CS 473: COMPILER DESIGN
Compilation in a Nutshell

Abstract Syntax Tree:

If
  Eq
  Assn
  b
  0
  a
  1

Semantic Analysis

Intermediate Representation

Code Analysis

Optimization

Backend

Intermediate code:

```
11:  %cnd = icmp eq i64 %b, 0
    br i1 %cnd, label %12, label %13
12:    store i64* %a, 1
    br label %13
13:
```

Assembly Code

```
11:
    cmpq %eax, $0
    jeq 12
    jmp 13
12:
    ...
```
Motivating Code Analyses

- There are lots of things that might influence the safety/applicability of an optimization
  - What algorithms and data structures can help?

- How do you know what is a loop?
- How do you know an expression is invariant?
- How do you know if an expression has no side effects?
- How do you keep track of where a variable is defined?
- How do you know where a variable is used?
- How do you know if two reference values may be aliases of one another?
Program 4 Questions

Top
Program 5: Instruction Selection

• (demo)
Questions

Top
Moving Towards Register Allocation

• The Tiger compiler so far generates as many temporary registers as it needs
• We can store as many as we want on the stack, but ideally we’d like to make as many as possible into registers
• Registers are a very limited resource! For most targets, between 16 and 32
• But we can *reuse* registers for multiple temporaries!

\[
\begin{align*}
  r_1 &= fp + 1 \\
  r_2 &= \left[ r_1 \right] \\
  r_3 &= r_2 + i \\
  r_4 &= fp + 2 \\
  r_5 &= \left[ r_4 \right] \\
  \left[ r_3 \right] &= r_5
\end{align*}
\]

• The program is always done with \( r_1 \) before it starts using \( r_4 \), so we can store both \( r_1 \) and \( r_4 \) in the same machine register

---

\*done with \( r_1 \)\n\*done with \( r_4 \)
Liveness

\[ r1 = fp + 1 \]
\[ r2 = [r1] \]
\[ r3 = r2 + i \]
\[ r4 = fp + 2 \]
\[ r5 = [r4] \]
\[ [r3] = r5 \]

- The program is always done with \( r1 \) before it starts using \( r4 \), so we can store both \( r1 \) and \( r4 \) in the same machine register.

- Two temps can be assigned to the same register if their values will not be needed at the same time.
  - A temp is *needed* if its contents will be used as a source operand in a later instruction.
- Such a variable is called “live”.
Liveness

- The program is always done with r1 before it starts using r4, so we can store both r1 and r4 in the same machine register.

```
  r1 = fp + 1  [r1]  r1 live
  r2 = [r1]     
  r3 = r2 + i
  r4 = fp + 2  [r4]  r4 live
  r5 = [r4]
  [r3] = r5
```

- Two temps can be assigned to the same register if their values will not be needed at the same time.
  - A temp is *needed* if its contents will be used as a source operand in a later instruction.
- Such a temp is called “*live*”.
- Two temps can share the same register if they are not live at the same time.

- Two temps can be assigned to the same register if their values will not be needed at the same time.

  - A temp is *needed* if its contents will be used as a source operand in a later instruction.

  - Such a temp is called “*live*”.

  - Two temps can share the same register if they are not live at the same time.
Questions
**Liveness Analysis**

- A temp starts being live when it’s defined, and stays live until it’s no longer used
  - So we’ll need to take into account all the *definitions* and *uses* of temps in the program

- Computing the live temps at each point in the program is called *liveness analysis*
  - Will probably have to be a conservative approximation, since we can never store two temps in one register at the same time!

- Liveness analysis is one example of *dataflow analysis*
  - Other examples: Available Expressions, Reaching Definitions, Constant-Propagation Analysis, ...
For the purposes of dataflow analysis, we use the *control flow graph* (CFG) intermediate form.

A *control flow graph* is a graph in which each node is a basic block:
- There is an edge from B1 to B2 if the control-flow instruction of B1 might jump to the entry label of B2
- There are no “dangling” edges – there is a block for every jump target

We can build CFGs at various levels: source, target, IRs
- Same general idea, but the exact details will differ
Dataflow over CFGs

- Sometimes it is helpful to think of the fall-through between sequential instructions as an edge of the control-flow graph too.
Liveness is Associated with *Edges*

\[
\begin{align*}
    r_1 &= fp + 1 \\
    r_2 &= [r_1] \\
    r_3 &= r_2 + i \\
    r_4 &= fp + 2 \\
    r_5 &= [r_4] \\
    [r_3] &= r_5
\end{align*}
\]

Instr

Live: a, b

Live: b, d, e

done with r1

done with r4
Liveness is Associated with *Edges*

- This is useful so that the same register can be used for different temporaries in the same statement
- Example: \( a = b + 1 \)

- Compiles to:

  ```
  Mov a, b  
  Add a, 1  
  Mov eax, eax  
  Add eax, 1  
  ```

  Live: a, b
  Live: b, d, e
  Register Allocate: a \( \rightarrow \) eax, b \( \rightarrow \) eax
Uses and Definitions

• Every instruction/statement *uses* some set of variables
  – i.e. reads from them

