

Proof strategy: Induction on $i$ for $f^i(\top)$

- Basis ($i = 0$): $f^0(\top) = \top \supseteq p$.
- Inductive Hypothesis: Assume that $f^i(\top) \supseteq p$. 
Computing the Maximum Fixed Point

For monotonic \( f : L \mapsto L \), if all descending chains are finite, then
\[ MFP(f) = f^{k+1}(\top) = f^k(\top) \]
such that \( f^{j+1}(\top) \neq f^j(\top) \), \( j < k \).

- \( \top \supseteq f(\top) \supseteq f^2(\top) \supseteq f^3(\top) \supseteq f^4(\top) \supseteq \ldots \)
- Since descending chains are finite, there must exist \( f^k(\top) \) such that \( f^{k+1}(\top) = f^k(\top) \) and
  \( f^{j+1}(\top) \neq f^j(\top) \), \( j < k \).
- If \( p \) is a fixed point of \( f \) then \( f^k(\top) \supseteq p \).

Proof strategy: Induction on \( i \) for \( f^i(\top) \)

- Basis (\( i = 0 \)): \( f^0(\top) = \top \supseteq p \).
- Inductive Hypothesis: Assume that \( f^i(\top) \supseteq p \).
- Proof:
  \[
  \begin{align*}
  f(f^i(\top)) & \supseteq f(p) & (f \text{ is monotonic}) \\
  \implies f(f^i(\top)) & \supseteq p & (f(p) = p) \\
  \implies f^{i+1}(\top) & \supseteq p
  \end{align*}
  \]
Computing the Maximum Fixed Point

For monotonic \( f : L \mapsto L \), if all descending chains are finite, then 
\[ \text{MFP}(f) = f^{k+1}(\top) = f^k(\top) \] 
such that \( f^{j+1}(\top) \neq f^j(\top), \ j < k \).

- \( \top \sqsupseteq f(\top) \sqsupseteq f^2(\top) \sqsupseteq f^3(\top) \sqsupseteq f^4(\top) \sqsupseteq \ldots \)

- Since descending chains are finite, there must exist 
  
  \( f^k(\top) \) such that 
  
  \( f^{k+1}(\top) = f^k(\top) \) and 
  
  \( f^{j+1}(\top) \neq f^j(\top), \ j < k \).

- If \( p \) is a fixed point of \( f \) then \( f^k(\top) \sqsupseteq p \).

Proof strategy: Induction on \( i \) for \( f^i(\top) \)

- Basis \((i = 0)\): \( f^0(\top) = \top \sqsupseteq p \).
- Inductive Hypothesis: Assume that \( f^i(\top) \sqsupseteq p \).
- Proof:
  \[ f(f^i(\top)) \sqsupseteq f(p) \quad (f \text{ is monotonic}) \]
  \[ \Rightarrow f(f^i(\top)) \sqsupseteq p \quad (f(p) = p) \]
  \[ \Rightarrow f^{i+1}(\top) \sqsupseteq p \]

- \( \Rightarrow f^{k+1}(\top) \) is the MFP.
Existence and Computation of the Maximum Fixed Point

- For monotonic $f : L \mapsto L$
  - **Existence:** $MFP(f) = \bigsqcup \text{Ext}(f) \in \text{Fix}(f)$
    Requires $L$ to be complete.
  - **Computation:** $MFP(f) = f^{k+1}(\top) = f^k(\top)$ such that $f^{j+1}(\top) \neq f^j(\top), j < k$.
    Requires all strictly descending chains to be finite.
- Finite strictly descending chains $\Rightarrow$ Completeness of lattice
  Completeness of lattice $\not\Rightarrow$ Finite strictly descending chains
  $\Rightarrow$ Even if MFP exists, it may not be reachable unless all strictly descending chains are finite.
Possible Assignments as Solutions of Data Flow Analyses

All possible assignments
Possible Assignments as Solutions of Data Flow Analyses

All possible assignments

All safe assignments
Possible Assignments as Solutions of Data Flow Analyses

- All possible assignments
- All safe assignments
- All fixed point solutions
Possible Assignments as Solutions of Data Flow Analyses

All possible assignments

\[ \forall i, In_i = Out_i = \top \]

All safe assignments

All fixed point solutions

Uday Khedker
IIT Bombay
Possible Assignments as Solutions of Data Flow Analyses

- All possible assignments
  \[ \forall i, \text{In}_i = \text{Out}_i = \top \]

- All safe assignments
  \[ \forall i, \text{In}_i = \text{Out}_i = \perp \]

- All fixed point solutions
Possible Assignments as Solutions of Data Flow Analyses

- All possible assignments: $\forall i, In_i = Out_i = \top$
- All safe assignments: $\forall i, In_i = Out_i = \bot$
- All fixed point solutions
Possible Assignments as Solutions of Data Flow Analyses

- All possible assignments: \( \forall i, In_i = Out_i = \top \)
- All safe assignments
- All fixed point solutions: \( \forall i, In_i = Out_i = \bot \)
- Meet Over Paths Assignment

Uday Khedker
IIT Bombay
Possible Assignments as Solutions of Data Flow Analyses

- All possible assignments
- All safe assignments
- All fixed point solutions

\[ \forall i, \text{In}_i = \text{Out}_i = \top \]

Meet Over Paths Assignment

\[ \forall i, \text{In}_i = \text{Out}_i = \bot \]

Maximum Fixed Point

Least Fixed Point

Uday Khedker
IIT Bombay
Safety of MFP Solution: $\text{MFP} \subseteq \text{MoP}$
Safety of MFP Solution: MFP $\subseteq$ MoP

\[ \text{MoP}(v) = \bigcap_{\rho \in \text{Paths}(v)} f_{\rho}(B = \text{Boundary_Info}) \]
Safety of MFP Solution: \( \text{MFP} \subseteq \text{MoP} \)

- \( \text{MoP}(v) = \bigcap_{\rho \in \text{Paths}(v)} \text{f}_\rho(B = \text{Boundary}_\text{Info}) \)
- Proof Obligation: \( \forall \rho_v \text{ MFP}(v) \subseteq \text{f}_{\rho_v}(B) \)
Safety of MFP Solution: MFP ⊆ MoP

- $MoP(v) = \bigcap_{\rho \in Paths(v)} f_\rho(B = \text{Boundary_Info})$
- Proof Obligation: $\forall \rho_v MFP(v) \subseteq f_{\rho_v}(B)$
- Claim 1: $\forall u \rightarrow v, MFP(v) \subseteq f_{u\rightarrow v}(MFP(u))$
Safety of MFP Solution: $\text{MFP} \subseteq \text{MoP}$

- $\text{MoP}(v) = \bigcap_{\rho \in \text{Paths}(v)} f_\rho(B = \text{Boundary_Info})$
- Proof Obligation: $\forall \rho v \ MFP(v) \subseteq f_{\rho v}(B)$
- Claim 1: $\forall u \rightarrow v, MFP(v) \subseteq f_{u \rightarrow v}(MFP(u))$
- Proof Outline: Induction on path length
  - Base case: Path of length 0.
    $MFP(\text{Entry}) = \text{MoP}(\text{Entry}) = B$
  - Inductive hypothesis: Assume it holds for paths consisting of $k$ edges (say at $u$)
    $MFP(u) \subseteq f_{\rho u}(B)$  (Inductive hypothesis)
    $MFP(v) \subseteq f_{u \rightarrow v}(MFP(u))$  (Claim 1)
    $\Rightarrow MFP(v) \subseteq f_{u \rightarrow v}(f_{\rho u}(B))$
    $\Rightarrow MFP(v) \subseteq f_{\rho v}(B)$
Assignments for Constant Propagation Example

\begin{enumerate}
\item $n_1$
- $a = 1$
- $b = 2$
- $c = a + b$
\item $n_2$
- $c = a + b$
- $d = a \times b$
\item $n_3$
- $d = c - 1$
- $a = 2$
- $b = 1$
- $c = a + b$
\end{enumerate}
Assignments for Constant Propagation Example

\begin{align*}
\text{n}_1: & \quad a = 1 \\
& \quad b = 2 \\
& \quad c = a + b \\
\text{n}_3: & \quad d = c - 1 \\
& \quad a = 2 \\
& \quad b = 1 \\
& \quad c = a + b \\
\text{MoP} & \quad \langle \top, \top, \top, \top \rangle \\
& \quad \langle 1, 2, 3, \top \rangle \\
& \quad \langle \bot, \bot, 3, 2 \rangle \\
& \quad \langle \bot, \bot, 3, 2 \rangle \\
& \quad \langle 2, 1, 3, 2 \rangle
\end{align*}
Assignments for Constant Propagation Example

\begin{align*}
\text{n}_1 & : \quad a = 1 \\
& \quad b = 2 \\
& \quad c = a + b
\end{align*}

\begin{align*}
\text{n}_2 & : \quad c = a + b \\
& \quad d = a \times b
\end{align*}

\begin{align*}
\text{n}_3 & : \quad d = c - 1 \\
& \quad a = 2 \\
& \quad b = 1 \\
& \quad c = a + b
\end{align*}

