Dataflow Analysis

Lecture 17

A COMPROMISE SOLUTION TO THE PI/TAU DISPUTE

Happy Pi day

March 14, 2018
Control Flow Graphs

- Directed graph structure that represents how execution flows during the program’s life.
- Typically consider CFGs on per-method basis.

Terminology:
- \( \text{PRED}(A) \): The set of Basic Blocks that are parents (predecessor) of basic block \( A \)
  - \( B \in \text{PRED}(A) \) if \( B \to A \) in CFG
- \( \text{SUCC}(A) \): The set of Basic Blocks that are children (successor) of basic block \( A \)
  - \( B \in \text{SUCC}(A) \) if \( A \to B \) in CFG
Predecessors and Successors

Compiler Construction
Example

```java
1 class Main {
2   a : Int;
3   b : Int;
4   c : Int;
5   i : Int;
6   main () : Object {{
7       a <- 2;
8       b <- 3;
9       if ( a = b ) then
10          while i < 10 loop {
11              c <- c + b;
12              i <- i + 1;
13          } pool
14       else
15          c <- 5
16          fi;
17  };
18 };}
19 }
```
Example (2)

```java
class Main {
    a : Int;
    b : Int;
    c : Int;
    i : Int;
    main() : Object {}{
        a <- 2;
        b <- 3;
        if (a = b) then
            goto main_1
        else
            goto main_2
    }
}
```

```
main_0:
a <- 2
b <- 3
if a = b
goto main_1
else goto main_2
```

```
main_1:
jmp main_4 ; why?
```

```
main_2:
c <- 5
jmp main_3
```

```
main_4:
if i < 10
goto main_6
else goto main_5
```

```
main_5:
r0 <- new Object ; why?
jmp main_3
```

```
main_6:
c <- c + b;
i <- i += 1;
jmp main_4
```

```
main_3:
return
```
Dataflow Analysis

- For any dataflow analysis, we come up with a set of rules that help us determine the desired property after executing particular instructions
  - Examples for liveness forthcoming
- We maintain sets that track the property throughout the analysis
  - e.g., the set of variables that are alive before and after an instruction!
- These sets get propagated around the CFG
  - The resulting sets allow us to conclude certain program properties!
Dataflow Analysis Intuition

1. Think about an interesting program property

2. Think about a set of rules that describe how that property changes between adjacent instructions

3. Think real hard

4. Do something useful with the resulting analysis
Dead code elimination

We use dataflow analysis to assess the liveness property of every variable in the program. Assignments to dead variables do not affect program output and can be removed!
Dead code elimination

- A statement $x := \text{rhs}$ is dead code if $x$ is dead after the assignment
  - You can delete dead code from the program

- You have to determine liveness for a variable to know whether assignments are dead!
  - This is not too bad—liveness is a boolean property

- In fact, we can express liveness in terms of information transferred between adjacent instructions
  - Build up a set of rules to help us determine liveness after each instruction!
  - **Eventually** apply these rules to reason about liveness of variables for blocks throughout the CFG!
Liveness Analysis

- *liveness* is a binary property of a variable that indicates whether the variable will be used at some point in the future.
- You can perform a *liveness analysis* over the program’s instructions for each variable.
- *Liveness analysis* is a type of *dataflow analysis* that tells us which variables are alive at which points of execution.

  - **Intuition:** A variable must have been alive if an instruction refers to the variable. A variable is not alive if an instruction assigns to it without referring to that variable.
Liveness rule 1

For a single instruction, we can say that $x$ is *live* when the instruction refers to $x$ on the rhs

$$\text{\textit{LIVE}}_{\text{in}}(\text{lhs := } x + ...) = \{x\}$$
Liveness rule 2

For a single instruction, we can say that $x$ is \texttt{dead} if the instruction refers to $x$ on the \texttt{lhs}

$$LIVE_{in}(x:=e) = \{\}$$
Liveness rule 3

For a single instruction that does not refer to $x$, the liveness of $x$ is unchanged

$$LIVE_{in}(s) = LIVE_{out}(s)$$
Liveness rule 4

In the case of a Basic Block with multiple children, we must conclude that $x$ is live if any of the children have concluded that $x$ is live!

