Code analysis and transformation

DFA

Simone Campanoni
simonec@eecs.northwestern.edu
Data Flow Analysis outline

• Why do we need DFA?

• Introduction to DFA

• A DFA example: reaching definitions

• Implementation of DFA
Constant propagation

• Find an instruction $i$ that defines a variable with a constant expression
  
  *Instruction* $i$: $b = \text{CONSTANT\_EXPRESSION}$

• Replace an use of $b$ in an instruction $j$ with that $\text{CONSTANT\_EXPRESSION}$ if
  • All control flows to $j$ includes $\text{CFA}$
  • There are no intervening definition of that variable $\text{DFA}$

![Diagram showing constant propagation]

- $i$: $b = 2$
- $j$: $\ldots = b$
- $i$: $b = 2$
- $j$: $\ldots = 2$
Constant propagation: code example

```c
int sumcalc (int a, int b, int N){
    int x, y;
    x = 0;
    y = 0;
    if (a > b){
        x = x + N;
    }
    if (b > N){ return y;}
    return x;
}
```

Data-flow analysis is a collection of techniques for compile-time reasoning about the run-time values.

We need to know the “data-flow” of the program.
Understanding the data-flow requires understanding the control-flow.
But constant propagation (CP) has been done already ...

- CP has been already designed and implemented

- Why should we study it? Why don’t we design and implement all possible transformations and analyses in a compiler and move on?

- It is always possible to invent new/better transformations

  **Full employment theorem for compiler writers**
New transformations and analyses

- New transformations (often) need to understand specific and new code properties related to how data might change through the code
  - So we need to know how to design a new data flow analysis that identifies these new code properties
- Generic recipe
  
  **Data flow analysis (DFA):**
  
  traverse the CFGs collecting information about what may happen at run time (Conservative approximation)

  **Transformation:**
  
  Modify the code based on the result of data flow analysis (Correctness guaranteed by the conservative approximation of DFA)
Data Flow Analysis outline

• Why do we need DFA?

• Introduction to DFA

• A DFA example: reaching definitions

• Implementation of DFA
Static program vs. dynamic execution

• **Static:**
  Finite program

• **Dynamic:**
  Can have infinitely many possible execution paths

• **Data flow analysis abstraction:**
  For each point in a program:
  combine information about all possible run-time instances of the same program point.

What are the possible values of b?
Example of data-flow questions

• What are the possible values of b just before an instruction “... = b”?
• Which instruction defines the value used in “... = b”?
• Has the expression “a * b” been computed before another instruction? (“... = a * b”)
• What are the instructions that might read the value produced by an instruction “b = ...”?
• What are the instructions that will (must) read the value produced by an instruction “b = ...”?
• ...


Data-flow expressed in CFG

Data-flow value:
set of all possible program states that can be observed at a given program point

e.g., all definitions in the program that might have been executed before that point

```
int x, y
x = 0
y = 0
If (a > b)
    x = x + N
If (b > N)
    return y
return x
```
**Data-flow expressed in CFG**

**Data-flow value:**
set of all possible program states that can be observed at a given program point

e.g., all definitions in the program that might have been executed before that point

**Data-flow analysis**
computes IN and OUT sets by computing the DFA-specific transfer functions
Transfer functions

- Let $i$ be an instruction: $\text{IN}[i]$ and $\text{OUT}[i]$ are the set of data-flow values before and after the instruction $i$ of a program.
- A transfer function $fs$ relates the data-flow values before and after an instruction $i$.
- In a forward data-flow problem
  \[ \text{OUT}[i] = fs(\text{IN}[i]) \]
- In a backward data-flow problem
  \[ \text{IN}[i] = fs(\text{OUT}[i]) \]

$fs$ is DFA-specific.
Transfer function internals: $Y[i] = fs(X[i])$

• It relies on information that reaches i

• It transforms such information to propagate the result to the rest of the CFG

• To do so, it relies on information specific to i
  • Encoded in GEN[i], KILL[i]
    • $fs$ uses GEN[i] and KILL[i] to compute its output

• GEN[i] and KILL[i] are DFA-specific and data/control flow independent!
DFA steps

1) Set the DFA-specific sets GEN[i] and KILL[i], for all i

2) Implement the DFA-specific transfer function $fs$

3) Compute all $IN[i]$ and $OUT[i]$ following a DFA-generic algorithm

\[
\begin{align*}
\text{OUT}[i] &= \text{fs}(\text{IN}[i]) \\
\text{IN}[i] &= \text{fs}(\text{OUT}[i])
\end{align*}
\]

Compilers have a data flow framework to help developing new DFAs
Data Flow Analysis outline

• Why do we need DFA?