MoP
\[
\langle \top, \top, \top, \top \rangle \\
\langle 1, 2, 3, \top \rangle \\
\langle \bot, \bot, 3, 2 \rangle \\
\langle \bot, \bot, 3, 2 \rangle \\
\langle \bot, \bot, 3, 2 \rangle \\
\langle 2, 1, 3, \bot \rangle
\]

MFP
\[
\langle \top, \top, \top, \top \rangle \\
\langle 1, 2, 3, \top \rangle \\
\langle \bot, \bot, \bot, \bot \rangle \\
\langle \bot, \bot, \bot, \bot \rangle \\
\langle \bot, \bot, \bot, \bot \rangle \\
\langle 2, 1, 3, \bot \rangle
\]

Uday Khedker IIT Bombay
Part 7

Performing Data Flow Analysis
Performing Data Flow Analysis

- Algorithms for computing MFP solution
- Complexity of data flow analysis
- Factor affecting the complexity of data flow analysis
Iterative Methods of Performing Data Flow Analysis

Successive recomputation after conservative initialization (T)

- *Round Robin*. Repeated traversals over nodes in a fixed order
  
  Termination: After values stabilise
  
  + Simplest to understand and implement
  
  − May perform unnecessary computations
Iterative Methods of Performing Data Flow Analysis

Successive recomputation after conservative initialization (⊤)

- **Round Robin.** Repeated traversals over nodes in a fixed order
  - Termination: After values stabilise
    - Simplest to understand and implement
    - May perform unnecessary computations

Our examples use this method.
Iterative Methods of Performing Data Flow Analysis

Successive recomputation after conservative initialization ($\top$)

- **Round Robin.** Repeated traversals over nodes in a fixed order
  
  Termination: After values stabilise
  - Simplest to understand and implement
  - May perform unnecessary computations

- **Work List.** Dynamic list of nodes which need recomputation
  
  Termination: When the list becomes empty
  - Demand driven. Avoid unnecessary computations.
  - Overheads of maintaining work list.

Our examples use this method.
Delayed computations of dependent data flow values of dependent nodes.

Find suitable single-entry regions.

- *Interval Based Analysis*. Uses graph partitioning.
- *$T_1$, $T_2$ Based Analysis*. Uses graph parsing.
Round Robin Iterative Algorithm for Computing MFP Assignment

1. \( ln_0 = \text{Boundary}_\text{Info} \)
2. \( \text{for all } j \neq 0, ln_j = T; \)
3. \( \text{change} = \text{true} \)
4. \( \text{while } \text{change} \text{ do} \)
5. \( \{ \text{change} = \text{false} \)
6. \( \text{for } j = 1 \text{ to } N - 1 \text{ do} \)
7. \( \{ \text{temp} = \prod_{p \in \text{pred}(j)} f_p(ln_p) \)
8. \( \text{if } \text{temp} \neq ln_j \text{ then} \)
9. \( \{ ln_j = \text{temp} \)
10. \( \text{change} = \text{true} \)
11. \( \} \)
12. \( \} \)
13. \( \} \)
Complexity of Round Robin Iterative Algorithm

- Unidirectional bit vector frameworks
  - Construct a spanning tree $T$ of $G$ to identify postorder traversal
  - Traverse $G$ in reverse postorder for forward problems and Traverse $G$ in postorder for backward problems
  - Depth $d(G, T)$: Maximum number of back edges in any acyclic path

<table>
<thead>
<tr>
<th>Task</th>
<th>Number of iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialisation</td>
<td>1</td>
</tr>
<tr>
<td>Convergence (until change remains true)</td>
<td>$d(G, T)$</td>
</tr>
<tr>
<td>Verifying convergence (change becomes false)</td>
<td>1</td>
</tr>
</tbody>
</table>
Complexity of Round Robin Iterative Algorithm

- Unidirectional bit vector frameworks
  - Construct a spanning tree $T$ of $G$ to identify postorder traversal
  - Traverse $G$ in reverse postorder for forward problems and Traverse $G$ in postorder for backward problems
  - Depth $d(G, T)$: Maximum number of back edges in any acyclic path

<table>
<thead>
<tr>
<th>Task</th>
<th>Number of iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialisation</td>
<td>1</td>
</tr>
<tr>
<td>Convergence (until change remains true)</td>
<td>$d(G, T)$</td>
</tr>
<tr>
<td>Verifying convergence (change becomes false)</td>
<td>1</td>
</tr>
</tbody>
</table>

- What about bidirectional bit vector frameworks?
Complexity of Round Robin Iterative Algorithm

- Unidirectional bit vector frameworks
  - Construct a spanning tree $T$ of $G$ to identify postorder traversal
  - Traverse $G$ in reverse postorder for forward problems and Traverse $G$ in postorder for backward problems
  - Depth $d(G, T)$: Maximum number of back edges in any acyclic path

<table>
<thead>
<tr>
<th>Task</th>
<th>Number of iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialisation</td>
<td>1</td>
</tr>
<tr>
<td>Convergence (until change remains true)</td>
<td>$d(G, T)$</td>
</tr>
<tr>
<td>Verifying convergence (change becomes false)</td>
<td>1</td>
</tr>
</tbody>
</table>

- What about bidirectional bit vector frameworks?
- What about other frameworks?
Example: Original formulation of PRE

- Information could flow along arbitrary paths
Example: Original formulation of PRE

- Information could flow along arbitrary paths
Complexity of Bidirectional Bit Vector Frameworks

Example: Original formulation of PRE

- Information could flow along arbitrary paths
Example: Original formulation of PRE

- Information could flow along arbitrary paths
Complexity of Bidirectional Bit Vector Frameworks

Example: Original formulation of PRE

- Information could flow along arbitrary paths
Complexity of Bidirectional Bit Vector Frameworks

Example: Original formulation of PRE

- Information could flow along arbitrary paths
Example: Original formulation of PRE

- Information could flow along arbitrary paths
Complexity of Bidirectional Bit Vector Frameworks

Example: Original formulation of PRE

- Information could flow along arbitrary paths
Complexity of Bidirectional Bit Vector Frameworks

Example: Original formulation of PRE

- Information could flow along arbitrary paths
Complexity of Bidirectional Bit Vector Frameworks

Example: Original formulation of PRE

- Information could flow along arbitrary paths
- Theoretically predicted number: 144
Complexity of Bidirectional Bit Vector Frameworks

Example: Original formulation of PRE

- Information could flow along arbitrary paths
- Theoretically predicted number: 144
- Practical number: 5.
Lacuna with PRE Complexity

- Lacuna with PRE: Complexity $O(n^2)$ traversals. Practical graphs may have up to 50 nodes.
  - Predicted number of traversals: 2,500.
  - Practical number of traversals: $\leq 5$.
- No explanation for about 14 years despite dozens of efforts.
- Not much experimentation with performing advanced optimizations involving bidirectional dependency.
Complexity of Round Robin Iterative Method

- Buy OTC (Over-The-Counter) medicine. No U-Turn 1 Trip
Complexity of Round Robin Iterative Method

- Buy OTC (Over-The-Counter) medicine. No U-Turn 1 Trip
- Buy cloth. Give it to the tailor for stitching. No U-Turn 1 Trip
Complexity of Round Robin Iterative Method

- Buy OTC (Over-The-Counter) medicine. No U-Turn 1 Trip
- Buy cloth. Give it to the tailor for stitching. No U-Turn 1 Trip
- Buy medicine with doctor’s prescription. 1 U-Turn 2 Trips
Complexity of Round Robin Iterative Method

- Buy OTC (Over-The-Counter) medicine. No U-Turn 1 Trip
- Buy cloth. Give it to the tailor for stitching. No U-Turn 1 Trip
- Buy medicine with doctor’s prescription. 1 U-Turn 2 Trips
- Buy medicine with doctor’s prescription. 2 U-Turns 3 Trips

The diagnosis requires X-Ray.
Complexity of Bidirectional Bit Vector Frameworks

- Every “incompatible” edge traversal
  $\Rightarrow$ One additional graph traversal
Every “incompatible” edge traversal ⇒ One additional graph traversal

Max. Incompatible edge traversals = Width of the graph = 0?

