CS406: Compilers

Spring 2022

Week 13:

More Dataflow Analysis – Uninitialized Variables, Available Expressions, Reaching Definitions

Register Allocation

Uninitialized Variables

- **Goal:** determine a set of variables that are possibly uninitialized at the beginning and end of a basic block.
  - E.g. to know if `x==null`?
- **Direction of the analysis:**
  - How does information flow w.r.t. control flow?
- **Join operator:**
  - What happens at merge points? E.g. what operator to use Union or Intersection?
- **Transfer function:**
  - Define sets UninitIn(b), UninitOut(b), Init(b), Uninit(b)
- **Initializations?**

Worksheet

Available Expressions

- **Goal:** determine a set of expressions that have already been computed.
  - E.g. to perform global CSE
- **Direction of the analysis:**
  - How does information flow w.r.t. control flow?
- **Join operator:**
  - What happens at merge points? E.g. what operator to use Union or Intersection?
- **Transfer function:**
  - Define sets AvailIn(b), AvailOut(b), Compute(b), Kill(b)
- **Initializations?**

Transfer functions for meet

- What do the transfer functions look like if we are doing a meet?

$$
\begin{aligned}
IN(S) &= \bigcap_{t \in pred(s)} OUT(t) \\
OUT(S) &= \mathbf{gen}(s) \cup (IN(S) - \mathbf{kill}(s))
\end{aligned}
$$

- gen(s): expressions that *must be* computed in this statement
- kill(s): expressions that use variables that *may* be defined in this statement
  - Note difference between these sets and the sets for reaching definitions or liveness
- Insight: gen and kill must never lead to incorrect results
  - Must not decide an expression is available when it isn't, but OK to be safe and say it isn't
  - Must not decide a definition *doesn't* reach, but OK to overestimate and say it does

Analysis initialization

- How do we initialize the sets?
  - If we start with everything initialized to $\perp$, we compute the smallest sets
  - If we start with everything initialized to $\top$, we compute the largest
- Which do we want? It depends!
  - Reaching definitions: a definition that *may* reach this point
    - We want to have as few reaching definitions as possible $\rightarrow$ $\perp$
  - Available expressions: an expression that *was definitely* computed earlier
    - We want to have as many available expressions as possible $\rightarrow$ $\top$
  - Rule of thumb: if confluence operator is $\sqcup$, start with $\perp$, otherwise start with $\top$

```
void             (int m, int n)

{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
                (m,j);              (i+1,n);
}
```

What is this piece of code doing?

[1] R. Sedgewick, "Implementing Quicksort Programs," *Comm. ACM*, **21**, 1978, pp. 847–857.

Intermediate code (assuming int is 4 bytes):

(Ignore the temporary counter value for now)

```
void quicksort(int m, int n)

{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
```

Intermediate code (after CSE):

```
void quicksort(int m, int n)

{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
```

```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
```

Intermediate code (assuming int is 4 bytes):

(assume next temporary counter value=11)

```
void quicksort(int m, int n)

{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

```
t11 = 4*i
```

Dataflow Analysis – Problem Categorization

- All path problem:
  - we want the property to hold at all the paths reaching a program point.
- Any path problem:
  - we want the property to hold at some path reaching a program point.

Orthogonal to the above categorization we can have:

- Forward flow problem:
  - Transfer of information done along the direction of the control flow
- Backward flow problem:
  - Transfer of information done opposite to the direction of the control flow

Reaching definitions

- What definitions of a variable *reach* a particular program point
  - A definition of variable x from statement s reaches a statement t if there is a path from s to t where x is not redefined
- Especially important if x is used in t
  - Used to build *def-use* chains and *use-def* chains, which are key building blocks of other analyses
    - Used to determine dependences: if x is defined in s and that definition reaches t then there is a flow dependence from s to t
  - We used this to determine if statements were loop invaraint
    - All definitions that reach an expression must originate from outside the loop, or themselves be invariant

Creating a reaching-def analysis

- Can we use a powerset lattice?
- At each program point, we want to know which definitions have reached a particular point
  - Can use powerset of set of definitions in the program
    - V is set of variables, S is set of program statements
    - Definition: $d \in V \times S$
      - Use a tuple, $\langle v, s \rangle$
  - How big is this set?
    - At most $|V \times S|$ definitions

Forward or backward?

- What do you think?

Choose confluence operator

- Remember: we want to know if a definition *may* reach a program point
- What happens if we are at a merge point and a definition reaches from one branch but not the other?
  - We don't know which branch is taken!
  - We should union the two sets – any of those definitions can reach
- We want to avoid getting too many reaching definitions → should start sets at ⊥

Transfer functions for RD

- Forward analysis, so need a slightly different formulation
  - Merged data flowing into a statement

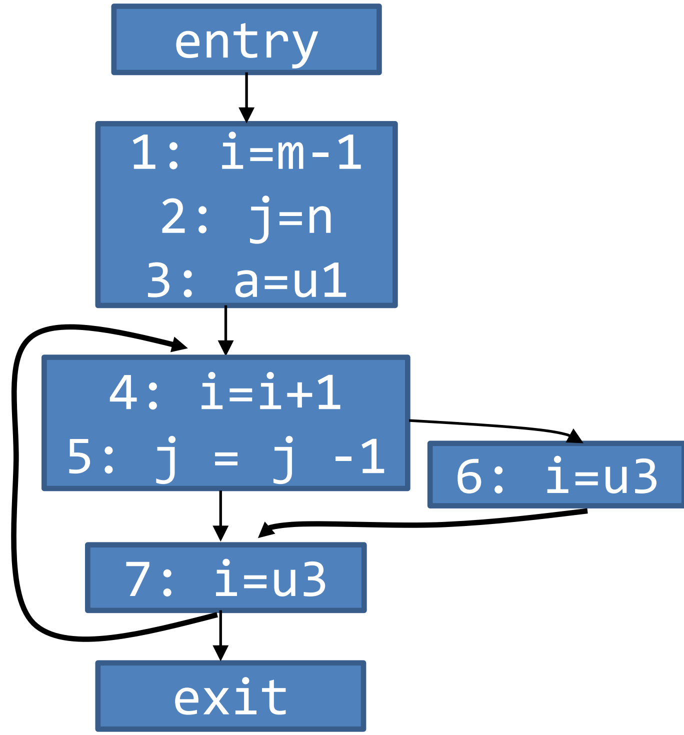
$$\begin{aligned} IN(s) &= \bigcup_{t \in pred(s)} OUT(t) \\ OUT(s) &= \mathbf{gen}(s) \cup (IN(s) - \mathbf{kill}(s)) \end{aligned}$$

- What are gen and kill?
  - gen(s): the set of definitions that *may* occur at s
    - e.g., gen(s_1: x = e) is <x, s_1>
  - kill(s): all previous definitions of variables that are *definitely* redefined by s
    - e.g., kill(s_1: x = e) is <x, *>

Reaching Definitions

- **Goal:** to know where in a program each variable x may have been defined when control reaches block b
- Definition d reaches block b if there is a path from point immediately following d to b, such that the variable defined in d is not redefined / killed along that path

$$In(b) = \bigcup_{i \in Pred(b)} Out(i)$$



$$Out(b) = gen(b) \cup (In(b) - kill(b))$$

//set that contains all statements that may define some variable x in b

gen(1:a=3;2:a=4)={2}

//set that contains all statements that define a variable x that is also defined in b

kill(1:a=3; 2:a=4)={1,2}