Dataflow Analysis and Dataflow-Analysis Frameworks
Roadmap

Last time:
– Data structures (and data) used to determine when it is safe (i.e., sound) to perform an optimizing transformation
  • Review dominators
  • SSA form
  • Dataflow analysis

This time:
– More dataflow analysis
  • Dataflow equations
  • Solving dataflow equations
– Dataflow-analysis frameworks
Dataflow-Analysis Example 1

Reaching definitions

Before $p_1$: $\emptyset$
After $p_1$: $\{<p_1, x>\}$

$p_1$: $x = 1$

... 

Before $p_2$: $\{<p_1, x>, ...\}$
After $p_2$: $\{<p_2, x>, ...\}$

$p_2$: $x = 2$

... 

Before $p_3$: $\{<p_2, x>, ...\}$
After $p_3$: $\{<p_2, x>, <p_3, y>, ...\}$

$p_3$: $y = x$

Transfer function: \[\lambda S. (S - \{< p_i, x >\}) \cup \{< p_2, x >\}\]

Data: sets of $<\text{program-point, variable}>$ pairs

Note: for expository purposes, it is convenient to assume we have a statement-level CFG rather than a basic-block-level CFG.
Dataflow-Analysis Example 1

Reaching definitions

Before p1: \(\emptyset\)
After p1: \(\{<p1, x>\}\)

Before p2: \(\{<p1, x>, ...\}\)
After p2: \(\{<p2, x>, ...\}\)

Before p3: \(\{<p2, x>, <p4, x>, ...\}\)
After p3: \(\{<p2, x>, <p3, y>, <p4, x>, ...\}\)

Meet operation: Union of sets (of <program-point, variable> pairs)

\(p1: x = 1;\)
\[
\vdots
\]
\(p2: x = 2;\)
\[
\vdots
\]
\(p3: y = x;\)
\(p4: x = 7;\)

Before p4: \(\emptyset\)
After p4: \(\{<p4, x>\}\)
Dataflow-Analysis Example 1

Reaching definitions: Why is it useful?
Answers the question “Where could this variable have been defined?”

Before p1: Ø
After p1: {<p1, x>}

p1: x = 1;

... 

Before p2: {<p1, x>, ...}
After p2: {<p2, x>, ...}

Before p3: {<p2, x>, <p4, x> ...}
After p3: {<p2, x>, <p3, y>, <p4, x>, ...}

Before p4: Ø
After p4: {<p4, x>}

p2: x = 2;

... 

p3: y = x;

p4: x = 7;
Dataflow-Analysis Example 2

Live Variables

Before p1: $\emptyset$
After p1: $\{x\}$

Before p2: $\{x\}$
After p2: $\{x, y\}$

Before p3: $\{x, y\}$
After p3: $\emptyset$

Before p4: $\emptyset$
After p4: $\{x\}$

Before p5: $\{x\}$
After p5: $\{x\}$

Before p6: $\{x\}$
After p6: $\emptyset$

Transfer function:

$\lambda S. (S - \{z\}) \cup \{x, y\}$

Data: sets of variables

z is not live after p5, and thus p5 is a useless assignment (= dead code)

Dataflow Analysis Example 2

Live Variables

Before p1: $\emptyset$
After p1: $\{x\}$

Before p2: $\{x\}$
After p2: $\{x, y\}$

Before p3: $\{x, y\}$
After p3: $\emptyset$

Before p4: $\emptyset$
After p4: $\{x\}$

Before p5: $\{x\}$
After p5: $\{x\}$

Before p6: $\{x\}$
After p6: $\emptyset$

Transfer function:

$\lambda S. (S - \{z\}) \cup \{x, y\}$

Data: sets of variables

z is not live after p5, and thus p5 is a useless assignment (= dead code)
Dataflow-Analysis Direction

Forward analysis
– Start at the beginning of a function’s CFG, work along the control edges (e.g., reaching definitions)

Backward analysis
– Start at the end of a function’s CFG, work against the control edges (e.g., live variables)
There are some obvious advantages to this format for program analysis:

- **Easy to see the live range of a given variable** \( x \) assigned to in statement \( s \):
  - The region from "\( x = \ldots; \)" until the last use(s) of \( x \) before \( x \) is redefined
  - In SSA form, from "\( x_i = \ldots; \)" to all uses of \( x_i \), e.g., "\( \ldots = f(\ldots, x_i, \ldots); \)"

- **Easy to see when an assignment is useless**
  - We have "\( x_i = \ldots; \)" and there are no uses of \( x_i \) in any expression or assignment RHS
  - "\( x_i = \ldots; \)" is a useless assignment
  - "\( x_i = \ldots; \)" is dead code

In other words, some useful information is pre-computed, or at least easily recoverable from SSA form.