Maximum number of traversals = 1 + Max. incompatible edge traversals
Complexity of Bidirectional Bit Vector Frameworks

- Every “incompatible” edge traversal ⇒ One additional graph traversal
- Max. Incompatible edge traversals = Width of the graph = 1?
- Maximum number of traversals = 1 + Max. incompatible edge traversals
Complexity of Bidirectional Bit Vector Frameworks

- Every “incompatible” edge traversal \( \Rightarrow \) One additional graph traversal
- Max. Incompatible edge traversals = \( \text{Width of the graph} = 2? \)
- Maximum number of traversals = \( 1 + \text{Max. incompatible edge traversals} \)
Complexity of Bidirectional Bit Vector Frameworks

- Every “incompatible” edge traversal ⇒ One additional graph traversal
- Max. Incompatible edge traversals = \( \text{Width of the graph} = 3 \)?
- Maximum number of traversals = \( 1 + \text{Max. incompatible edge traversals} \)
Complexity of Bidirectional Bit Vector Frameworks

- Every "incompatible" edge traversal ⇒ One additional graph traversal
- Max. Incompatible edge traversals = Width of the graph = 3?
- Maximum number of traversals = 1 + Max. incompatible edge traversals
Complexity of Bidirectional Bit Vector Frameworks

- Every “incompatible” edge traversal ⇒ One additional graph traversal
- Max. Incompatible edge traversals = \textit{Width} of the graph = 3?
- Maximum number of traversals = 1 + Max. incompatible edge traversals
Complexity of Bidirectional Bit Vector Frameworks

- Every “incompatible” edge traversal $\Rightarrow$ One additional graph traversal
- Max. Incompatible edge traversals $= \text{Width of the graph} = 3$?
- Maximum number of traversals $= 1 + \text{Max. incompatible edge traversals}$
Complexity of Bidirectional Bit Vector Frameworks

- Every “incompatible” edge traversal $\Rightarrow$ One additional graph traversal

- Max. Incompatible edge traversals $= \text{Width of the graph} = 4$

- Maximum number of traversals $= 1 + \text{Max. incompatible edge traversals}
Complexity of Bidirectional Bit Vector Frameworks

• Every “incompatible” edge traversal ⇒ One additional graph traversal

• Max. Incompatible edge traversals = \textit{Width} of the graph = 4

• Maximum number of traversals = \(1 + 4 = 5\)
Framework Properties Influencing Complexity

Depends on the loop closure properties of the framework

\[ f^*(x) = x \cap f(x) \cap f^2(x) \cap \ldots \cap f^{k-1}(x) \]
Framework Properties Influencing Complexity

Depends on the loop closure properties of the framework

$k$-Bounded Frameworks

Fast Frameworks ($k = 2$)

\[ f^2(x) \supseteq f(x) \cap x \]

\[ f^*(x) = x \cap f(x) \cap f^2(x) \cap \ldots \cap f^{k-1}(x) \]

Necessary and sufficient
Framework Properties Influencing Complexity

Depends on the loop closure properties of the framework

\[ f^2(x) \supseteq f(x) \]

\[ f^2(x) \supseteq f(x) \cap x \]

\[ f^*(x) = x \cap f(x) \cap f^2(x) \cap \ldots \cap f^{k-1}(x) \]

- *k*-Bounded Frameworks
- Fast Frameworks \((k = 2)\)
- Rapid Frameworks

Necessary but not sufficient

Necessary and sufficient
Framework Properties Influencing Complexity

Depends on the loop closure properties of the framework

\[ f^2(x) = f(x) \]

\[ f^2(x) \supseteq f(x) \]

\[ f^2(x) \supseteq f(x) \cap x \]

\[ f^*(x) = x \cap f(x) \cap f^2(x) \cap \ldots \cap f^{k-1}(x) \]
Complexity of Round Robin Iterative Algorithm

- Unidirectional rapid frameworks

<table>
<thead>
<tr>
<th>Task</th>
<th>Number of iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Irreducible $G$</td>
</tr>
<tr>
<td>Initialisation</td>
<td>1</td>
</tr>
<tr>
<td>Convergence (until change remains true)</td>
<td>$d(G, T) + 1$</td>
</tr>
<tr>
<td>Verifying convergence (change becomes false)</td>
<td>1</td>
</tr>
</tbody>
</table>
Loop Closures in Bit Vector Frameworks

- Flow functions in bit vector frameworks have constant Gen and Kill

\[ f^*(x) = x \cap f(x) \cap f^2(x) \cap f^3(x) \cap \ldots \]
\[ f^2(x) = f \left( \text{Gen} \cup (x - \text{Kill}) \right) \]
\[ = \text{Gen} \cup ((\text{Gen} \cup (x - \text{Kill})) - \text{Kill}) \]
\[ = \text{Gen} \cup ((\text{Gen} - \text{Kill}) \cup (x - \text{Kill})) \]
\[ = \text{Gen} \cup (\text{Gen} - \text{Kill}) \cup (x - \text{Kill}) \]
\[ = \text{Gen} \cup (x - \text{Kill}) = f(x) \]
\[ f^*(x) = x \cap f(x) \]
Loop Closures in Bit Vector Frameworks

- Flow functions in bit vector frameworks have constant Gen and Kill

\[ f^*(x) = x \cap f(x) \cap f^2(x) \cap f^3(x) \cap \ldots \]
\[ f^2(x) = f(Gen \cup (x - Kill)) \]
\[ = Gen \cup ((Gen \cup (x - Kill)) - Kill) \]
\[ = Gen \cup ((Gen - Kill) \cup (x - Kill)) \]
\[ = Gen \cup (Gen - Kill) \cup (x - Kill) \]
\[ = Gen \cup (x - Kill) = f(x) \]

- Loop Closures of Bit Vector Frameworks are 2-bounded.
Loop Closures in Bit Vector Frameworks

- Flow functions in bit vector frameworks have constant Gen and Kill

\[
f^*(x) = x \sqcap f(x) \sqcap f^2(x) \sqcap f^3(x) \sqcap \ldots
\]
\[
f^2(x) = f(Gen \cup (x - Kill))
\]
\[
= Gen \cup ((Gen \cup (x - Kill)) - Kill)
\]
\[
= Gen \cup ((Gen - Kill) \cup (x - Kill))
\]
\[
= Gen \cup (Gen - Kill) \cup (x - Kill)
\]
\[
= Gen \cup (x - Kill) = f(x)
\]
\[
f^*(x) = x \sqcap f(x)
\]

- Loop Closures of Bit Vector Frameworks are 2-bounded.

- Intuition: Since Gen and Kill are constant, same things are generated or killed in every application of $f$.

  Multiple applications of $f$ are not required unless the input value changes.
What About Constant Propagation?

1. 

2. \( a = b + 1 \)

3. \( b = c + 1 \)

4. \( c = d + 1 \)

5. \( d = 2 \)
What About Constant Propagation?

Iteration #1
What About Constant Propagation?

1. \( a = b + 1 \)
2. \( b = c + 1 \)
3. \( c = d + 1 \)
4. \( d = 2 \)

Iteration #1

1. \( a = b + 1 \)
2. \( a = b + 1 \)
3. \( b = c + 1 \)
4. \( c = d + 1 \)
5. \( d = 2 \)

Iteration #2
What About Constant Propagation?