• Introduction to DFA

• A DFA example: reaching definitions

• Implementation of DFA
int sumcalc (int a, int b, int N){
    int x,y;
    \textcolor{red}{x = 0;}
    \textcolor{green}{y = 0;}
    if (a > b){
        \textcolor{red}{x = x + N;}
    }
    if (b > N){ return y;}
    return x;
}

Information needed just before an instruction $i$: what are the definitions that might execute before $i$?

\text{IN}[\text{return y}] = \{y=0\}
\text{IN}[\text{return x}] = \{x=0, \; x = x + N\}

if (b > N){ return 0;}

reach
Data-flow example: reaching definitions

- A definition $D$ reaches a program point $X$ if there is a control flow from $D$ to $X$ such that $D$ is not killed (invalidated) along that path.

- The reaching definition \textbf{data-flow problem} for a flow graph is to compute all definitions that reach an instruction $i$ (i.e., $\text{IN}[i]$, $\text{OUT}[i]$) for all $i$ in that graph.

\begin{align*}
\ldots \\
D : v &= 0 \\
\ldots \\
J : v &= v + n \\
\ldots \\
X : \ldots &= v \ldots
\end{align*}
Computing INs and OUTs

\[
\begin{align*}
\text{GEN}[0] &= \{ \} \\
\text{GEN}[1] &= \{ 1 \} \\
\text{GEN}[2] &= \{ 2 \} \\
\text{GEN}[3] &= \{ \}
\end{align*}
\]

\[
\begin{align*}
\text{IN} &= \{ \} \\
0: \text{int } x,y \\
1: x = 0 \\
2: y = 0 \\
3: \text{If } (a > b)
\end{align*}
\]

\[
\begin{align*}
\text{OUT} &= \{ x=0 \} \\
\text{GEN}[0] &= \{ \} \\
\text{GEN}[1] &= \{ 1 \} \\
\text{GEN}[2] &= \{ 2 \} \\
\text{GEN}[3] &= \{ \}
\end{align*}
\]

- Forward or backward?
- \( \text{OUT}[i] = \text{fs}(\text{IN}[i]) \)
- \( \text{GEN}[i] = \text{what } i \text{ generates} \)
- \( \text{KILL}[i] = \text{what } i \text{ invalidates} \)
- \( \text{fs} \text{ within a basic block?} \)
- Let \( i \) be an instruction and \( p \) be its only predecessor
  \[\text{IN}[i] = \text{OUT}[p]\]
  \[\text{OUT}[i] = \text{GEN}[i] \cup (\text{IN}[i] - \text{KILL}[i])\]
Data-flow example: reaching definitions

- A definition \( d \) reaches a program point \( X \) if there is a path from \( d \) to \( X \) such that \( d \) is not killed along that path.

- The data-flow problem for a flow graph is to compute \( \text{IN}[i] \) and \( \text{OUT}[i] \) for all \( i \) in that graph:

\[
\text{IN}[i] = \bigcup_{p \text{ a predecessor of } i} \text{OUT}[p] \\
\text{OUT}[i] = \text{GEN}[i] \cup (\text{IN}[i] - \text{KILL}[i]) \\
\text{IN}[\text{entry}] = \{\} \\
\]

Global reaching definitions

```
0: int x, y
1: x = 0
2: y = 0
3: if (a > b)
4: x = x + N
5: if (b > N)
```
Data Flow Analysis outline

• Why do we need DFA?

• Introduction to DFA

• A DFA example: reaching definitions

• Implementation of DFA
• So far, we have defined data-flow equations (i.e., IN and OUT equations)

• How can we actually compute them?
Iterative algorithm for reaching definitions

• Given GEN[i], KILL[i] for all instructions i, we compute IN[i] and OUT[i] for all i

\[
\text{OUT[ENTRY]} = \{ \};
\]

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

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

| GEN[0] = {} | KILL[0] = {} |
| GEN[1] = {1} | KILL[1] = {4} |

Why do we need to reach a fixed point?

