Live and Dead Code
what is dead code?

• Some instructions don’t do anything (especially after other optimization has been done) and are **dead code**

```
1: A = B * C
2: A = C + X
```

First computation of A produces a value that won’t be used

• Difficulty: not always obvious that an instruction is dead: property is transitive

```
1: A = B * C
2: B = C * A
3: D = A + B
4: E = D + A
5: E = 7
```

Instructions 1 through 4 are all dead, but it’s hard to see that
turn it around: what is live?

• Easier to focus on the dual problem: what code is **live**
  • A variable is **live** if it has a value that *may* be used in the future
  • At any point in code, multiple variables can be live

• Question: how do you know what is going to happen in the future?
• Answer: go backwards!
executing backwards

- A variable is live if its value may be used in the future
- At the end of a basic block, we can make a good guess about what is live
  - Temporaries are not live (they only get used during the execution of single statements, so are not used in the future)
  - Local variables and global variables may be used elsewhere, so they are live
- Can then propagate this information backwards
next: liveness analysis
propagating liveness

• Suppose we have a set of variables that are live at a particular point in the program
• What does it mean to “execute” a statement backwards?

{???, ????}

A = B * C

{A, D}
propagating liveness

• Suppose we have a set of variables that are live at a particular point in the program

• What does it mean to “execute” a statement backwards?

\[
\begin{align*}
\{???, \quad ??? \} \\
A &= B \times C \\
\{A, D\} \\
L_{in} &= (L_{out} - K) \cup G
\end{align*}
\]
propagating liveness

• Suppose we have a set of variables that are live at a particular point in the program

• What does it mean to “execute” a statement backwards?

\[
\begin{align*}
\{???, \{A, D\}\} \\
A &= B \times C \\
L_{in} &= (L_{out} - K) \cup G
\end{align*}
\]

variables “generated,” or used, by a statement

variables “killed,” or defined, by a statement
liveness example

• What is live in this code?

```plaintext
1:   A = B + C
2:   C = A + B
3:   T1 = B + C
4:   T2 = T1 + C
5:   D = T2
6:   E = A + B
7:   B = E + D
8:   A = C + D
9:   T3 = A + B
10:  WRITE(T3)
```
liveness example

- What is live in this code?

1: \[ A = B + C \]
2: \[ C = A + B \]
3: \[ T1 = B + C \]
4: \[ T2 = T1 + C \]
5: \[ D = T2 \]
6: \[ E = A + B \]
7: \[ B = E + D \]
8: \[ A = C + D \]
9: \[ T3 = A + B \]
10: WRITE(T3)

1: \{A, B\}
2: \{A, B, C\}
3: \{A, B, C, T1\}
4: \{A, B, C, T2\}
5: \{A, B, C, D\}
6: \{C, D, E\}
7: \{B, C, D\}
8: \{A, B\}
9: \{T3\}
10: \{}
what about aliasing?

• Aliasing, as usual is a problem
• Remember: compilers must be conservative
• Liveness is a *may* property → OK to say something is live when it isn’t
  • This *may* be used in the future (even if it really won’t be)

• Deal with aliasing by being conservative:
  • A variable stops being live when it is written to
  • Only *kill* variables that are *definitely* written to
Finding Dead Code
how do we find dead code?

• Easy answer: if the variable being written by an instruction is not live, the code is dead

• Intuition: the value you are generating is not being used anywhere else, so generating this value is pointless

```
1:  A = B + C
2:  C = A + B
3:  T1 = A + B
4:  D = T1 + C
5:  T2 = D + T1
6:  D = A + B
7:  WRITE(D)
```
how do we find dead code?

• Easy answer: if the variable being written by an instruction is not live, the code is dead

• Intuition: the value you are generating is not being used anywhere else, so generating this value is pointless

1: \( A = B + C \)  
2: \( C = A + B \)  
3: \( T1 = A + B \)  
4: \( D = T1 + C \)  
5: \( T2 = D + T1 \)  
6: \( D = A + B \)  
7: WRITE(D)

1: \{A, B\}  
2: \{A, B, C\}  
3: \{A, B, C, T1\}  
4: \{A, B, D, T1\}  
5: \{A, B\}  
6: \{D\}  
7: \{\}
how do we find dead code?