Iteration #1

a = b + 1
b = c + 1
c = d + 1
d = 2

Iteration #2

a = b + 1
b = c + 1
c = 3
d = 2

Iteration #3

a = b + 1
b = 4
c = 3
d = 2

Uday Khedker
IIT Bombay
What About Constant Propagation?

Iteration #1:

\[
\begin{align*}
1 & : a = b + 1 \\
2 & : a = b + 1 \\
3 & : b = c + 1 \\
4 & : c = d + 1 \\
5 & : d = 2
\end{align*}
\]

Iteration #2:

\[
\begin{align*}
1 & : a = b + 1 \\
2 & : a = b + 1 \\
3 & : b = c + 1 \\
4 & : c = 3 \\
5 & : d = 2
\end{align*}
\]

Iteration #3:

\[
\begin{align*}
1 & : a = b + 1 \\
2 & : a = b + 1 \\
3 & : b = 4 \\
4 & : c = 3 \\
5 & : d = 2
\end{align*}
\]

Iteration #4:

\[
\begin{align*}
1 & : a = 5 \\
2 & : a = 5 \\
3 & : b = 3 \\
4 & : c = d + 1 \\
5 & : d = 2
\end{align*}
\]
Larger Values of Loop Closure Bounds

- Fast Frameworks (eg. bit vector frameworks)
  Data flow values of different entities are independent
  (Entities refer to expressions, variables etc.)

- Non-fast frameworks
  Data flow values of different entities are inter-dependent
  Loop closure bound depends on the number of entities
Separability

\[ f : L \mapsto L \text{ is } \langle \hat{h}_1, \hat{h}_2, \ldots, \hat{h}_m \rangle \text{ where } \hat{h}_i \text{ computes the value of } \hat{x}_i \]
Separability

\[ f : L \leftrightarrow L \text{ is } \langle \hat{h}_1, \hat{h}_2, \ldots, \hat{h}_m \rangle \text{ where } \hat{h}_i \text{ computes the value of } \hat{x}_i \]

| Separable | Non-Separable |
Separability

\[ f : L \mapsto L \text{ is } \langle \hat{h}_1, \hat{h}_2, \ldots, \hat{h}_m \rangle \text{ where } \hat{h}_i \text{ computes the value of } \hat{x}_i \]

**Separable**

\[
\begin{align*}
\langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \\
\downarrow f \\
\langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle
\end{align*}
\]

**Non-Separable**

\[
\begin{align*}
\langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \\
\downarrow f \\
\langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle
\end{align*}
\]
Separability

\[ f : L \mapsto L \text{ is } \langle \hat{h}_1, \hat{h}_2, \ldots, \hat{h}_m \rangle \text{ where } \hat{h}_i \text{ computes the value of } \hat{x}_i \]

### Separable

- \( \langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \)
- \( \hat{h}_2 \)
- \( \langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle \)

### Non-Separable

- \( \langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \)
- \( f \)
- \( \langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle \)
Separability

\[ f : L \leftrightarrow L \text{ is } \langle \hat{h}_1, \hat{h}_2, \ldots, \hat{h}_m \rangle \text{ where } \hat{h}_i \text{ computes the value of } \hat{x}_i \]

**Separable**

\[ \langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \]

\[ \hat{h}_2 \]

\[ \langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle \]

\[ \hat{h} : \hat{L} \leftrightarrow \hat{L} \]

**Non-Separable**

\[ \langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \]

\[ f \]

\[ \langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle \]
Separability

\[ f : L \leftrightarrow L \text{ is } \langle \hat{h}_1, \hat{h}_2, \ldots, \hat{h}_m \rangle \text{ where } \hat{h}_i \text{ computes the value of } \hat{x}_i \]

**Separable**

\[ \langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \]

\[ \hat{h}_2 \]

\[ \langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle \]

**Non-Separable**

\[ \langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \]

\[ \hat{h}_2 \]

\[ \langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle \]
Separability

\[ f : L \mapsto L \text{ is } \langle \hat{h}_1, \hat{h}_2, \ldots, \hat{h}_m \rangle \text{ where } \hat{h}_i \text{ computes the value of } \hat{x}_i \]

Separable

\[
\langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \\
\hat{h}_2
\]

\[
\langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle \\
\hat{h} : \hat{L} \mapsto \hat{L}
\]

Example: All bit vector frameworks

Non-Separable

\[
\langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle \\
\hat{h}_2
\]

\[
\langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle \\
\hat{h} : \hat{L} \mapsto \hat{L}
\]

Example: Constant Propagation
Separability of Bit Vector Frameworks

- $\hat{L}$ is $\{0, 1\}$, $L$ is $\{0, 1\}^m$
- $\hat{\sqcap}$ is either boolean AND or boolean OR
- $\hat{\top}$ and $\hat{\bot}$ are 0 or 1 depending on $\hat{\sqcap}$.
- $\hat{h}$ is a *bit function* and could be one of the following:

<table>
<thead>
<tr>
<th>Raise</th>
<th>Lower</th>
<th>Propagate</th>
<th>Negate</th>
</tr>
</thead>
<tbody>
<tr>
<td>[\hat{\top}, \hat{\bot}]</td>
<td>[\hat{\top}, \hat{\bot}]</td>
<td>[\hat{\top}, \hat{\bot}]</td>
<td>[\hat{\top}, \hat{\bot}]</td>
</tr>
<tr>
<td>[\hat{\bot}, \hat{\bot}]</td>
<td>[\hat{\bot}, \hat{\bot}]</td>
<td>[\hat{\bot}, \hat{\bot}]</td>
<td>[\hat{\bot}, \hat{\bot}]</td>
</tr>
</tbody>
</table>
Separability of Bit Vector Frameworks

- \( \hat{L} \) is \( \{0, 1\} \), \( L \) is \( \{0, 1\}^m \)
- \( \sqcap \) is either boolean AND or boolean OR
- \( \sqcap \) and \( \sqcup \) are 0 or 1 depending on \( \sqcap \).
- \( \hat{h} \) is a *bit function* and could be one of the following:

<table>
<thead>
<tr>
<th></th>
<th>Raise</th>
<th>Lower</th>
<th>Propagate</th>
<th>Negate</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \hat{\top} )</td>
<td>( \hat{\top} )</td>
<td>( \hat{\top} )</td>
<td>( \hat{\top} )</td>
<td>( \hat{\top} )</td>
</tr>
<tr>
<td>( \hat{\bot} )</td>
<td>( \hat{\bot} )</td>
<td>( \hat{\bot} )</td>
<td>( \hat{\bot} )</td>
<td>( \hat{\bot} )</td>
</tr>
</tbody>
</table>

Non-monotonicity
Data Flow Equations for Non-Separable Flows

- General flow functions can be written as

\[ f_n(X) = (X - \text{Kill}_n(X)) \cup \text{Gen}_n(X) \]

where Gen and Kill have constant and dependent parts

\[
\begin{align*}
\text{Gen}_n(X) &= \text{ConstGen}_n \cup \text{DepGen}_n(X) \\
\text{Kill}_n(X) &= \text{ConstKill}_n \cup \text{DepKill}_n(X)
\end{align*}
\]
Data Flow Equations for Non-Separable Flows

- General flow functions can be written as

\[ f_n(X) = (X - \text{Kill}_n(X)) \cup \text{Gen}_n(X) \]

where Gen and Kill have constant and dependent parts

\[
\text{Gen}_n(X) = \text{ConstGen}_n \cup \text{DepGen}_n(X)
\]

\[
\text{Kill}_n(X) = \text{ConstKill}_n \cup \text{DepKill}_n(X)
\]

- Bit vector frameworks are a special case

\[
\text{DepGen}_n(X) = \text{DepKill}_n(X) = \emptyset
\]
Part 8

Heap Reference Analysis
An Overview

- A reference (called a *link*) can be represented by an *access path*.
  
