Code analysis and transformation

DFA foundation

Simone Campanoni
simone.campanoni@northwestern.edu
We have seen several examples of DFAs

• Are they correct?

• Are they precise?

• Will they always terminate?

• How long will they take to converge?
Outline

• Lattice and data-flow analysis

• DFA correctness

• DFA precision

• DFA complexity
Understanding DFAs

• We need to understand all of them
  • Liveness analysis: is it correct? Precision? Convergence?
  • Reaching definitions: is it correct? Precision? Convergence?
  • ...

• Idea: create a framework to help reasoning about them
  • Provide a single formal model that describes all data-flow analyses
  • Formalize the notions of “correctness,” “conservativeness,” and “optimality”
  • Correctness proof for DFAs
  • Place bounds on the time complexity of iterative DFAs
  • This is not to drive the implementation, but to reason about data-flow equations
Lattice

- Lattice $L = (V, \leq)$:
  - $V$ is a (possible infinite) set of elements
  - $\leq$ is a binary relation over elements of $V$
- Lower bound
  - $z$ is a lower bound of $x$ and $y$ iff $z \leq x$ and $z \leq y$
- Upper bound
  - $z$ is a upper bound of $x$ and $y$ iff $x \leq z$ and $y \leq z$
- Operations: meet ($\wedge$) and join ($\vee$)
  - $b \vee c$: least upper bound
  - $b \wedge c$: greater lower bound
Lattice

• Lattice $L = (V, \leq)$:
  • $V$ is a (possible infinite) set of elements
  • $\leq$ is a binary relation over elements of $V$

• Lower bound
  • $z$ is a lower bound of $x$ and $y$ iff $z \leq x$ and $z \leq y$

• Upper bound
  • $z$ is a upper bound of $x$ and $y$ iff $x \leq z$ and $y \leq z$

• Operations: meet ($\land$) and join ($\lor$)
  • $b \lor c$: least upper bound
  • $b \land c$: greater lower bound
  • An useful property: if $e \leq b$ and $e \leq c$, then $e \leq b \land c$
Lattice

• Lattice $L = (V, \leq)$:
  • $V$ is a (possible infinite) set of elements
  • $\leq$ is a binary relation over elements of $V$

• Properties of $\leq$:
  • $\leq$ is a partial order (reflexive, transitive, anti-symmetric)
  • Every pair of elements in $V$ has
    • An unique greatest lower bound (a.k.a. meet) and
    • An unique least upper bound (a.k.a. join)

• Top ($T$) = unique greatest element of $V$ (if it exists)
• Bottom ($\bot$) = unique least element of $V$ (if it exists)
• Height of $L$: longest path from $T$ to $\bot$
  • Infinite large lattice can still have finite height

If you know nothing, this is still a correct, but conservative, solution
Lattice and DFA

- A lattice $L = (V, \leq)$ describes all possible solutions of a given DFA
  - A lattice for reaching definitions
  - Another lattice for liveness analysis
  - ...
  - For DFAs that look for solutions per point in the CFG:
    one “lattice instance” per point

- The relation $\leq$ connects all solutions of its related DFA from the best one ($T$) to the worst one --most conservative one--($\perp$)
  - Liveness analysis: variables that might be used after a given point in the CFG
    $T = \text{no variable is alive} = \{\}$
    $\perp = \text{all variables are alive} = V$

Why? $\perp$ = all variables are alive = $V$

- To solve a data-flow analysis: we traverse the lattice of a given DFA to find the correct solution in a given point of the CFG
  - We repeat it for every point in the CFG
Lattice example

• How many apples I must have?
• \( V \) = sets of apples
• \( \leq \) = set inclusion
  \[ \{\text{apple}\} \leq \{\text{apple}, \text{apple}\} \]
• \( T \) = (best case) = all apples
• \( \bot \) = (worst case) no apples (empty set)

Apples, definitions, variables, expressions ...
Another lattice example

• How many apples I may have?
• $V = \text{sets of apples}$
• $\leq = \text{set inclusion}$
  $$\{\ \}, \{\ \} \leq \{\ \}$$
• $T = \text{no apples (empty set)}$
• $\bot = \text{(most conservative) all apples}$
How can we use this mathematical framework, lattice, to study a DFA?
Use of lattice for DFA

- Define domain of program properties (flow values --- apple sets) computed by data-flow analysis, and organize the domain of elements as a lattice.

- Define how to traverse this domain to compute the final solution using lattice operations.

