CS738: Advanced Compiler Optimizations

Data Flow Analysis

Amey Karkare
karkare@cse.iitk.ac.in

http://www.cse.iitk.ac.in/~karkare/cs738
Department of CSE, IIT Kanpur

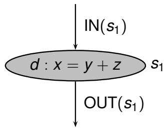
Agenda

- *Intraprocedural* Data Flow Analysis: Classical Examples
  - Last lecture: Reaching Definitions
  - Today: Available Expressions
  - Discussion about the similarities/differences

Available Expressions Analysis

- An expression e is available at a point p if
  - **Every** path from the *Entry* to p has at least one evaluation of e
  - There is no assignment to any component variable of e **after the last evaluation** of e prior to p
- Expression e is *generated* by its evaluation
- Expression e is *killed* by assignment to its component variables

AvE Analysis of a Structured Program

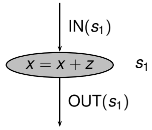


$$\begin{aligned} \text{OUT}(s_1) &= \text{IN}(s_1) - \text{KILL}(s_1) \cup \text{GEN}(s_1) \\ \text{GEN}(s_1) &= \{y+z\} \\ \text{KILL}(s_1) &= E_x \end{aligned}$$

where E_x: set of all expression having x as a component

This may not work in general – WHY?

AvE Analysis of a Structured Program

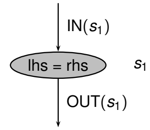


$$\begin{aligned} \text{OUT}(s_1) &= \text{IN}(s_1) - \text{KILL}(s_1) \cup \text{GEN}(s_1) \\ \text{GEN}(s_1) &= \{x+z\} \\ \text{KILL}(s_1) &= E_x \end{aligned}$$

Incorrectly marks $x + z$ as available after s_1

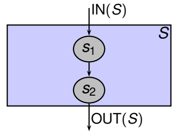
$$\text{GEN}(s_1) = \emptyset \text{ for this case}$$

AvE Analysis of a Structured Program



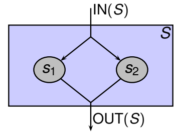
$$\begin{aligned} \text{OUT}(s_1) &= \text{IN}(s_1) - \text{KILL}(s_1) \cup \text{GEN}(s_1) \\ \text{GEN}(s_1) &= \{\text{rhs} \mid \text{lhs is not part of rhs}\} \\ \text{KILL}(s_1) &= E_{\text{lhs}} \end{aligned}$$

AvE Analysis of a Structured Program



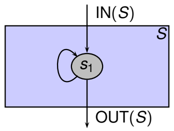
$$\begin{aligned} \text{GEN}(S) &= \text{GEN}(s_1) - \text{KILL}(s_2) \cup \text{GEN}(s_2) \\ \text{KILL}(S) &= \text{KILL}(s_1) - \text{GEN}(s_2) \cup \text{KILL}(s_2) \\ \text{IN}(s_1) &= \text{IN}(S) \\ \text{IN}(s_2) &= \text{OUT}(s_1) \\ \text{OUT}(S) &= \text{OUT}(s_2) \end{aligned}$$

AvE Analysis of a Structured Program



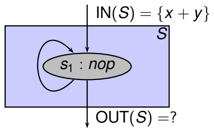
$$\begin{aligned} \text{GEN}(S) &= \text{GEN}(s_1) \cap \text{GEN}(s_2) \\ \text{KILL}(S) &= \text{KILL}(s_1) \cup \text{KILL}(s_2) \\ \text{IN}(s_1) &= \text{IN}(s_2) = \text{IN}(S) \\ \text{OUT}(S) &= \text{OUT}(s_1) \cap \text{OUT}(s_2) \end{aligned}$$

AvE Analysis of a Structured Program



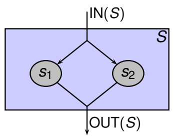
$$\begin{aligned} \text{GEN}(S) &= \text{GEN}(s_1) \\ \text{KILL}(S) &= \text{KILL}(s_1) \\ \text{OUT}(S) &= \text{OUT}(s_1) \\ \text{IN}(s_1) &= \text{IN}(S) \cap \text{GEN}(s_1)\ ? \\ \text{IN}(s_1) &= \text{IN}(S) \cap \text{OUT}(s_1)?? \end{aligned}$$

AvE Analysis of a Structured Program



Is x + y available at OUT(S)?