| IN[0] = {} | OUT[0] = {} |
| IN[1] = {} | OUT[1] = {1} |
| IN[2] = {1} | OUT[2] = {1,2} |
| IN[3] = {1,2} | OUT[3] = {1,2} |
| IN[4] = {1,2} | OUT[4] = {2,4} |
| IN[5] = {1,2,4} | OUT[5] = {1,2,4} |

0: int x,y
1: x = 0
2: y = 0
3: if (a > b)
4: x = x + N
5: if (b > N)

\[ \text{IN}[i] = \bigcup_{p \text{ a predecessor}} \text{OUT}[p] \]
\[ \text{OUT}[i] = \text{GEN}[i] \cup (\text{IN}[i] - \text{KILL}[i]) \]
Implementation aspects

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

• Memory representation of data flow values
• Operations performed on them
• What is an element in a set?
Can we optimize the analysis?

\[
\text{OUT}[\text{ENTRY}] = \{ \};
\]

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

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

Optimization 1: basic blocks

OUT[ENTRY] = { };

for (each basic block B other than ENTRY) OUT[B] = { };

while (changes to any OUT occur)
  for (each basic block B other than ENTRY) {
    IN[B] = \bigcup p \text{ a predecessor of } B \text{ OUT}[p];
    OUT[B] = GEN[B] \cup (IN[B] \setminus KILL[B]);
  }

Contains all definitions killed by instructions in block B

Contains all definitions in block B that are visible immediately after B
Optimization 2: bit-sets

OUT[ENTRY] = { }; for (each basic block B other than ENTRY) OUT[B] = { }; while (changes to any OUT occur) 
  for (each basic block B other than ENTRY) 
    IN[B] = \bigcup \text{p a predecessor of } B \text{ OUT}[p];
    OUT[B] = \text{GEN}[B] \bigcup (\text{IN}[B] \setminus \text{KILL}[B]);
  }
Optimization 2: bit-sets

• Assign a bit to each element that might be in the set
  • Union: bit OR
  • Intersection: bit AND
  • Subtraction: bit NEGATE and AND

• Fast implementation
  • 64 elements packed to each word on today’s commodity processors
  • AND and OR are single machine code instructions (single cycle latency)
Optimization 3: work list

OUT[ENTRY] = { };
for (each basic block B other than ENTRY) OUT[B] = { };
while (changes to any OUT occur)

   for (each basic block B other than ENTRY) {
      IN[B] = \bigcup p \text{ a predecessor of } B \text{ OUT}[p];
      OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B]);
   }

}
Optimization 3: work list

OUT[ENTRY] = { };  
for (each basic block \( B \) other than ENTRY) \( \text{OUT}[B] = \{ \} \);  
workList = all basic blocks  
while (workList isn’t empty)

\( B = \text{pick and remove a block from workList} \)  
oldOUT = \( \text{OUT}[B] \)  
\( \text{IN}[B] = \bigcup \ p \quad \text{a predecessor of } B \quad \text{OUT}[p] \);  
\( \text{OUT}[B] = \text{GEN}[B] \bigcup (\text{IN}[B] – \text{KILL}[B]) \);  
if (oldOut != \( \text{OUT}[B] \)) workList = workList U \{ \text{all successors of } B \}  
}
Optimization 4: block order

\text{OUT}[\text{ENTRY}] = \{ \} ;

\text{for (each basic block } B \text{ other than ENTRY) } \text{OUT}[B] = \{ \} ;

\text{workList} = \text{all basic blocks}

\text{while (workList isn't empty)}

\text{B} = \text{pick and remove a block from workList}

\text{oldOUT} = \text{OUT}[B]

\text{IN}[B] = \bigcup p \text{ a predecessor of } B \ \text{OUT}[p] ;

\text{OUT}[B] = \text{GEN}[B] \cup (\text{IN}[B] - \text{KILL}[B] ) ;

\text{if (oldOut != OUT}[B])] \text{workList} = \text{workList U \{all successors of B\}}

\}


• What we learned was for forward data-flow analysis

\[ \text{OUT}[s] = fs( \text{IN}[s] ) \]

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

• What about backward data-flow analysis?

\[ \text{IN}[s] = fs( \text{OUT}[s] ) \]
Food for thought

• Correctness: is the answer ALWAYS correct?
• Meaning: what is exactly the meaning of the answer?
• Precision: how good is the answer?
• Convergence:
  • Will the analysis ALWAYS terminate?
  • Under what conditions does the iterative algorithm converge?
• Speed: how long does it take to converge?