- Exploit lattice theory in achieving goals.
Data-flow analysis and lattice

• Elements of the lattice (V) represent flow values (e.g., an IN[] set)
  • e.g., Sets of apples
Data-flow analysis and lattice

- Elements of the lattice (V) represent flow values (e.g., an IN[] set)
  - *e.g.*, Sets of live variables for liveness
- \( \perp \) “worst-case” information
  - *e.g.*, Universal set
- \( T \) “best-case” information
  - *e.g.*, Empty set
- If \( x \leq y \), then \( x \) is a conservative approximation of \( y \)
  - *e.g.*, Superset
Data-flow analysis and lattice (reaching defs)

• Elements of the lattice (V) represent flow values (IN[], OUT[])
  • *e.g.*, Sets of definitions

• T represents “best-case” information
  • *e.g.*, Empty set

• ⊥ represents “worst-case” information
  • *e.g.*, Universal set

• If \( x \leq y \), then \( x \) is a conservative approximation of \( y \)
  • *e.g.*, Superset
How do we choose which element in our lattice is the data-flow value of a given point of the input program?
We traverse the lattice

How many apples I must have?

We found out there is no guarantee we have the green apple
We traverse the lattice

for (each instruction $i$ other than ENTRY) $\text{OUT}[i] = \{ \}$;

\[ T = \{ \} \]

\[ \{ d_1 \} \]
\[ \{ d_2 \} \]
\[ \{ d_3 \} \]
\[ \{ d_1, d_2 \} \]
\[ \{ d_1, d_3 \} \]
\[ \{ d_2, d_3 \} \]
\[ \bot = \{ d_1, d_2, d_3 \} \]

• New information discovered while computing the IN/OUT sets will bring us down in the lattice
• New information is merged into the current knowledge/state/current-point-in-the-lattice

...let’s see how
Merging information

• New information is found
  • e.g., a new definition (d1) reaches a given point in the CFG

• New information is described as a point in the lattice
  • e.g. \{d1\}

• We use the “meet” operator (\wedge) of the lattice to merge the new information with the current one
  • e.g., set union
  • Current information: \{d2\}
  • New information: \{d1\}
  • Result: \{d1\} U \{d2\} = \{d1, d2\}
We traverse the lattice

As long as we know how to get new information, then we know how to traverse the lattice to converge to the final solution.

We discover:
- a new definition, $d_1$, reaches our point in the CFG
  - New fact = $\{d_1\}$
  - $\emptyset \land \{d_1\} = \{d_1\}$

We traverse the lattice:

\[ T = \{ \} \]

\[ \{ d_1 \} \quad \{ d_2 \} \quad \{ d_3 \} \]

\[ \{d_1,d_2\} \quad \{d_1,d_3\} \quad \{d_2,d_3\} \]

\[ \perp = \{d_1,d_2,d_3\} \]
How can we find new facts/information?
Computing a data-flow value (ideal)

- For a forward problem, consider all possible paths from the entry to a given program point, compute the flow values at the end of each path, and then meet these values together.

- Meet-over-all-paths (MOP) solution at each program point
  - It’s a correct solution
Computing MOP solution for reaching definitions
The problem of ideal solution

• **Problem**: all preceding paths must be analyzed
  • Exponential blow-up
• To compute the MOP solution in BB2:
  0-1-A, 1-2-A
  0-1-A, 1-2-B
  0-1-B, 1-2-A
  0-1-B, 1-2-B
From ideal to practical solution

• **Problem**: all preceding paths must be analyzed
  • Exponential blow-up

• **Solution**: compute meets early (at merge points) rather than at the end
  • Maximum fixed-point (MFP)

\[ \text{IN}[i] = \bigcup_{p \text{ a predecessor of } i} \text{OUT}[p]; \]

• **Questions**:
  • Is MFP correct?
  • What’s the precision of MFP?
Outline

• Lattice and data-flow analysis

• DFA correctness

• DFA precision

• DFA complexity
Correctness

$V_{\text{correct}} \leq V_{\text{MOP}}$

$T = \{ \}$

$\{ d1 \}$
$\{ d2 \}$
$\{ d3 \}$

$\{ d1,d2 \}$
$\{ d1,d3 \}$
$\{ d2,d3 \}$

$\bot = \{ d1,d2,d3 \}$
Correctness

• Key idea:
  • “Is MFP correct?” iff $V_{\text{MFP}} \leq V_{\text{MOP}}$

• Both start from $V_{\text{start}}$
  • $V_{p1} = f_{s_{p1}}(V_{\text{start}})$
  • $V_{p2} = f_{s_{p2}}(V_{\text{start}})$
  • $V_{\text{MOP}} = f_{s_{p3}}(V_{p1}) \land f_{s_{p3}}(V_{p2})$
**Correctness**  
*fs is monotonic => MFP is correct!*