• Every instruction/statement *defines* some set of variables
  – i.e. writes to them

• For a node/statement $s$ define:
  – $\text{use}[s]$ : set of variables used by $s$
  – $\text{def}[s]$ : set of variables defined by $s$

• Examples:
  – $a = b + c$ $\text{use}[s] = \{b, c\}$ $\text{def}[s] = \{a\}$
  – $a = a + 1$ $\text{use}[s] = \{a\}$ $\text{def}[s] = \{a\}$
Liveness, Formally

- A variable \( v \) is \textit{live} on edge \( e \) if there is:
  - a node \( n \) in the CFG such that \( \text{use}[n] \) contains \( v \), \textit{and}
  - a path from \( e \) to \( n \) such that for every statement \( t \) on the path, \( \text{def}[t] \) does not contain \( v \)

- The first clause says that \( v \) will be used on some path starting from edge \( e \)
- The second clause says that \( v \) won’t be redefined on that path before the use

- Questions:
  - How can we compute this efficiently?
  - How can we use this information (e.g. for register allocation)?
  - How does the choice of IR affect this?
Questions
Simple, inefficient algorithm

- A variable \( v \) is live on an edge \( e \) if there is a node \( n \) in the CFG using it \( \text{and} \) a directed path from \( e \) to \( n \) passing through no def of \( v \).

- Backtracking Algorithm:
  - For each use of a variable \( v \),
  - Follow all paths backwards through the control-flow graph until either \( v \) is defined or a previously visited node has been reached
  - Mark the variable \( v \) live across each edge traversed

- Inefficient because it explores the same paths many times (for different uses and different variables)
Dataflow Analysis

• Idea: compute liveness information for all variables simultaneously
  – Keep track of sets of information about each node

• Approach: define *equations* that should be true in any correct liveness algorithm

• Solve the equations by iteratively converging on a solution
  – Start with a rough approximation to the answer (e.g., nothing is live)
  – Refine the answer using the equations in each iteration
  – Keep going until no more refinement is possible: a *fixpoint* has been reached

• This is an instance of a general framework for computing program properties: dataflow analysis
Dataflow Value Sets for Liveness

- Nodes are program statements, so:
  - use[n]: set of variables used by n
  - def[n]: set of variables defined by n
  - in[n]: set of variables live on entry to n
  - out[n]: set of variables live on exit from n

- Associate in[n] and out[n] with the collected information about incoming/outgoing edges

- For liveness: what constraints are there among these sets?
Other Dataflow Constraints

- \( \text{in}[n] \) must include \( \text{use}[n] \): a variable must be live on entry to \( n \) if it is used by \( n \).

- \( \text{in}[n] \) must also include \( \text{out}[n] - \text{def}[n] \): if a variable is live on exit from \( n \), and \( n \) doesn’t define it, it is live on entry to \( n \).

- And \( \text{out}[n] \) should include \( \text{in}[n'] \) for every \( n' \in \text{succ}[n] \): if a variable is live on entry to a successor node of \( n \), it must be live on exit from \( n \).

- Equations:
  \[
  \text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])
  \]
  \[
  \text{out}[n] = \bigcup_{n'\in\text{succ}[n]} \text{in}[n']
  \]
Iterative Dataflow Analysis

- Equations:
  \[ \text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \]
  \[ \text{out}[n] = \bigcup_{n' \in \text{succ}[n]} \text{in}[n'] \]

- Find a solution to those constraints by starting from a rough guess
- Start with: \( \text{in}[n] = \emptyset \) and \( \text{out}[n] = \emptyset \) for every \( n \)
  - They don’t satisfy the equations yet!

- Iteratively re-compute \( \text{in}[n] \) and \( \text{out}[n] \) by applying the equations
  - Each iteration will add variables to the sets \( \text{in}[n] \) and \( \text{out}[n] \)

- Stop when \( \text{in}[n] \) and \( \text{out}[n] \) satisfy the equations, i.e., an iteration doesn’t add any new variables to any set
for all \( n \), \( \text{in}[n] := \emptyset \), \( \text{out}[n] := \emptyset \)

repeat until no change in \( \text{in} \) and \( \text{out} \)

for all \( n \)

\[
\text{out}[n] := \bigcup_{n' \in \text{succ}[n]} \text{in}[n']
\]

\[
\text{in}[n] := \text{use}[n] \cup (\text{out}[n] - \text{def}[n])
\]

end

end

• Finds a \textit{fixpoint} of the \textit{in} and \textit{out} equations
  – The algorithm is guaranteed to terminate... why?