  Eg. “\( x \rightarrow lptr \rightarrow rptr \)”

- A link may be accessed in multiple ways

- Setting links to NULL
  
  ▶️ *Alias Analysis*. Identify all possible ways of accessing a link
  
  ▶️ *Liveness Analysis*. For each program point, identify “dead” links (i.e. links which are not accessed after that program point)
  
  ▶️ *Availability and Anticipability Analyses*. Dead links should be reachable for making NULL assignment.
  
  ▶️ *Code Transformation*. Set “dead” links to NULL
Assumptions

For simplicity of exposition

- Java model of heap access
  - Root variables are on stack and represent references to memory in heap.
  - Root variables cannot be pointed to by any reference.

- Simple extensions for C++
  - Root variables can be pointed to by other pointers.
  - Pointer arithmetic is not handled.
**Key Idea #1: Access Paths Denote Links**

- Root variables: $x, y, z$
- Field names: rptr, lptr
- Access path: $x \rightarrow rptr \rightarrow lptr$
  Semantically, sequence of “links”
- Frontier: name of the last link
- Live access path: Iff the link corresponding to its frontier is used in future
Liveness Analysis

Program

Statement involving memory references

Effect of the statement on the access paths

Live Access Paths

Semantic Information

Live Access Paths
Key Idea #2: Transfer of Access Paths

\[ x = x.n \]
Key Idea #2: Transfer of Access Paths

\[ x = x.n \]
Key Idea #2: Transfer of Access Paths

\[ x = x.n \]

\[ \ldots = x.r.d \]
Key Idea #2: Transfer of Access Paths

\[ x = x.n \]

\[ \{ x, x \rightarrow r \} \]

\[ \ldots = x.r.d \]
Key Idea #2: Transfer of Access Paths

\[ x = x.n \]

\[ \ldots = x.r.d \]

\[ \{x, x \rightarrow r\} \]
Key Idea #2: Transfer of Access Paths

\[ x = x.n \]

\[ \ldots = x.r.d \]

\{x, x \to r\}
Key Idea #2: Transfer of Access Paths

\[ x = x.n \]

\[ \ldots = x.r.d \]

Analysis

\[ \{ x, x \rightarrow r \} \]
Key Idea #2: Transfer of Access Paths

\[ x = x.n \]

\[ \ldots = x.r.d \]

Analysis

Generated

- Constant: \( \{x, x\rightarrow n\} \)
- Dependent: \( \{x\rightarrow n\rightarrow r\} \)

Killed

- Constant: \( \{x, x\rightarrow r\} \)
- Dependent: \( \emptyset \)

\( x \) after the assignment is same as the \( x\rightarrow n \) before the assignment.

Uday Khedker

IIT Bombay
Access Paths as Data Flow Values

\[ x = x.n \]

\[ \{ x, x \rightarrow r \} \]

\[ \ldots = x.r.d \]
Access Paths as Data Flow Values

\[ x = x.n \]
\[ \{x, x \rightarrow r\} \]
\[ \ldots = x.r.d \]

\[ \{x, x \rightarrow r\} \]
Access Paths as Data Flow Values

\[ x \rightarrow n \] extended with \( r \)

\[ \begin{align*}
  x &= x.n \\
  \{x, x \rightarrow r\} \\
  \ldots &= x.r.d \\
\end{align*} \]
Access Paths as Data Flow Values

\[ x = x.n \]
\[ \{x, x \rightarrow n, x \rightarrow n \rightarrow r\} \]
\[ \{x, x \rightarrow r\} \]
\[ \{x, x \rightarrow r\} \]
\[ \ldots = x.r.d \]
Access Paths as Data Flow Values

\[ x = x.n \]

\[ \{ x, x \rightarrow n, x \rightarrow n \rightarrow r \} \]

\[ \{ x, x \rightarrow r \} \]

\[ \{ x, x \rightarrow r \} \]

\[ \ldots = x.r.d \]
Access Paths as Data Flow Values

Anticipability of Heap References: An *All Paths* problem

![Diagram showing data flow analysis]

- \( x = x.n \)
- \( \{x, x \rightarrow n, x \rightarrow n \rightarrow r\} \)
- \( \{x, x \rightarrow r\} \)
- \( \{x, x \rightarrow r\} \)
- \( \ldots = x.r.d \)
Anticipability of Heap References: An *All Paths* problem

\[
\begin{align*}
\{x, x \to n, x \to n \to r\} \\
\{x, x \to r\} \cap \{x, x \to n, x \to n \to r\} \\
\{x, x \to r\} \\
\ldots = x.r.d
\end{align*}
\]
Access Paths as Data Flow Values

Anticipability of Heap References: An *All Paths* problem

\[
x = x.n
\]

\[
\{x\}
\]

\[
\{x, x \rightarrow n, x \rightarrow n \rightarrow r\}
\]

\[
\ldots = x.r.d
\]
Access Paths as Data Flow Values

Anticipability of Heap References: An All Paths problem

\[ x = x.n \]
\[ \{x\} \]
\[ \{x, x \rightarrow r\} \]
\[ \ldots = x.r.d \]
Liveness of Heap References: An *Any Path* problem

\[
x = x.n \quad \{ x, x \to n, x \to n \to r \}
\]

\[
\{ x, x \to r \}
\]

\[
\quad \{ x, x \to r \}
\]

\[
\ldots = x.r.d
\]
Access Paths as Data Flow Values

Liveness of Heap References: An *Any Path* problem

\[
x = x.n
\]

\[
\{x, x \rightarrow n, x \rightarrow n \rightarrow r\}
\]

\[
\{x, x \rightarrow r\} \cup \{x, x \rightarrow n, x \rightarrow n \rightarrow r\}
\]

\[
\{x, x \rightarrow r\}
\]

\[
\ldots = x.r.d
\]

Uday Khedker
Liveness of Heap References: An *Any Path* problem

\[ x \rightarrow n \text{ extended with } r, n, \text{ and } n \rightarrow r \]

\[ \{x, x \rightarrow r, x \rightarrow n, x \rightarrow n \rightarrow r\} \]

\[ \{x, x \rightarrow r\} \]

\[ \ldots = x.r.d \]
Access Paths as Data Flow Values

Liveness of Heap References: An Any Path problem

\[
\begin{align*}
\{x, x \mapsto n, x \mapsto n \mapsto n, x \mapsto n \mapsto r, x \mapsto n \mapsto n \mapsto r\} \\
\{x, x \mapsto r, x \mapsto n, x \mapsto n \mapsto r\} \\
\{x, x \mapsto r\} \\
\ldots = x.r.d
\end{align*}
\]
Liveness of Heap References: An *Any Path* problem

\[ x = x \cdot n \]

\[ \{x, x \rightarrow n, x \rightarrow n \rightarrow r, x \rightarrow n \rightarrow r, x \rightarrow n \rightarrow \cdots \rightarrow n \rightarrow r\} \]

\[ \{x, x \rightarrow r, x \rightarrow n, x \rightarrow n \rightarrow r, x \rightarrow n \rightarrow \cdots \rightarrow n \rightarrow r\} \]

\[ \{x, x \rightarrow r\} \]

\[ \ldots = x \cdot r \cdot d \]

*Infinite Number of Unbounded Access Paths*
Key Idea #3: Using Graphs as Data Flow Values

Finite Number of Bounded Structures
Key Idea #4: Include Program Point in Graphs

\[
\{x, x\rightarrow n, x\rightarrow n\rightarrow n, x\rightarrow n\rightarrow n\rightarrow n, \ldots\}
\]

Different occurrences of n’s in an access path are **Indistinguishable**

\[
\{x, x\rightarrow n, x\rightarrow n\rightarrow n, x\rightarrow n\rightarrow n\rightarrow r\}
\]

Different occurrences of n’s in an access path are **Distinct**
Key Idea #4: Include Program Point in Graphs

1. \( x = x.n \)

   \( \{ x, x \rightarrow n, x \rightarrow n \rightarrow n, x \rightarrow n \rightarrow n \rightarrow n, \ldots \} \)

   Different occurrences of \( n \)'s in an access path are \textbf{Indistinguishable}

2. \( x = x.n.r.d \)