**Warning 1:** Dead code = useless assignments + unreachable code

**Warning 2:** There is another concept called “live variables.”
- When variable \( x \) is “not live,” a convenient shorthand is “Variable \( x \) is dead.”
- When \( x \) is dead just after a statement \( s \), that does not imply that \( s \) is dead code. (E.g., suppose \( s \) assigns to \( y \).)
- When \( s \) is a useless assignment to \( x \)
  - Statement \( s \) is dead code (because dead = useless or unreachable)
  - \( x \) is not live just after \( s \) (“Variable \( x \) is dead just after \( s \)”)  
  - Because variable \( x \) is dead, \( s \) is a useless assignment, and thus statement \( s \) is dead code.
Dataflow-Analysis Example 3

Reachable uses

Before p1: {...}  
After p1: {<p3, z>, <p2, z>, ...}  
p1: z = 1;  
   ...  

Before p2: {<p3, z>, <p2, z>, ...}  
After p2: {<p3, x>, <p3, z>, ...}  
p2: x = z;  

Before p3: {<p3, x>, <p3, z>}  
After p3: ∅  
p3: y = x+z;  

Transfer function:  
\[ \lambda S. (S - \{< p_1, x >\}) \cup \{< p_2, z >\} \]  

Data: sets of <program-point, variable> pairs
Dataflow-Analysis Example 3

Reachable uses

Before p1: \{<p3, z>, <p2, z>, <p4, z>, \ldots\}
After p1: \{<p3, z>, <p2, z>, <p4, z>, \ldots\}

Before p2: \{<p3, z>, <p2, z>, \ldots\}
After p2: \{<p3, x>, <p3, z>, \ldots\}

Before p3: \{<p3, x>, <p3, z>\}
After p3: \emptyset

p1: if (…) …

p2: x = z; …

p3: y = x+z;

p4: x = 3*z; Before p4: \{<p3, z>, <p4, z>\}
After p4: \{<p3, x>, <p3, z>\}

Meet operation: Union of sets (of \(<program-point, variable>\) pairs)
Dataflow-Analysis Example 3

 reachable uses: Why is it useful?

 Answers the question “What could this variable definition reach?”

 After p0: \(<p3, z>, <p2, z>, <p4, z>, \ldots\> \]

 Before p1: \(<p3, z>, <p2, z>, <p4, z>, \ldots\> \]

 After p1: \(<p3, z>, <p2, z>, <p4, z>, \ldots\> \]

 Before p2: \(<p3, z>, <p2, z>, \ldots\> \]

 After p2: \(<p3, x>, <p3, z>, \ldots\> \]

 Before p3: \(<p3, x>, <p3, z>\> \]

 After p3: \(\emptyset\)
Dataflow-Analysis Example 3

Reachable uses: Why is it useful?

Answers the question “What could this variable definition reach?”

Before p0: \{<p3, z>, <p2, z>, <p4, z>, ...\}
After p0: \{<p3, z>, <p2, z>, <p4, z>, ...\}

Before p1: \{<p3, z>, <p2, z>, <p4, z>, ...\}
After p1: \{<p3, z>, <p2, z>, <p4, z>, ...\}

Before p2: \{<p3, z>, <p2, z>, ...\}
After p2: \{<p3, x>, <p3, z>, ...\}

Before p3: \{<p3, x>, <p3, z>\}
After p3: \emptyset

Reachable uses: really just an indexing question. At which end of the edges do you want to collect the information?
Obtaining a Dataflow-Analysis Solution

Successive approximation:

– Assign to each node in the CFG a (dataflow-problem-specific) default value
  • Typically either $\emptyset$ or the universe of the sets you are working with, e.g., {all variables in the procedure}
– Assign a special value to the entry node
– Propagate values until quiescence, as follows:
  Repeatedly
    • Pick a node
    • Find input values from predecessors
    • Apply transfer function
  Until no change is possible
Example: Reaching Definitions

Before p1: ∅  
After p1: ∅  

Before p2: ∅  
After p2: {<p2,min>}

Before p3: {<p2,min>}
After p3: {<p2,min>, <p3,x>}

Before p4: {<p2,min>, <p3,x>}
After p4: {<p2,min>, <p3,x>}

Before p5: {<p2,min>, <p3,x>}
After p5: {<p2,min>, <p3,x>}

Before p6: {<p2,min>, <p3,x>}
After p6: {<p6,min>, <p3,x>}

Before p7: {<p2,min>, <p6,min>, <p3,x>}
After p7: {<p2,min>, <p6,min>, <p7,x>}
Example: Reaching Definitions

Before p1: ∅
After p1: ∅

Before p2: ∅
After p2: {<p2, min>}

Before p3: {<p2, min>}
After p3: {<p2, min>, <p3, x>}

Before p4: {<p2, min>, <p3, x>}
After p4: {<p2, min>, <p3, x>, <p6, min>, <p7, x>}