AvE Analysis is Approximate

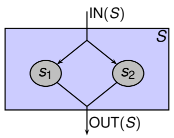


- Assumption: All paths are feasible.
- Example:

```
if (true) s1;
else      s2;
```

		Fact	**Computed**		**Actual**
GEN(S)	=		GEN(s_1) ∩ GEN(s_2)	⊆	GEN(s_1)
KILL(S)	=		KILL(s_1) ∪ KILL(s_2)	⊇	KILL(s_1)

AvE Analysis is Approximate



- Thus,
  true GEN(S) ⊇ analysis GEN(S)
  true KILL(S) ⊆ analysis KILL(S)
- Fewer expressions marked available than actually do!
- Later we shall see that this is SAFE approximation
  - prevents optimizations
  - but NO wrong optimization

AvE for Basic Blocks

- Expr e is available at the start of a block if
  - It is available at the end of all predecessors

$$\text{IN}(B) = \bigcap_{P \in \text{PRED}(B)} \text{OUT}(P)$$

- Expr e is available at the end of a block if
  - Either it is generated by the block
  - Or it is available at the start of the block and not killed by the block

$$\text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B)$$

Solving AvE Constraints

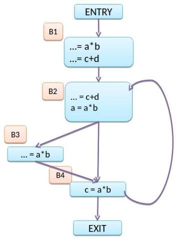
- KILL & GEN known for each BB.
- A program with N BBs has $2N$ equations with $2N$ unknowns.
  - Solution is possible.
  - Iterative approach (on the next slide).

```
for each block B {
    OUT(B) = 𝒰; 𝒰 = "universal" set of all exprs
}
OUT(Entry) = ∅; // remember reaching defs?
change = true;
while (change) {
    change = false;
    for each block B other than Entry {
        IN(B) = ⋂_{P∈PRED(B)} OUT(P);
        oldOut = OUT(B);
        OUT(B) = IN(B) − KILL(B) ∪ GEN(B);
        if (OUT(B) ≠ oldOut) then {
            change = true;
        }
    }
}
```

Some Issues

- What is $\mathcal{U}$ – the set of *all* expressions?
- How to compute it efficiently?
- Why *Entry* block is initialized differently?

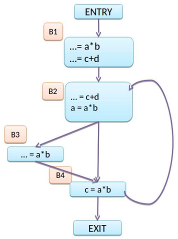
Available Expressions: Example



BB	GEN	KILL
B1	{a*b, c+d}	{ }
B2	{c+d}	{a*b}
B3	{a*b}	{}
B4	{a*b}	{c+d}

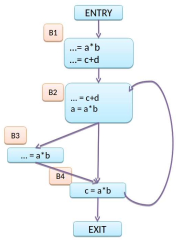
$\mathcal{U}$= {a*b, c+d}

Available Expressions: Example



Pass#	Pt	B1	B2	B3	B4
Init	IN	-	-	-	-
	OUT	$\mathcal{U}$	$\mathcal{U}$	$\mathcal{U}$	$\mathcal{U}$
1	IN	$\emptyset$	a*b, c+d	c+d	c+d
	OUT	a*b, c+d	c+d	a*b, c+d	a*b
2	IN	$\emptyset$	a*b	c+d	c+d
	OUT	a*b, c+d	c+d	a*b, c+d	a*b
3	IN	$\emptyset$	a*b	c+d	c+d
	OUT	a*b, c+d	c+d	a*b, c+d	a*b

