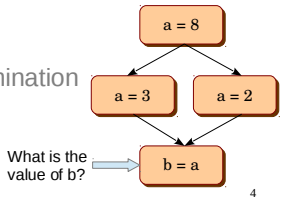
Data Flow Analysis

Rupesh Nasre.

CS6843 Program Analysis
IIT Madras
Jan 2014

Data Flow Analysis

- Flow-sensitive: Considers the control-flow in a function
- Operates on a flow-graph with nodes as basic-blocks and edges as the control-flow
- Examples
  - Constant propagation
  - Common subexpression elimination
  - Dead code elimination

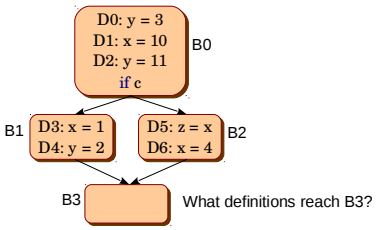


Outline

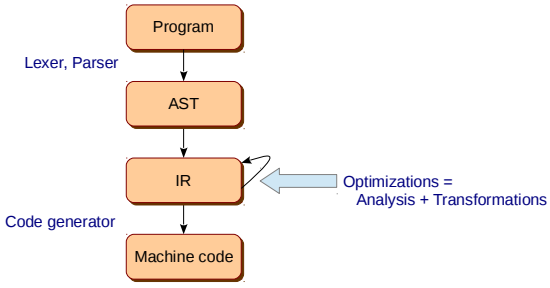
- What is DFA?
  - Reaching definitions
  - Live variables
- DFA framework
  - Monotonicity
  - Confluence operator
  - MFP/MOP solution
- Analysis dimensions

Reaching Definitions

- Every assignment is a definition
- A definition d reaches a program point p if there exists a path from the point immediately following d to p such that d is not killed along the path.



Compiler Organization

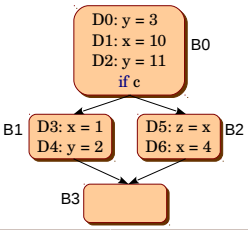


DFA Equations

- in(B) = set of data flow facts entering block B
- out(B) = …
- gen(B) = set of data flow facts generated in B
- kill(B) = set of data flow facts from the other blocks killed in B

DFA for Reaching Definitions

- in(B) = U out(P) where P is a predecessor of B
- out(B) = gen(B) U (in(B) – kill(B))
- Initially, out(B) = { }



gen(B0) = {D1, D2} kill(B0) = {D3, D4, D6}
gen(B1) = {D3, D4} kill(B1) = {D0, D1, D2, D6}
gen(B2) = {D5, D6} kill(B2) = {D1, D3}
gen(B3) = { } kill(B3) = { }

	in1	out1	in2	out2	in3	out3
B0	{}	{D1, D2}	{}	{D1, D2}	{}	{D1, D2}
B1	{}	{D3, D4}	{D1, D2}	{D3, D4}	{D1, D2}	{D3, D4}
B2	{}	{D5, D6}	{D1, D2}	{D2, D5, D6}	{D1, D2}	{D2, D5, D6}
B3	{}	{}	{D3, D4, D5, D6}	{D3, D4, D5, D6}	{D2, D3, D4, D5, D6}	{D2, D3, D4, D5, D6}

Direction and Confluence

	Forward	Backward
U	Reaching Definitions	Live Variables
∩	Common Subexpressions	Very Busy Expressions

DFA for Reaching Definitions

Domain	Sets of definitions
Transfer function	in(B) = U out(P) out(B) = gen(B) U (in(B) - kill(B))
Direction	Forward
Meet / confluence operator	U
Initialization	out(B) = { }

Data Flow Framework

- Point: start or end of a basic block
- Information flow direction: forward / backward
- Transfer functions
- Meet / confluence operator
- One can define a transfer function over a path in the CFG $f_k(f_{k-1}(...f_2(f_1(f_0(T))...))$
- $MOP(x) = \Pi f_q(T)$

Meet over all paths
Path enumeration is expensive

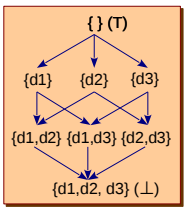
DFA for Live Variables

Domain	Sets of variables
Transfer function	in(B) = use(B) U (out(B) - def(B)) out(B) = U in(S) where S is a successor of B
Direction	Backward
Meet / confluence operator	U
Initialization	in(B) = { }

A variable v is live at a program point p if v is used along some path in the flow graph starting at p.
Otherwise, the variable v is dead.