   \( \{ x, x \rightarrow n, x \rightarrow n \rightarrow n, x \rightarrow n \rightarrow n \rightarrow r \} \)

   Different occurrences of \( n \)'s in an access path are \textbf{Distinct}

Access Graph:

\[ \text{Access Graph: } x \rightarrow n \rightarrow n_1 \rightarrow n_2 \rightarrow r \rightarrow r_2 \]
Key Idea #4: Include Program Point in Graphs

1. \( x = x.n \)

\( \{x, x \rightarrow n, x \rightarrow n \rightarrow n, x \rightarrow n \rightarrow n \rightarrow n, \ldots \} \)

*Different occurrences of n’s in an access path are Indistinguishable*

Access Graph:

\[ x \xrightarrow{n} n_1 \xleftarrow{n} n \]

2. \( x = x.n.r.d \)

\( \{x, x \rightarrow n, x \rightarrow n \rightarrow n, x \rightarrow n \rightarrow r\} \)

*Different occurrences of n’s in an access path are Distinct*

Access Graph:

\[ x \xrightarrow{n} n_1 \xrightarrow{n} n_2 \xrightarrow{r} r_2 \]
Inclusion of Program Point Facilitates Summarization

2  = x.n.d

3  x = x.r

4  = x.n.d
Inclusion of Program Point Facilitates Summarization

\[ x = x.r \]

\[ x = x.n.d \]

\[ G_4 \]

\[ x \rightarrow n \rightarrow n_4 \]

\[ x \rightarrow n \rightarrow n_4 \]

Uday Khedker
IIT Bombay
Inclusion of Program Point Facilitates Summarization
Inclusion of Program Point Facilitates Summarization

\[ x = x.r \]

\[ x = x.n.d \]

\[ x = x.r \]

\[ x = x.n.d \]
Inclusion of Program Point Facilitates Summarization

\[ G_1 = G_2 \cup_G G_3 \]
Inclusion of Program Point Facilitates Summarization

Iteration #1

Analysis

1 \( x = x.n \)

2 \( \ldots = x.r.d \)
Inclusion of Program Point Facilitates Summarization

Iteration #1

1 \( x = x.n \)

2 \( \ldots = x.r.d \)

\( x \rightarrow r \rightarrow r_2 \)
Inclusion of Program Point Facilitates Summarization

Iteration #1

1: $x = x.n$

2: ... $= x.r.d$

$\xrightarrow{r} r_2$

$\xrightarrow{r} r_2$
Inclusion of Program Point Facilitates Summarization

Iteration #1

1. \( x = x.n \)

2. \( \ldots = x.r.d \)

Analysis
Inclusion of Program Point Facilitates Summarization

Iteration #1

1 \( x = x.n \)

2 \( \ldots = x.r.d \)
Inclusion of Program Point Facilitates Summarization

Iteration #2

Analysis

1 \( x = x.n \)

2 \( \ldots = x.r.d \)
Inclusion of Program Point Facilitates Summarization

Iteration #2

1 \( x = x.n \)

2 \( \ldots = x.r.d \)
Inclusion of Program Point Facilitates Summarization

Iteration #2

1. $x = x \cdot n$

2. $\ldots = x \cdot r \cdot d$

Uday Khedker
IIT Bombay
Inclusion of Program Point Facilitates Summarization

Analysis

1 $x = x.n$

2 $\ldots = x.r.d$

Iteration #2

$x \rightarrow n \rightarrow n_1 \rightarrow r \rightarrow r_2$

$x \rightarrow n \rightarrow n_1 \leftarrow r \rightarrow r_2$

$x \rightarrow r \rightarrow r_2$
Inclusion of Program Point Facilitates Summarization

Iteration #3

$\text{Analysis}$

1. $x = x.n$

2. $\ldots = x.r.d$

$\bigcup_G x \stackrel{n}{\rightarrow} n_1 \stackrel{r}{\rightarrow} r_2$

$\bigcup_G x \stackrel{n}{\rightarrow} n_1 \stackrel{r}{\rightarrow} r_2$

$\bigcup_G x \stackrel{n}{\rightarrow} n_1 \stackrel{r}{\rightarrow} r_2$
Inclusion of Program Point Facilitates Summarization

Iteration #3
Inclusion of Program Point Facilitates Summarization

Iteration #3

1. $x = x.n$

2. $... = x.r.d$
Inclusion of Program Point Facilitates Summarization

Iteration #3

1. \( x = x.n \)
2. \( \ldots = x.r.d \)
Lattice of Access Graphs

- Finite number of nodes in an access graph for a variable
- $\bigcup G$ induces a partial order on access graphs
  - a finite (and hence complete) lattice
  - All standard results of classical data flow analysis can be extended to this analysis.

Termination and boundedness, convergence on MFP, complexity etc.
Access Graph Operations

- **Union.** $G \cup_G G'$
- **Path Removal.**
  $G \ominus \rho$ removes those access paths in $G$ which have $\rho$ as a prefix.
- **Factorization (\).**
- **Extension.**
Access Graph Operations: Examples

Program

1. \[ x = x.l \]
2. \[ y = x.r.d \]

Access Graphs

1. \[ g_1: x \]
2. \[ g_2: x \rightarrow r_2 \]
3. \[ g_3: x \rightarrow l_1 \]
4. \[ g_4: x \rightarrow l_1 \rightarrow r_2 \]
5. \[ g_5: x \rightarrow l_1 \rightarrow r_2 \]
6. \[ g_6: x \rightarrow l_1 \rightarrow r_2 \]

Remainder Graphs

1. \[ rg_1 \]
2. \[ rg_2 \]

<table>
<thead>
<tr>
<th>Union</th>
<th>Path Removal</th>
<th>Factorisation</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Access Graph Operations: Examples

Program | Access Graphs | Remainder Graphs
--- | --- | ---
1 \(x = x.l\) | \(g_1\) \(\xrightarrow{x}\) | \(rg_1\) \(\xrightarrow{r_2}\)
2 \(y = x.r.d\) | \(g_2\) \(\xrightarrow{x r_2}\) | \(rg_2\) \(\xrightarrow{l_1 r_2}\)

<table>
<thead>
<tr>
<th>Union</th>
<th>Path Removal</th>
<th>Factorisation</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>(g_3 \cup g_4 = g_4)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(g_2 \cup g_4 = g_5)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(g_5 \cup g_4 = g_5)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(g_5 \cup g_6 = g_6)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Uday Khedker
IIT Bombay
### Access Graph Operations: Examples

#### Program

1. $x = x.l$
2. $y = x.r.d$

#### Access Graphs

<table>
<thead>
<tr>
<th>Access Graphs</th>
<th>Remainder Graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g_1$</td>
<td>$g_1 \Rightarrow x$</td>
</tr>
<tr>
<td>$g_2$</td>
<td>$g_2 \Rightarrow x \rightarrow r_2$</td>
</tr>
<tr>
<td>$g_3$</td>
<td>$g_3 \Rightarrow x \rightarrow l_1$</td>
</tr>
<tr>
<td>$g_4$</td>
<td>$g_4 \Rightarrow x \rightarrow l_1 \rightarrow r_2$</td>
</tr>
<tr>
<td>$g_5$</td>
<td>$g_5 \Rightarrow x \rightarrow l_1 \rightarrow r_2$</td>
</tr>
<tr>
<td>$g_6$</td>
<td>$g_6 \Rightarrow x \rightarrow l_1 \rightarrow r_2$</td>
</tr>
<tr>
<td>$rg_1$</td>
<td>$rg_1 \Rightarrow r_2$</td>
</tr>
<tr>
<td>$rg_2$</td>
<td>$rg_2 \Rightarrow l_1 \rightarrow r_2$</td>
</tr>
</tbody>
</table>