Before p5: {<p2, min>, <p3, x>, <p6, min>, <p7, x>}
After p5: {<p2, min>, <p3, x>, <p6, min>, <p7, x>}

Before p6: {<p2, min>, <p3, x>, <p6, min>, <p7, x>}
After p6: {<p2, min>, <p3, x>, <p6, min>, <p7, x>}

Before p7: {<p2, min>, <p3, x>, <p6, min>, <p7, x>}
After p7: {<p2, min>, <p3, x>, <p6, min>, <p7, x>}

Before p8: {<p2, min>, <p3, x>, <p6, min>, <p7, x>}
After p8: {<p2, min>, <p3, x>, <p6, min>, <p7, x>}

Before p9: {<p2, min>, <p3, x>, <p6, min>, <p7, x>}
After p9: {<p2, min>, <p3, x>, <p6, min>, <p7, x>}

1: start
2: min = +∞
3: read(x)
4: while x>0
5: if x<min
6: min = x
7: read(x)
8: write(min)
9: end
Obtaining a Dataflow-Analysis Solution by Successive Approximation

for all nodes $n$, $\text{RdBefore}[n] := \emptyset$ and $\text{RdAfter}[n] := \emptyset$
workset := \{ start\}
while (workset $\neq$ $\emptyset$) {
    select and remove a node $n$ from workset
    oldValueAfter := $\text{RdAfter}[n]$
    $\text{RdBefore}[n] := \bigcup_{<p,n> \in Edges} \text{RdAfter}[p]$
    $\text{RdAfter}[n] := F_n(\text{RdBefore}[n])$
    if oldValueAfter $\neq$ $\text{RdAfter}[n]$ then
        for all $<n,w> \in Edges$, insert $w$ into workset
}
Successive Approximation!? Does That Always Work?

To find a solution $x^* = F(x^*)$, perform $x_{k+1} = F(x_k)$

Let’s try: $x^2 = 2$, using $x = \frac{2}{x}$

Iterate on $x_{k+1} = \frac{2}{x_k}$

Pick any $x_0 \neq 0$,

$x_1 = \frac{2}{x_0}, x_2 = x_0, x_3 = \frac{2}{x_0}, x_4 = x_0$, failure 😞
Successive Approximation!? Does That Always Work?

To find a solution \( x^* = F(x^*) \), perform \( x_{k+1} = F(x_k) \)

\[ x^2 = 2, \text{ so } x = \frac{2}{x} \]

Add \( x \) to both sides: \( x + x = x + \frac{2}{x} \) That is, \( 2x = x + \frac{2}{x} \)

Iterate on \( x_{k+1} = \frac{1}{2} \left( x_k + \frac{2}{x_k} \right) \)

\[
\begin{align*}
    x_0 & = 1.00000 \\
    x_1 & = 1.50000 \\
    x_2 & = 1.41666 \\
    x_3 & = 1.41421 \\
    x_4 & = 1.41421
\end{align*}
\]
Iterative method converges

Iterative method diverges

**Theorem 2.1** Let \( g(x) \) be an iteration function satisfying Assumptions 2.1 and 2.3. Then \( g(x) \) has exactly one fixed point \( \xi \) in \( I \), and starting with any point \( x_0 \) in \( I \), the sequence \( x_1, x_2, \ldots \) generated by fixed-point iteration of Algorithm 2.6 converges to \( \xi \).

To prove this theorem, recall that we have already proved the existence of a fixed point \( \xi \) for \( g(x) \) in \( I \). Now let \( x_0 \) be any point in \( I \). Then, as we remarked earlier, fixed-point iteration generates a sequence \( x_1, x_2, \ldots \) of points all lying in \( I \), by Assumption 2.1. Denote the error in the \( n \)th iterate by

\[
e_n = \xi - x_n \quad n = 0, 1, 2, \ldots
\]

Then since \( \xi = g(\xi) \) and \( x_n = g(x_{n-1}) \), we have

\[
e_n = \xi - x_n - g(\xi) = g(x_{n-1}) - g(x_n) - g'(x_n)\epsilon_{n-1}
\]

(2.19)
Successive Approximation!?
Does That Always Work?

To find a solution $x^* = F(x^*)$, perform $x_{k+1} = F(x_k)$

- Fact: For reaching definitions and live variables, successive approximation always works
- Why?
  - (An approximation to) an answer is two sets per program point
  - The sets at each program point are finite and of a priori bounded size
  - Each sets always increases in size ($\subseteq$)
  - Approximations to answers get bigger and bigger, but cannot grow without bound
    - Therefore the algorithm must terminate
    - When the algorithm terminates, the sets solve the equations
Equations? What Equations?