Available Expressions: Bitvectors



a bit for each expression:

a*b	c+d

Pass#	Pt	B1	B2	B3	B4
Init	IN	-	-	-	-
	OUT	11	11	11	11
1	IN	00	11	01	01
	OUT	11	01	11	10
2	IN	00	10	01	01
	OUT	11	01	11	10
3	IN	00	10	01	01
	OUT	11	01	11	10

Available Expressions: Bitvectors

- Set-theoretic definitions:

$$\text{IN}(B) = \bigcap_{P \in \text{PRED}(B)} \text{OUT}(P)$$

$$\text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B)$$

- Bitvector definitions:

$$\text{IN}(B) = \bigwedge_{P \in \text{PRED}(B)} \text{OUT}(P)$$

$$\text{OUT}(B) = \text{IN}(B) \wedge \neg\text{KILL}(B) \vee \text{GEN}(B)$$

- Bitwise $\vee, \wedge, \neg$ operators

Available Expressions: Application

- Common subexpression elimination in a block *B*
  - Expression *e* available at the entry of *B*
  - *e* is also computed at a point *p* in *B*
  - Components of *e* are not modified from entry of *B* to *p*
- *e* is "upward exposed" in *B*
- Expressions generated in *B* are "downward exposed"

Comparison of RD and AvE

- *Some* vs. *All* path property
- Meet operator: $\cup$ vs. $\cap$
- Initialization of *Entry*: $\emptyset$
- Initialization of other BBs: $\emptyset$ vs. $\mathcal{U}$
- Safety: "More" RD vs. "Fewer" AvE

AvE: alternate Initialization

- What if we Initialize:

$$\text{OUT}(B) = \emptyset, \forall B \text{ including } Entry$$

- Would we find "extra" available expressions?
  - More opportunity to optimize?
- OR would we miss some expressions that are available?
  - Loose on opportunity to optimize?

Live Variables

- A variable *x* is live at a point *p* if
  - There is a point *p′* along some path in the flow graph starting at *p* to the *Exit*
  - Value of *x* could be used at *p′*
  - There is no definition of *x* between *p* and *p′* along this path
- Otherwise *x* is dead at *p*

Live Variables: GEN

- GEN(*B*): Set of variables whose values may be used in block *B* prior to any definition
  - Also called "use(*B*)"
- "upward exposed use" of a variable in *B*

Live Variables: KILL

- KILL(*B*): Set of variables defined in block *B* prior to any use
  - Also called "def(*B*)"
- "upward exposed definition" of a variable in *B*

Live Variables: Equations

- Set-theoretic definitions:

$$\text{OUT}(B) = \bigcup_{S \in \text{SUCC}(B)} \text{IN}(S)$$

$$\text{IN}(B) = \text{OUT}(B) - \text{KILL}(B) \cup \text{GEN}(B)$$

- Bitvector definitions:

$$\text{OUT}(B) = \bigvee_{S \in \text{SUCC}(B)} \text{OUT}(S)$$

$$\text{IN}(B) = \text{OUT}(B) \wedge \neg\text{KILL}(B) \vee \text{GEN}(B)$$

- Bitwise $\vee, \wedge, \neg$ operators

Very Busy Expressions

- Expression *e* is very busy at a point *p* if
  - *Every* path from *p* to *Exit* has at least one evaluation of *e*
  - On every path, there is no assignment to any component variable of *e* before the first evaluation of *e* following *p*
- Also called *Anticipable expression*

QQ

- Expression *e* is very busy at a point *p* if
  - **Every** path from *p* to *Exit* has at least one evaluation of *e* and there is no assignment to any component variable of *e* before the first evaluation of *e* following *p* on these paths.
- Set up the data flow equations for Very Busy Expressions (VBE). You have to give equations for GEN, KILL, IN, and OUT.
- Think of an optimization/transformation that uses VBE analysis. Briefly describe it (2-3 lines only)
- Will your optimization be *safe* if we replace "*Every*" by "*Some*" in the definition of VBE?