#### Table

<table>
<thead>
<tr>
<th>Union</th>
<th>Path Removal</th>
<th>Factorisation</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g_3 \cup_G g_4 = g_4$</td>
<td>$g_6 \ominus x \rightarrow l = g_2$</td>
<td>$g_5 \ominus x = \mathcal{E}_G$</td>
<td></td>
</tr>
<tr>
<td>$g_2 \cup_G g_4 = g_5$</td>
<td>$g_5 \ominus x = \mathcal{E}_G$</td>
<td>$g_4 \ominus x \rightarrow r = g_4$</td>
<td></td>
</tr>
<tr>
<td>$g_5 \cup_G g_4 = g_5$</td>
<td>$g_4 \ominus x \rightarrow r = g_4$</td>
<td>$g_4 \ominus x \rightarrow l = g_1$</td>
<td></td>
</tr>
<tr>
<td>$g_5 \cup_G g_6 = g_6$</td>
<td>$g_4 \ominus x \rightarrow l = g_1$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Access Graph Operations: Examples**

<table>
<thead>
<tr>
<th>Program</th>
<th>Access Graphs</th>
<th>Remainder Graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 $x = x.l$</td>
<td>$g_1 \xrightarrow{x}$</td>
<td>$rg_1 \xrightarrow{r_2}$</td>
</tr>
<tr>
<td>2 $y = x.r.d$</td>
<td>$g_2 \xrightarrow{x \rightarrow l \rightarrow r_2}$</td>
<td>$g_3 \xrightarrow{x \rightarrow l_1}$</td>
</tr>
<tr>
<td></td>
<td>$g_4 \xrightarrow{x \rightarrow l_1 \rightarrow r_2}$</td>
<td>$g_5 \xrightarrow{x \rightarrow l_1 \rightarrow r_2}$</td>
</tr>
<tr>
<td></td>
<td>$g_6 \xrightarrow{x \rightarrow l_1 \rightarrow r_2}$</td>
<td></td>
</tr>
</tbody>
</table>

**Union**

<table>
<thead>
<tr>
<th>Union Operation</th>
<th>Path Removal</th>
<th>Factorisation</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g_3 \cup_G g_4 = g_4$</td>
<td>$g_6 \ominus x \rightarrow l = g_2$</td>
<td>$g_2 / (g_1, {x}) = {rg_1}$</td>
<td></td>
</tr>
<tr>
<td>$g_2 \cup_G g_4 = g_5$</td>
<td>$g_5 \ominus x = \epsilon_G$</td>
<td>$g_5 / (g_1, {x}) = {rg_1, rg_2}$</td>
<td></td>
</tr>
<tr>
<td>$g_5 \cup_G g_4 = g_5$</td>
<td>$g_4 \ominus x \rightarrow r = g_4$</td>
<td>$g_5 / (g_2, {r_2}) = {\epsilon_{RG}}$</td>
<td></td>
</tr>
<tr>
<td>$g_5 \cup_G g_6 = g_6$</td>
<td>$g_4 \ominus x \rightarrow l = g_1$</td>
<td>$g_4 / (g_2, {r_2}) = \emptyset$</td>
<td></td>
</tr>
</tbody>
</table>
Access Graph Operations: Examples

<table>
<thead>
<tr>
<th>Program</th>
<th>Access Graphs</th>
<th>Remainder Graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 $x = x.l$</td>
<td>$g_1$</td>
<td>$rg_1$</td>
</tr>
<tr>
<td>2 $y = x.r.d$</td>
<td>$g_4$, $g_5$</td>
<td>$rg_2$</td>
</tr>
</tbody>
</table>

**Union**

- $g_3 \cup_G g_4 = g_4$
- $g_2 \cup_G g_4 = g_5$
- $g_5 \cup_G g_4 = g_5$
- $g_5 \cup_G g_6 = g_6$

**Path Removal**

- $g_6 \ominus x \rightarrow l = g_2$
- $g_5 \ominus x = \mathcal{E}_G$
- $g_4 \ominus x \rightarrow r = g_4$
- $g_4 \ominus x \rightarrow l = g_1$

**Factorisation**

- $g_2/ (g_1, \{x\}) = \{rg_1\}$
- $g_5/ (g_1, \{x\}) = \{rg_1, \{r_2\}\}$
- $g_5/ (g_2, \{r_2\}) = \{\epsilon_{RG}\}$
- $g_4/ (g_2, \{r_2\}) = \emptyset$

**Extension**

- $(g_3, \{l_1\}) \# \{rg_1\} = g_4$
- $(g_3, \{l_1\}) \# \{rg_1, rg_2\} = g_6$
- $(g_2, \{r_2\}) \# \{\epsilon_{RG}\} = g_2$
- $(g_2, \{r_2\}) \# \emptyset = \mathcal{E}_G$
Availability Analysis of Example Program

1. \( w = x \)
2. \( \text{while} \ (x.data < \text{max}) \)
3. \( x = x.rptr \)
4. \( y = x.lptr \)
5. \( z = \text{New class_of_z} \)
6. \( y = y.lptr \)
7. \( z.sum = x.data + y.data \)
Anticipability Analysis of Example Program

1. \( w = x \)

2. \( \text{while } (x\text{.data} < \text{max}) \)

3. \( x = x\text{.rptr} \)

4. \( y = x\text{.lptr} \)

5. \( z = \text{New class\_of\_z} \)

6. \( y = y\text{.lptr} \)

7. \( z\text{.sum} = x\text{.data} + y\text{.data} \)

\{x\}

\{x\}

\{x\}

\{x, x\rightarrow\text{rptr} \}

\{x\}

\{x\}

\{x, x\rightarrow\text{lptr}, x\rightarrow\text{lptr}\rightarrow\text{lptr} \}

\{x, y, y\rightarrow\text{lptr} \}

\{x, y, y\rightarrow\text{lptr}, z \}

\{x, y, z \}

\{x\}

∅
Liveness Analysis of Example Program: 1st Iteration

1. \( w = x \)

2. while (x.data < max)

3. \( x = x.rptr \)

4. \( y = x.lptr \)

5. \( z = \text{New class} \_ \text{of} \_ z \)

6. \( y = y.lptr \)

7. \( z.\text{sum} = x.data + y.data \)
Liveness Analysis of Example Program: 2nd Iteration

1. \( w = x \)

2. \( \text{while (x.data < max)} \)

3. \( x = x.rptr \)

4. \( y = x.lptr \)

5. \( z = \text{New class of } z \)

6. \( y = y.lptr \)

7. \( z.sum = x.data + y.data \)
Liveness Analysis of Example Program: 3rd Iteration

1. \( w = x \)

2. \( \text{while (} x\text{.data < max) } \)

3. \( x = x\text{.rptr} \)

4. \( y = x\text{.lptr} \)

5. \( z = \text{New class of } z \)

6. \( y = y\text{.lptr} \)

7. \( z\text{.sum = } x\text{.data + y.datal} \)
Liveness Analysis of Example Program: 4th Iteration

1. \( w = x \)

2. \( \text{while} \ (x.\text{data} < \text{max}) \)

3. \( x = x.\text{rptr} \)

4. \( y = x.\text{lptr} \)

5. \( z = \text{New class of } z \)

6. \( y = y.\text{lptr} \)

7. \( z.\text{sum} = x.\text{data} + y.\text{data} \)
Which Access Paths Can be Nullified?

- Consider extensions of accessible paths for nullification.

Let \( \rho \) be accessible at \( p \) (i.e. available or anticipable) for each reference field \( f \) of the object pointed to by \( \rho \) if \( \rho \rightarrow f \) is not live at \( p \) then

Insert \( \rho \rightarrow f = \text{NULL} \) at \( p \) subject to profitability

- For simple access paths, \( \rho \) is empty and \( f \) is the root variable name.
Which Access Paths Can be Nullified?

- Consider extensions of accessible paths for nullification.

Let $\rho$ be accessible at $p$ (i.e. available or anticipable) for each reference field $f$ of the object pointed to by $\rho$ if $\rho \rightarrow f$ is not live at $p$ then

Insert $\rho \rightarrow f = \text{NULL}$ at $p$ subject to profitability

- For simple access paths, $\rho$ is empty and $f$ is the root variable name.
Which Access Paths Can be Nullified?

- Consider extensions of accessible paths for nullification.

```
Let ρ be accessible at p (i.e., available or anticipable)
for each reference field f of the object pointed to by ρ
if ρ→f is not live at p then
    Insert ρ→f = NULL at p subject to profitability
```

- For simple access paths, ρ is empty and f is the root variable name.
Which Access Paths Can be Nullified?