$$LIVE_{out}(p) = \bigcup_{SUCC(p)} \{LIVE_{in}(s)\}$$
Liveness analysis example

LIVE = {}
x <- 3
B > 0 ?
LIVE = {}
LIVE = {}
y <- z + w
LIVE = {}
LIVE = {}
y <- 0
LIVE = {}
LIVE = {}
x <- x * x
LIVE = {}
x <- 4
A < B
LIVE = {}
LIVE = {}
LIVE = {}
Liveness analysis example

LIVE = {}
x <- 3
LIVE = {}
B > 0 ?
LIVE = {}
LIVE = {}
y <- z + w
LIVE = {}
LIVE = {}
y <- 0
LIVE = {}
LIVE = {}
x <- x * x
LIVE = {x}
x <- 4
LIVE = {}
A < B
LIVE = {}
LIVE = {x}
Liveness analysis example

\[ \text{LIVE} = {} \]

\[ x \leftarrow 3 \quad \text{LIVE} = {} \]
\[ B > 0 ? \quad \text{LIVE} = {} \]
\[ \text{LIVE} = {} \]

\[ y \leftarrow z + w \quad \text{LIVE} = {} \]
\[ \text{LIVE} = {} \]
\[ \text{LIVE} = \{x\} \]

\[ \text{LIVE} = {} \]

\[ y \leftarrow 0 \quad \text{LIVE} = {} \]
\[ \text{LIVE} = {} \]
\[ \text{LIVE} = \{x\} \]

\[ \text{LIVE} = {} \]

\[ x \leftarrow x \times x \quad \text{LIVE} = {} \]
\[ x \leftarrow 4 \quad \text{LIVE} = {} \]
\[ A < B \quad \text{LIVE} = {} \]
\[ \text{LIVE} = {} \]
Liveness analysis example

\[
\begin{align*}
&x \leftarrow 3 \\
&B > 0 ? \\
&y \leftarrow z + w \\
&y \leftarrow 0 \\
&x \leftarrow x \times x \\
&x \leftarrow 4 \\
&A < B
\end{align*}
\]

LIVE = {}
Liveness analysis example

\[
\begin{align*}
&\text{LIVE} = \{\} \\
&x \leftarrow 3 \quad \text{LIVE} = \{\} \\
&B > 0 \quad \text{?} \quad \text{LIVE} = \{\} \\
&\text{LIVE} = \{\} \\
&y \leftarrow z + w \quad \text{LIVE} = \{x\} \\
&\text{LIVE} = \{x\} \\
&y \leftarrow 0 \quad \text{LIVE} = \{x\} \\
&\text{LIVE} = \{x\} \\
&x \leftarrow x \times x \quad \text{LIVE} = \{x\} \\
&x \leftarrow 4 \quad \text{LIVE} = \{\} \\
&A < B \quad \text{LIVE} = \{\} \\
&\text{LIVE} = \{\} \\
&\text{LIVE} = \{\} \\
\end{align*}
\]
Liveness analysis example

LIVE = {}

x <- 3  LIVE = {}
B > 0 ?  LIVE = {}
LIVE = {x}

LIVE = {x}
y <- z + w  LIVE = {x}
LIVE = {x}

LIVE = {x}
y <- 0  LIVE = {x}
LIVE = {x}

LIVE = {x}
x <- x * x  LIVE = {}
x <- 4  LIVE = {}
A < B  LIVE = {}
LIVE = {}
Liveness analysis example

- $x \leftarrow 3$, $\text{LIVE} = \{x\}$
- $B > 0 \ ?$, $\text{LIVE} = \{x\}$
- $y \leftarrow z + w$, $\text{LIVE} = \{x\}$
- $y \leftarrow 0$, $\text{LIVE} = \{x\}$
- $x \leftarrow x \times x$, $\text{LIVE} = \{\}$
- $x \leftarrow 4$, $\text{LIVE} = \{\} $
- $A < B$, $\text{LIVE} = \{\}$
Liveness analysis example

\[
\begin{align*}
x &\leftarrow 3 & \text{LIVE} &= \{\} \\
B &> 0 \ ? & \text{LIVE} &= \{x\} \\
\text{LIVE} &= \{x\} \\
y &\leftarrow z + w & \text{LIVE} &= \{x\} \\
\text{LIVE} &= \{x\} \\
y &\leftarrow 0 & \text{LIVE} &= \{x\} \\
\text{LIVE} &= \{x\} \\
x &\leftarrow x \times x & \text{LIVE} &= \{x\} \\
x &\leftarrow 4 & \text{LIVE} &= \{\} \\
A &< B & \text{LIVE} &= \{\} \\
\text{LIVE} &= \{\} \\
\text{LIVE} &= \{x\} \\
\end{align*}
\]
Liveness analysis example