- **Key idea:**
  - “Is MFP correct?” iff $V_{MFP} \leq V_{MOP}$

- **Focus on merges:**

  $\text{IN}[i] = \bigcup_{p \text{ a predecessor of } i} \text{OUT}[p]$;

  - $V_{MOP} = fs_{p3}(V_{p1}) \land fs_{p3}(V_{p2})$
  - $V_{MFP} = fs_{p3}(V_{p1} \land V_{p2})$
  - $V_{MFP} \leq V_{MOP}$ iff $fs_{p3}(V_{p1} \land V_{p2}) \leq fs_{p3}(V_{p1}) \land fs_{p3}(V_{p2})$

- **If fs is monotonic: X ≤ Y then fs(X) ≤ fs(Y)**
  - $(V_{p1} \land V_{p2}) \leq V_{p1}$ by definition of meet
  - $(V_{p1} \land V_{p2}) \leq V_{p2}$ by definition of meet
  - So $fs_{p3}(V_{p1} \land V_{p2}) \leq fs_{p3}(V_{p1})$ and $fs_{p3}(V_{p1} \land V_{p2}) \leq fs_{p3}(V_{p2})$
  - Therefore $fs(V_{p1} \land V_{p2}) \leq fs(V_{p1}) \land fs(V_{p2})$
  - And therefore $V_{MFP} \leq V_{MOP}$

---

An useful property: if $e \leq b$ and $e \leq c$, then $e \leq b \land c$
Monotonicity

• $X \leq Y$ then $f_s(X) \leq f_s(Y)$

• If the flow function $f$ is applied to two members of $V$, the result of applying $f$ to the “lesser” of the two members will be under the result of applying $f$ to the “greater” of the two

• More conservative inputs leads to more conservative outputs (never more optimistic outputs)
Convergence

• **From lattice theory**
  If $fs$ is monotonic,
  then the maximum number of times $fs$ can be applied
  w/o reaching a fixed point is $\text{Height}(V) - 1$

• Iterative DFA is guaranteed to terminate
  if the $fs$ is monotonic and
  the lattice has finite height
Outline

• Lattice and data-flow analysis
• DFA correctness
• DFA precision
• DFA complexity
Precision

• $V_{MOP}$: the best solution
• $V_{MFP} \leq V_{MOP}$
  • $fs(V_{p1} \land V_{p2}) \leq fs(V_{p1}) \land fs(V_{p2})$
• Distributive $fs$ over $\land$
  • $fs(V_{p1} \land V_{p2}) = fs(V_{p1}) \land fs(V_{p2})$
  • $V_{MFP} = V_{MOP}$
• Is reaching definition $fs$ distributive?
  • (did having performed $\land$ earlier in the CFG change anything?)
A new DFA example: reaching constants

• Goal
  • Compute the value that a variable must have at a program point (no SSA)

• Flow values (V)
  • Set of (variable,constant) pairs

• Merge function
  • Intersection

• Data-flow equations
  • Effect of node $n$: $x = c$
    • $\text{KILL}[n] = \{(x,k) \mid \forall k\}$
    • $\text{GEN}[n] = \{(x,c)\}$
  • Effect of node $n$: $x = y + z$
    • $\text{KILL}[n] = \{(x,k) \mid \forall k\}$
    • $\text{GEN}[n] = \{(x,c) \mid c = \text{val}_y + \text{val}_z, (y, \text{val}_y) \in \text{IN}[n], (z, \text{val}_z) \in \text{IN}[n]\}$
Reaching constants: characteristics

• ⊥ = ?
• IN = ?
• OUT = ?
• Let’s study this analysis
  • Does it convergence?
    • is $fs$ monotonic? Has the lattice a finite height?
  • What is the precision of the solution?
    • is $fs$ distributive?
Outline

• Lattice and data-flow analysis
• DFA correctness
• DFA precision
• DFA complexity
Complexity

\[
\text{OUT}[\text{ENTRY}] = \{ \}; \\
\text{for (each instruction } i \text{ other than ENTRY) } \text{OUT}[i] = \{ \}; \\
d\{ \\
\text{for (each instruction } i \text{ other than ENTRY) } \{ \\
\text{IN}[i] = \bigcup_{p \text{ a predecessor of } i} \text{OUT}[p]; \\
\text{OUT}[i] = \text{GEN}[i] \cup (\text{IN}[i] - \text{KILL}[i]); \\
\} \\
\} \text{ while (changes to any OUT occur);} 
\]
Complexity