Structure in Data Flow Framework

- A semilattice $\mathcal{L}$ with a binary meet operator Π, such that a, b, c ∈ $\mathcal{L}$
  - Idempotency: a Π a = a
  - Commutativity: a Π b = b Π a
  - Associativity: a Π (b Π c) = (a Π b) Π c
- Π imposes an order on $\mathcal{L}$
  - a >= b ⇔ a Π b = b
- $\mathcal{L}$ has a bottom element ⊥, a Π ⊥ = ⊥
- $\mathcal{L}$ has a top element T, a Π T = T



Reaching Definitions Lattice

Monotone Framework

- Let $\mathcal{F}: \mathcal{L} \rightarrow \mathcal{L}$ be a function that transforms a semilattice into another
- A framework $<\mathcal{L}, \sqcap, \mathcal{F}>$ is monotone if $\mathcal{F}$ is monotonic, i.e.,

  $(\forall f \in F)(\forall x, y \in \mathcal{L}), x \geq y \Rightarrow f(x) \geq f(y)$
- If a data-flow framework is monotonic, the convergence (termination) is guaranteed for finite height lattices.

Distributive Framework

- A framework $<\mathcal{L}, \sqcap, \mathcal{F}>$ is distributive if $\mathcal{F}$ is distributive, i.e.,

  $(\forall f \in F)(\forall x, y \in \mathcal{L}) f(x \sqcap y) \leq f(x) \sqcap f(y)$
- Maximal fixed point (MFP) solution is obtained with our iterative DFA.
- MFP is unique and order independent.
- The best we can do is MOP (most feasible, but undecidable).
- In general, MFP ≤ MOP ≤ Perfect solution.
- If distributive, MFP = MOP.
- Every distributive function is also monotonic.

Outline

- What is DFA?
  - Reaching definitions
  - Live variables
- DFA framework
  - Monotonicity
  - Confluence operator
  - MFP/MOP solution
- Analysis dimensions

Analysis Dimensions

An analysis's precision and efficiency is guided by various design decisions.

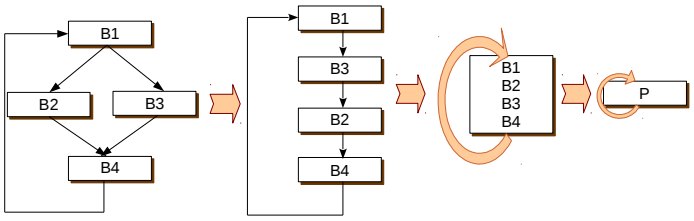
- Flow-sensitivity
- Context-sensitivity
- Path-sensitivity
- Field-sensitivity

Flow-sensitivity

```
L0:  a = 0;
L1:  a = 1;
L2:  ...
```

Flow-sensitive solution: *at L1 a is 0, at L2 a is 1*
Flow-insensitive solution: *in the program a is in {0, 1}*

Flow-insensitive analyses ignore the control-flow in the program.



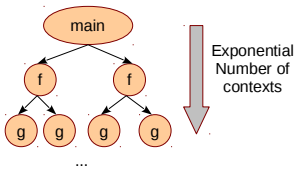
Context-sensitivity

```
main() {
    L0:  fun(0);
    L1:  fun(1);
}
```

```
fun(int x) {
    y = x;
}
```

Context-sensitive solution:
y is 0 along L0, y is 1 along L1

Context-insensitive solution:
y is in {0, 1} in the program



Exponential Number of contexts

Along main-f1-g1, ...
Along main-f1-g2, ...
Along main-f2-g1, ...
Along main-f2-g2, ...

Exponential time requirement

Exponential storage requirement

Context-sensitivity

```
main() {
    L0:  fun(0);
    L1:  fun(1);
}
```

```
fun(int x) {
    y = x;
}
```

Context-sensitive solution:
y is 0 along L0, y is 1 along L1

Context-insensitive solution:
Inter-procedural → *y is in {0, 1} in the program*
intra-procedural → *y is in {-∞, +∞} in the program*

Path-sensitivity

```
if (a == 0)
    b = 1;
else
    b = 2;
```

Path-sensitive solution:
b is 1 when a is 0, b is 2 when a is not 0

Path-insensitive solution:
b is in {1, 2} in the program

```
if (c1)
  while (c2) {
    if (c3)
      …
    else
      for (; c4; )
        ...
  }
else
  ...
```

c1 and c2 and c3, …
c1 and c2 and !c3 and c4, …
c1 and c2 and !c3 and !c4, …
c1 and !c2, …
!c1 ...
...

Field-sensitivity

```
struct T s;

s.a = 0;
s.b = 1;
```

Field-sensitive solution:
s.a is 0, s.b is 1

Field-insensitive solution:
s is in {0, 1}

Aggregates are collapsed into a single variable.
e.g., arrays, structures, unions.

This reduces the number of variables tracked during the analysis and reduces precision.

A Note on Abstraction

Maintain one bit for x == 0
Initialized to F (false)

```
?
x = 0;
T
++x;
F
--x;
?
```


A Note on Choosing Abstraction

Maintain one bit for x == 0
Initialized to F (false)

```
?
x = 0;
T
++x;
F
--x;
?
```

Maintain two bits for value of x
Initialized to 00

```
??
x = 0;
00
++x;
01
--x;
00
```

Maintain one bit for x == 0
Another bit for x < 2
Initialized to 00

```
??
x = 0;
11
++x;
01
--x;
11
```


Acknowledgements

Course notes from

- Katheryn McKinley
- Monica Lam
- Y. N. Srikant
- Uday Khedker