\[
x \leftarrow 3 \quad \text{LIVE} = \{\}
\]
\[
B > 0 \quad \text{?} \quad \text{LIVE} = \{x\}
\]
\[
\text{LIVE} = \{x\}
\]
\[
y \leftarrow z + w \quad \text{LIVE} = \{x\}
\]
\[
\text{LIVE} = \{x\}
\]
\[
y \leftarrow 0 \quad \text{LIVE} = \{x\}
\]
\[
\text{LIVE} = \{x\}
\]
\[
x \leftarrow x \ast x \quad \text{LIVE} = \{x\}
\]
\[
\text{LIVE} = \{x\}
\]
\[
x \leftarrow 4 \quad \text{LIVE} = \{\}
\]
\[
\text{LIVE} = \{\}
\]
\[
A < B \quad \text{LIVE} = \{x\}
\]
\[
\text{LIVE} = \{x\}
\]

dead code
Liveness analysis example

dead code?

\[
\begin{align*}
x &\leftarrow 3 \\
B &> 0 ? \\
&\text{LIVE} = \{x\}
\end{align*}
\]

\[
\begin{align*}
y &\leftarrow z + w \\
&\text{LIVE} = \{x\}
\end{align*}
\]

\[
\begin{align*}
x &\leftarrow x * x \\
x &\leftarrow 4 \\
A &< B \\
&\text{LIVE} = \{x\}
\end{align*}
\]

dead code
Remarks on Liveness

- You can always reach a **fixed point** because the rules only add variables to liveness sets.

- You only need to change the liveness sets once per sweep per block, so this analysis will terminate.

- Once the analysis is complete, search for assignment instructions where the assigned variable is not live!

- Liveness analysis is **backwards** analysis: you push liveness sets backwards through the CFG.
  - Other analysis can be forward (constant propagation).
Summary for Liveness

- All basic blocks, instructions in CFG initialized with $LIVE_{in} = \emptyset$ and $LIVE_{out} = \emptyset$
- Repeat until $LIVE'_{in} = LIVE_{in}$ for all blocks
- For each block $b$ in CFG
  - $LIVE_{out} = \bigcup_{b' \text{ 's children}} LIVE_{in} (b' \text{'s children})$
  - Starting with $myset = LIVE_{out}$, move backwards through instructions
  - Remove assignee of instruction if present
  - Add operands
  - At top of block, $LIVE'_{in} = myset$
Correctness in Liveness

```
x <- li 3
y <- li 3
```

```
z <- + y 1
```

```
y <- li 2
z <- li 3
```

```
return w
```

What could go wrong?
Correctness

If we want to remove an instruction \( x \leftarrow \text{rhs} \), then we must consider a correctness condition:

On every path following an instruction \( x \leftarrow \text{rhs} \), the variable \( x \) is not alive

Remember: preserving semantics is key!
Constant Propagation

▶ We can also use dataflow analysis to **propagate constant values** throughout the CFG

▶ Similar to liveness analysis, we track **sets** before and after each instruction in each CFG

▶ What would the transfer rules look like for constant propagation?
Constant Propagation rule 1

For a single instruction that does not involve $x$, we can say that the value of $x$ is unknown if it was unknown before that instruction

$$C_{out}(x, s) = \bot \text{ if } C_{in}(x, s) = \bot$$
For a single instruction, we can say that $x$ is $c$ if the instruction refers to $x$ on the lhs and assigns the value $c$

$C_{out}(x,x:=c) = c$ if $c$ is a constant
Constant Propagation rule 3

For function calls, we conservatively assume that we can’t determine what the value is (i.e., $x$ could be many values depending on $f(...)$’s return value

$$C_{out}(x, x := f(...)) = \top$$
Constant Propagation rule 4

For a single instruction that does not refer to $x$, we say the value of $x$ is unchanged.

$C_{out}(x,y := ...) = a$ if $C_{in}(x,y := ...) = a$, $x \neq y$
Constant Propagation rule 5

In the case of a Basic Block with multiple parents, we must conclude that $x$ is $\top$ if any of the parents have concluded that $x$ is $\top$!