• Easy answer: if the variable being written by an instruction is not live, the code is dead

• Intuition: the value you are generating is not being used anywhere else, so generating this value is pointless

1: \(A = B + C\)
2: \(C = A + B\)
3: \(T1 = A + B\)
4: \(D = T1 + C\)
5: \(T2 = D + T1\)
6: \(D = A + B\)
7: `WRITE(D)`

1: \{A, B\}
2: \{A, B, C\}
3: \{A, B, C, T1\}
4: \{A, B, D, T1\}
5: \{A, B\}
6: \{D\}
7: \{\}
how do we find dead code?

• Easy answer: if the variable *being written* by an instruction is not live, the code is dead

• Intuition: the value you are generating is not being used anywhere else, so generating this value is pointless

1:  A = B + C
2:  C = A + B
3:  T1 = A + B
4:  D = T1 + C
5:  T2 = D + T1
6:  D = A + B
7:  WRITE(D)

1:  \{A, B\}
2:  \{A, B, C\}
3:  \{A, B, C, T1\}
4:  \{A, B, D, T1\}
5:  \{A, B\}
6:  \{D\}
7:  \{\}
how do we find dead code?

• After you remove dead code, it changes liveness information
• Recompute and iterate

1: \( A = B + C \)  
2: \( C = A + B \)  
3: \( T1 = A + B \)  
4: \( D = T1 + C \)  
5: \( T2 = D + T1 \)  
6: \( D = A + B \)  
7: WRITE(D)

1: \{A, B\}  
2: \{A, B, C\}  
3: \{A, B, C, T1\}  
4: \{A, B, D, T1\}  
5: \{A, B\}  
6: \{D\}  
7: \{\}
how do we find dead code?

- After you remove dead code, it changes liveness information
- Recompute and iterate

1:   A = B + C
2:   C = A + B
3:   T1 = A + B
4:   D = T1 + C
6:   D = A + B
7:   WRITE(D)

1:   {A, B}
2:   {A, B, C}
3:   {A, B, C, T1}
4:   {A, B}
6:   {D}
7:   { }
how do we find dead code?

• After you remove dead code, it changes liveness information
• Recompute and iterate

1: \(A = B + C\)  
2: \(C = A + B\)  
3: \(T_1 = A + B\)  
4: \(D = T_1 + C\)  
5: \(D = A + B\)  
6: \(\text{WRITE}(D)\)  

1: \(\{A, B\}\)  
2: \(\{A, B, C\}\)  
3: \(\{A, B, C, T_1\}\)  
4: \(\{A, B\}\)  
5: \(\{D\}\)  
6: \(\{\}\\)
how do we find dead code?

- After you remove dead code, it changes liveness information
- Recompute and iterate

1: \( A = B + C \)  
2: \( C = A + B \)  
3: \( T1 = A + B \)  

1: \( \{A, B\} \)  
2: \( \{A, B, C\} \)  
3: \( \{A, B, C, T1\} \)  

6: \( D = A + B \)  
7: \( \text{WRITE}(D) \)  
6: \( \{D\} \)  
7: \( \{} \)
how do we find dead code?

• After you remove dead code, it *changes liveness information*

• Recompute and iterate

1: \( A = B + C \)  \hspace{1cm} 1: \{A, B\}

2: \( C = A + B \)  \hspace{1cm} 2: \{A, B\}

3: \( T1 = A + B \)  \hspace{1cm} 3: \{A, B\}

6: \( D = A + B \)  \hspace{1cm} 6: \{D\}

7: \text{WRITE}(D) \hspace{1cm} 7: \{\}
how do we find dead code?

• After you remove dead code, it changes liveness information
• Recompute and iterate

1: \( A = B + C \)
2: \( C = A + B \)
3: \( T1 = A + B \)
6: \( D = A + B \)
7: \text{WRITE}(D)

1: \{A, B\}
2: \{A, B\}
3: \{A, B\}
6: \{D\}
7: \{\}
how do we find dead code?

- After you remove dead code, it changes liveness information
- Recompute and iterate

```
1: A = B + C
6: D = A + B
7: WRITE(D)
```
```
1: {A, B}
6: {D}
7: {}
```
can we do this faster?

- Recomputing and iterating is slow!
- We can speed this up by computing **use-def** chains
  - Track how uses of variables are connected to definitions of those variables
- Can trace backwards from live code along use def chains
  - Instruction is “backwards reachable” from live code $\rightarrow$ instruction is live
  - Instruction is *not* backwards reachable $\rightarrow$ no definition from this instruction eventually propagates to live code, instruction is dead
- This generalizes to a program analysis technique called **program slicing**