- Consider extensions of accessible paths for nullification.

  Let \( \rho \) be accessible at \( p \) (i.e., available or anticipable) for each reference field \( f \) of the object pointed to by \( \rho \)

  \[
  \text{if } \rho \rightarrow f \text{ is not live at } p \text{ then}
  \]

  Insert \( \rho \rightarrow f = \text{NULL} \) at \( p \) subject to profitability

- For simple access paths, \( \rho \) is empty and \( f \) is the root variable name.
Key Idea #5: Liveness Closure Under Link Aliasing

\[ x = y \]

\[ \ldots = x.n \]

\[ x \text{ and } y \text{ are node aliases} \]
Key Idea #5: Liveness Closure Under Link Aliasing

\[ x = y \]
\[ \ldots = x.n \]

\( x \) and \( y \) are node aliases

\( x.n \) and \( y.n \) are link aliases
Key Idea #5: Liveness Closure Under Link Aliasing

\[ x = y \]

\[ \ldots = x.n \]

\[ x \text{ and } y \text{ are node aliases} \]

\[ x.n \text{ and } y.n \text{ are link aliases} \]

\[ x \rightarrow n \text{ is live} \Rightarrow y \rightarrow n \text{ is live} \]
Key Idea #5: Liveness Closure Under Link Aliasing

\[ x = y \]

\[ \ldots = x.n \]

\[ x \quad \text{and} \quad y \quad \text{are node aliases} \]

\[ x.n \quad \text{and} \quad y.n \quad \text{are link aliases} \]

\[ x \rightarrow n \quad \text{is live} \quad \Rightarrow \quad y \rightarrow n \quad \text{is live} \]

Nullifying \( y \rightarrow n \) will have the side effect of nullifying \( x \rightarrow n \)
Issues in Using Access Graphs for Complete Liveness

• Explicit Liveness at $p$
  Liveness purely due to the program beyond $p$.
  The effect of execution before $p$ is not incorporated.
Issues in Using Access Graphs for Complete Liveness

- Explicit Liveness at $p$
  Liveness purely due to the program beyond $p$.
  The effect of execution before $p$ is not incorporated.

- Implicit Liveness at $p$
  Access paths that become live under link alias closure.
Issues in Using Access Graphs for Complete Liveness

- Explicit Liveness at $p$
  Liveness purely due to the program beyond $p$.
  The effect of execution before $p$ is not incorporated.

- Implicit Liveness at $p$
  Access paths that become live under link alias closure.
  - The set of implicitly live access paths may not be prefix closed.
Issues in Using Access Graphs for Complete Liveness

• Explicit Liveness at $p$
  Liveness purely due to the program beyond $p$.
  The effect of execution before $p$ is not incorporated.

• Implicit Liveness at $p$
  Access paths that become live under link alias closure.
    ▶ The set of implicitly live access paths may not be prefix closed.
    ▶ These *paths* are not accessed, their frontiers are accessed through some other access path
Issues in Using Access Graphs for Complete Liveness

- Explicit Liveness at $p$
  Liveness purely due to the program beyond $p$.
  The effect of execution before $p$ is not incorporated.

- Implicit Liveness at $p$
  Access paths that become live under link alias closure.
  - The set of implicitly live access paths may not be prefix closed.
  - These *paths* are not accessed, their frontiers are accessed through some other access path

- All paths in an access graph may not be access paths
Issues in Using Access Graphs for Complete Liveness

- **Explicit Liveness at** \( p \)
  Liveness purely due to the program beyond \( p \).
  The effect of execution before \( p \) is not incorporated.

- **Implicit Liveness at** \( p \)
  Access paths that become live under link alias closure.
  - The set of implicitly live access paths may not be prefix closed.
  - These *paths* are not accessed, their frontiers are accessed through some other access path

- **All paths in an access graph may not be access paths**
  - Define *intermediate* and *final* nodes in access graphs
  - Paths ending on final nodes are access paths through some other access path
Live and Accessible Paths

1. \( w = x \)

2. \( \text{while} \ (x.\text{data} < \text{max}) \)

3. \( x = x.\text{rptr} \)

4. \( y = x.\text{lptr} \)

5. \( z = \text{New class of z} \)

6. \( y = y.\text{lptr} \)

7. \( z.\text{sum} = x.\text{data} + y.\text{data} \)

Uday Khedker

IIT Bombay
Creating NULL Assignments from Live and Accessible Paths

\[\begin{align*}
\text{y} &= \text{z} = \text{NULL} \\
1 & \quad \text{w} = \text{x} \\
2 & \quad \text{w} = \text{NULL} \\
2 & \quad \text{while (x.data < max)} \\
3 & \quad \text{x} = \text{x.rptr} \\
3 & \quad \text{y} = \text{x.lptr} \\
4 & \quad \text{x.lptr} = \text{NULL} \\
4 & \quad \text{x.lptr.lptr} = \text{NULL} \\
4 & \quad \text{x.lptr.lptr.rptr} = \text{NULL} \\
5 & \quad \text{z} = \text{New class of z} \\
5 & \quad \text{z.lptr} = \text{z.rptr} = \text{NULL} \\
6 & \quad \text{y} = \text{y.lptr} \\
6 & \quad \text{y.lptr} = \text{y.rptr} = \text{NULL} \\
7 & \quad \text{z.sum} = \text{x.data + y.data} \\
7 & \quad \text{x} = \text{y} = \text{z} = \text{NULL}
\end{align*}\]
The Resulting Program

```
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

6  y = y.lptr
   y.lptr = y.rptr = null

7  z.sum = x.data + y.data
   x = y = z = null
```
Issues Not Covered in This Presentation

• Precision of information
  ▶ Implicit Vs. Explicit Liveness
  ▶ May Vs. Must Alias Analysis
  ▶ Cyclic Data Structure
  ▶ Eliminating Redundant NULL Assignments

• Properties of Data Flow Analysis:
  Monotonicity, Distributivity, Boundedness, Complexity

• Interprocedural Analysis

• Extensions for C/C++
Part 9

Conclusions
Conclusions

- Data flow analysis is a very powerful program analysis technique
- Requires us to design appropriate
  - Set of values with reasonable approximations
    ⇒ Acceptable partial order and merge operation
  - Monotonic functions which are closed under composition
Conclusions

- Data flow analysis can be used for discovering complex semantics
- Unbounded information can summarized using interesting insights
  - Example: Heap Analysis

  *Heap manipulations consist of repeating patterns which bear a close resemblance to program structure*

  Analysis of heap data is possible despite the fact that the mappings between access expressions and l-values keep changing
  
  - The basic theory can be applied by a careful design of representations and operations
BTW, What is Static Analysis of Heap?

Static vs Dynamic: 

- **Static**: Analysis done without execution of the program.
- **Dynamic**: Analysis done during the execution of the program.
BTW, What is Static Analysis of Heap?

Abstract, Bounded, Single Instance

Concrete, Unbounded, Infinitely Many

Static

Program Code

Dynamic

Program Execution
BTW, What is Static Analysis of Heap?

Abstract, Bounded, Single Instance

Concrete, Unbounded, Infinitely Many

Static

Program Code

Dynamic

Program Execution

Heap Memory

Heap Memory

Heap Memory

Heap Memory

Heap Memory

Heap Memory
BTW, What is Static Analysis of Heap?

Abstract, Bounded, Single Instance

Concrete, Unbounded, Infinitely Many

Static

Program Code

Summary Heap Data

Dynamic

Program Execution

Heap Memory

Heap Memory

Heap Memory

Heap Memory

?
BTW, What is Static Analysis of Heap?

Abstract, Bounded, Single Instance

Concrete, Unbounded, Infinitely Many

Static

Program Code

Summary

Heap Data

Dynamic

Profiling

Program Execution

Heap Memory

Uday Khedker

IIT Bombay
BTW, What is Static Analysis of Heap?

Abstract, Bounded, Single Instance

Concrete, Unbounded, Infinitely Many

Static

Program Code

Static Analysis

Summary Heap Data

Dynamic

Program Execution

Heap Memory

Uday Khedker
IIT Bombay
Last but not the least . . .

Thank You!