If $\exists C_{out}(x, p_i) = \top$, then $C_{in}(x, s) = \top$
In the case of a Basic Block with multiple parents, we must conclude that \( x \) is \( \top \) even if more than one parent has concluded that \( x \) is a constant value.

If \( C_{out}(x, p_i) = c \) and \( C_{out}(x, p_j) = d \) and \( d \neq c \), then \( C_{in}(x, s) = \top \).
In the case of a Basic Block with multiple parents, we must conclude that $x$ is $c$ if the parents have concluded $x = c$ or $x = \perp$.

If $C_{out}(x, p_i) = c$ or $\perp$, then $C_{in}(x, s) = c$. 

Constant Propagation rule 7
In the case of a Basic Block with multiple parents, we must conclude that $x = \bot$ if all the parents conclude $x = \bot$.

If $\forall i. C_{out}(x, p_i) = \bot$, then $C_{in}(x, s) = \bot$. 
Remarks on Constant Propagation

- We use $\bot$ (bottom) to represent a value for which a variable is not currently known (we haven’t seen it yet).
- We use $\{1, 2, 3, ...\}$ to represent the concrete constant values a variable could take if we can conclude a variable takes a constant value.
- We use $\top$ (top) to denote a variable that we cannot conclude takes a single constant value.

- e.g., if ... then $x <- 4$ else $x <- 5$ fi
- We can’t conclude which value $x$ takes!
  - $x = \top$ in this case.
Remarks on Constant Propagation

- Start out by saying $C_{in}(x,s) = \top$ at the entry to the program
- $C_{out}(x,s) = C_{out}(x,s) = \bot$ everywhere else
- Move from top to bottom of each block, update $C$ values accordingly
Constant Propagation example

x = ⊤
x <- 3 x = ⊥
B > 0 ? x = ⊥
x = ⊥

y <- z + w x = ⊥
x = ⊥
y = 0 x = ⊥
x = ⊥

x = ⊥
x <- x * x x = ⊥
x <- 4 x = ⊥
A < B x = ⊥
x = ⊥
Constant Propagation example

\[ x = \top \]
\[ B > 0 \? x = 3 \]
\[ x = 3 \]
\[ x = 3 \]
\[ x = 3 \]
\[ x = \bot \]
\[ y <- 0 x = \bot \]
\[ x = \bot \]
\[ x <- x * x x = \bot \]
\[ x <- 4 x = \bot \]
\[ A < B x = \bot \]
\[ x = \bot \]
\[ x = 0 \]
\[ y <- z + w x = 3 \]
\[ x = 3 \]
Constant Propagation example

\[
\text{\begin{array}{l}
\text{x = } \top \\
\text{x <- 3 \quad x = 3} \\
\text{B > 0 ? \quad x = 3} \\
\text{x = 3} \\
y <- z + w \quad \text{x = 3} \\
\text{x = 3} \\
y <- 0 \quad \text{x = 3} \\
\text{x = 3} \\
x <- x \times x \quad \text{x = 3} \\
x <- 4 \quad \text{x = 3} \\
A < B \\
\text{x = 3} \\
\text{x = 3} \\
\text{x = 3} \\
\end{array}}
\]
Orderings and Lattices

- Constant propagation terminates because values only move from $\perp$ to \{1, 2, 3, ...\} to $\top$. Values cannot move the other way!
  - At most 2 iterations per block to reach $\top$!

- We say that $\perp < c < \top$

```
... -1 0 1 ...

\top
```

Compiler Construction
General Dataflow Analysis

- Dataflow analysis helps us reason about program properties
  - e.g., “variable x is alive”, “variable y is always 4”
- General dataflow analysis often involves several traits:
  - Depends on knowing a property $P$ at a particular point in program execution
  - Proving $P$ at any point requires knowledge of the entire method body
  - $P$ is typically undecidable
- Dataflow Analysis is conservative
Conservatism

- We must maintain correctness
  - Correctness vs. aggressiveness tradeoff
  - Either we definitely know for sure that the property is true
    - “x is definitely 3”
  - Or we don’t know for sure
    - “can’t tell was x is”

- Err on the side of correctness—if you can’t make a definite determination, it’s always OK to know you “don’t know”