Two equations for each node $n$:

\[
\begin{align*}
RdBefore[n] &= \bigcup_{<p,n>\in Edges} RdAfter[p] \\
RdAfter[n] &= F_n(RdBefore[n])
\end{align*}
\]

Successive approximation:

\[
\begin{align*}
RdBefore_{k+1}[n] &= \bigcup_{<p,n>\in Edges} RdAfter_k[p] \\
RdAfter_{k+1}[n] &= F_n(RdBefore_k[n])
\end{align*}
\]

In iterative algorithm:

\[
\begin{align*}
RdBefore[n] &:= \bigcup_{<p,n>\in Edges} RdAfter[p] \\
RdAfter[n] &:= F_n(RdBefore[n])
\end{align*}
\]
Equations: What Equations?

Equations:
\[ x = 3y + 4z \]
\[ y = 2w + 2 \]
\[ z = 7w - x \]
\[ w = 17 \]
DATAFLOW-ANALYSIS FRAMEWORKS
What is a Dataflow Framework?

Many analyses can be formulated in terms of how data is transformed over the control flow graph

– Propagate information from:
  • After (before) some node, to
  • Before (after) some other node

– Put information together when control flow merges (or diverges)

A framework captures these uniformities

– In object-oriented-program terms: like an abstract class AC

– To use the framework
  • You define certain data and methods (required by AC)
  • AC supplies other methods (already implemented, so you don’t have to worry about implementing them yourself)
Dataflow Framework: What You Supply

The type of data (a.k.a. dataflow facts)
– A collection of values with an order, such as \( \subseteq \)
– (Sometimes called a “meet semi-lattice”)
– Default value and value to use at entry (or exit)

Transfer functions
– Specify how data is propagated across a node

A meet operation (\( \sqcap \))
– The operation for combining values that come across multiple edges

Direction (forward or backward)
Dataflow Framework Instantiated for Reaching-Definitions Analysis

The type of data (a.k.a. dataflow facts):

Sets of <program-point, variable> pairs

Transfer functions:

For “p: id = exp;” and “p: read id”
\[ \lambda S. (S - \{< p_i, id >\}) \cup \{< p, id >\} \]

For “if exp ...” and “write exp
\[ \lambda S. S \]

The meet operation (for combining values that come across multiple edges):

Set union (\( \cup \))

Direction:

Forward
Dataflow Framework Instantiated for Live-Variable Analysis

The type of data (a.k.a. dataflow facts):

Sets of variables

Transfer functions:

For “id = exp;”
\[ \lambda S. (S - \{id\}) \cup \{x \in exp\} \]

For “if exp”, and “write exp”
\[ \lambda S. S \cup \{x \in exp\} \]

For “read id”
\[ \lambda S. (S - \{id\}) \]

The meet operation (for combining values that come across multiple edges):

Set union (\(\cup\))

Direction:

Backward
Obtaining a Dataflow-Analysis Solution by Successive Approximation

for all nodes n, ValBefore[n] := \text{T} and ValAfter[n] := \text{T}
workset := \{\text{start}\}
while (workset \neq \emptyset) {
    select and remove a node n from workset
    oldValueAfter := ValAfter[n]
    ValBefore[n] := \prod_{<p,n> \in Edges} ValAfter[p]
    ValAfter[n] := F_n(ValBefore[n])
    if oldValueAfter \neq ValAfter[n] then
        for all <n, w> \in Edges, insert w into workset
}
Obtaining a Dataflow-Analysis Solution by Successive Approximation

for all nodes $n$, $ValAfter[n] := T$ and $ValBefore[n] := T$

workset := \{ end\}

while (workset $\neq \emptyset$) {
    select and remove a node $n$ from workset

    oldValueBefore := $ValBefore[n]$

    $ValAfter[n] := \prod_{<n,p> \in Edges} ValBefore[p]$

    $ValBefore[n] := F_n(ValAfter[n])$

    if oldValueBefore $\neq ValBefore[n]$ then
        for all $<w, n> \in Edges$, insert $w$ into workset
    }

Dataflow-Analysis Example 3

Available-expressions analysis
- Whether an expression that has been previously computed may be reused
- Forward dataflow problem: from expression to points of re-use
- Meet semi-lattice:

```
\text{True}
\downarrow
\text{False}
```

- Meet operation:
  - AND of all predecessors
- At the beginning of each block, everything is True
  * This causes some problems for loops
Dataflow-Analysis Example 4

Very-Busy-Expression analysis
– An expression is very busy at a point \( p \) if it is guaranteed that it will be computed at some time in the future

– Backwards dataflow problem: from computation to use
– Meet Lattice:

```
True
|
False
```

– Meet operation: AND
The end: or is it?

Covered a broad range of topics
– Some formal concepts
– Some practical concepts

What we skipped
– Linking and loading
– Interpreters
– Register allocation
– Performance analysis / Proofs