• N instructions (N definitions at most)
• Complexity of the computation of IN[i]
  • Each IN/OUT set has at most N elements
  • Each set-union operation takes O(N) time
• Constant number of set operations per instruction
  • The computation of IN[i] is O(N) time

OUT[ENTRY] = { }; for (each instruction i other than ENTRY) OUT[i] = { };
do {
  for (each instruction i other than ENTRY) {
    IN[i] = \bigcup p \text{ a predecessor of } i \text{ OUT}[p];
    OUT[i] = GEN[i] \cup (IN[i] - KILL[i]);
  }
} while (changes to any OUT occur);
Complexity

- N instructions (N definitions at most)
- Complexity of the computation of IN[i]
  - O(N)
- Complexity of an iteration
  - Constant number of set operations per iteration
  - O(N)

```
OUT[ENTRY] = { };
for (each instruction i other than ENTRY) OUT[i] = { };
do {
  for (each instruction i other than ENTRY) {
    IN[i] = \bigcup_{p \text{ a predecessor of } i} OUT[p];
    OUT[i] = GEN[i] \cup (IN[i] \setminus KILL[i]);
  }
} while (changes to any OUT occur);
```
Complexity

• N instructions (N definitions at most)
• Complexity of the computation of IN[i]
  • O(N)
• Complexity of an iteration
  • O(N)
• Complexity of an invocation
  • O(N) instructions ⇒ O(N^2) time per invocation of the loop

```plaintext
OUT[ENTRY] = { }; for (each instruction i other than ENTRY) OUT[i] = { }; do {
  for (each instruction i other than ENTRY) {
    IN[i] = \bigcup p \text{ a predecessor of } i \text{ OUT}[p];
    OUT[i] = GEN[i] \cup (IN[i] \setminus KILL[i]);
  }
} while (changes to any OUT occur);
```
Complexity

• N instructions (N definitions at most)
  • Complexity of the computation of IN[i]
    • O(N)
  • Complexity of an iteration
    • O(N)
  • Complexity of an invocation
    • O(N^2)
  • Complexity of do-while
    • Each do-while iteration modifies in the worst case only one set
    • Each modification can only add one element in the worst case
    • So the computation of a single set can take up to O(N^3)
    • There are N sets: O(N^4)

```plaintext
OUT[ENTRY] = { };
for (each instruction i other than ENTRY) OUT[i] = { };
do {
  for (each instruction i other than ENTRY) {
    IN[i] = ∪_p a predecessor of i OUT[p];
    OUT[i] = GEN[i] ∪ (IN[i] ─ KILL[i]);
  }
} while (changes to any OUT occur);
```
Complexity

- N instructions (N definitions at most)
- Complexity of the computation of IN[i]
  - O(N)
- Complexity of an iteration
  - O(N)
- Complexity of an invocation
  - O(N²)
- Complexity of do-while
  - O(N⁴)

```plaintext
OUT[ENTRY] = { };  
for (each instruction i other than ENTRY) OUT[i] = { };  
do {  
  for (each instruction i other than ENTRY) {  
    IN[i] = \( \bigcup \) p a predecessor of i OUT[p];  
    OUT[i] = GEN[i] \( \cup \) (IN[i] ─ KILL[i]);  
  }  
} while (changes to any OUT occur);
```
Complexity

• N instructions (N definitions at most)
• Complexity of do-while
  • $O(N^4)$

```
OUT[ENTRY] = { };  
for (each instruction i other than ENTRY) OUT[i] = { };  
do {  
  for (each instruction i other than ENTRY) {  
    IN[i] = $\cup p$ a predecessor of i OUT[p];  
    OUT[i] = GEN[i] $\cup$ (IN[i] ─ KILL[i]);  
  }  
} while (changes to any OUT occur);  
```
Complexity

- N instructions (N definitions at most)
- Complexity of do-while
  - Worst case: $O(N^4)$

- Typical case: 2 to 3 invocations with good ordering, work-list, basic-block, and sparse sets
  - Between N and $N^2$

```
N=500
Worst case: 62,500,000,000
Optimized average case: 500 – 250,000
```
Always have faith in your ability

Success will come your way eventually

Best of luck!