• Why do we start with \( \emptyset \)?
Example Liveness Analysis

- Example flow graph:

```java
int def: e

int use: e

int in: e = 1

int out: e = 1

int in: if x > 0

int out: if x > 0

int in: z = e * e

int out: z = e * e

int in: y = e * x

int out: y = e * x

int in: x = x - 1

int out: x = x - 1

int in: if (x & 1)

int out: if (x & 1)

int in: e = z

int out: e = z

int in: e = y

int out: e = y

int in: return x

int out: return x
def: e

int use: x

int in: z = e * e

int out: z = e * e

int in: y = e * x

int out: y = e * x

int in: x = x - 1

int out: x = x - 1

int in: if (x & 1)

int out: if (x & 1)

int in: e = z

int out: e = z

int in: e = y

int out: e = y

int in: return x

int out: return x

int in: if x > 0

int out: if x > 0

int in: z = e * e

int out: z = e * e

int in: y = e * x

int out: y = e * x

int in: x = x - 1

int out: x = x - 1

int in: if (x & 1)

int out: if (x & 1)

int in: e = z

int out: e = z

int in: e = y

int out: e = y

int in: return x

int out: return x

int in: e = 1

int out: e = 1

int in: if x > 0

int out: if x > 0

int in: z = e * e

int out: z = e * e

int in: y = e * x

int out: y = e * x

int in: x = x - 1

int out: x = x - 1

int in: if (x & 1)

int out: if (x & 1)

int in: e = z

int out: e = z

int in: e = y

int out: e = y

int in: return x

int out: return x

e = 1;
while (x > 0) {
    
    z = e * e;
    y = e * x;
    x = x - 1;
    if (x & 1) {
        e = z;
    } else {
        e = y;
    }
}
return x;
```

\[ \text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \]
\[ \text{out}[n] = \bigcup_{n' \in \text{succ}[n]} \text{in}[n'] \]
Example Liveness Analysis

- **Iteration 1:**

  \[\text{in}[2] = x\]
  \[\text{in}[3] = e\]
  \[\text{in}[4] = x\]
  \[\text{in}[5] = e,x\]
  \[\text{in}[6] = x\]
  \[\text{in}[7] = x\]
  \[\text{in}[8] = z\]
  \[\text{in}[9] = y\]

\[\text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])\]
\[\text{out}[n] = \bigcup_{n' \in \text{succ}[n]} \text{in}[n']\]
Example Liveness Analysis

- **Iteration 2:**
  
  $\text{out}[1] = x$
  
  $\text{in}[1] = x$
  
  $\text{out}[2] = e, x$
  
  $\text{in}[2] = e, x$
  
  $\text{out}[3] = e, x$
  
  $\text{in}[3] = e, x$
  
  $\text{out}[5] = x$
  
  $\text{out}[6] = x$
  
  $\text{out}[7] = z, y$
  
  $\text{in}[7] = x, z, y$
  
  $\text{out}[8] = x$
  
  $\text{in}[8] = x, z$
  
  $\text{out}[9] = x$
  
  $\text{in}[9] = x, y$

$\text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])$

$\text{out}[n] = \bigcup_{n' \in \text{succ}[n]} \text{in}[n']$
Example Liveness Analysis

- Iteration 3:
  out[1] = e, x
  out[6] = x, y, z
  in[6] = x, y, z
  out[7] = x, y, z
  out[8] = e, x
  out[9] = e, x

in[n] = use[n] ∪ (out[n] – def[n])
out[n] = ∪_{n' ∈ succ[n]} in[n']
Example Liveness Analysis

- Iteration 4:
  out[5] = x,y,z
  in[5] = e,x,z

\[
\begin{align*}
in[n] &= \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \\
\text{out}[n] &= \bigcup_{n' \in \text{succ}[n]} \text{in}[n']
\end{align*}
\]
Example Liveness Analysis

- Iteration 5:
  out[3] = e, x, z

Done!

\[
\text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])
\]
\[
\text{out}[n] = \bigcup_{n' \in \text{succ}[n]} \text{in}[n']
\]
Questions

Top
Improving the Algorithm

- This algorithm requires us to go through every node in the graph in every iteration. How can we do better?
  \[ \text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \]
  \[ \text{out}[n] = \bigcup_{n' \in \text{succ}[n]} \text{in}[n'] \]

- The only nodes we update are the ones that have new information coming to them from their successors, using the rule
  \[ \text{out}[n] = \bigcup_{n' \in \text{succ}[n]} \text{in}[n'] \]
  - This is the only rule that involves more than one node

- So if a node’s successors haven’t changed, then the node itself won’t change.

- Keep track of which nodes’ successors have changed in an iteration, update only those nodes in the next iteration
A Worklist Algorithm

- Use a queue $w$ of nodes to be updated ("worklist")

for all $n$, $\text{in}[n] := \emptyset$, $\text{out}[n] := \emptyset$

$w$ = new queue with all nodes

repeat until $w$ is empty

let $n = w.\text{pop}()$  // pull a node off the queue
old_in = $\text{in}[n]$  // remember old $\text{in}[n]$

$\text{out}[n] := \bigcup_{n' \in \text{succ}[n]} \text{in}[n']$

$\text{in}[n] := \text{use}[n] \cup (\text{out}[n] - \text{def}[n])$

if (old_in != $\text{in}[n]$),  // if $\text{in}[n]$ has changed
  for all $m$ in $\text{pred}[n]$, $w.\text{push}(m)$  // add predecessors to worklist
end
Example Liveness Analysis

- Example flow graph:

```java
int x = 1;
while (x > 0) {
    int z = e * e;
    int y = e * x;
    x = x - 1;
    if (x & 1) {
        e = z;
    } else {
        e = y;
    }
}
return x;
```

\[
in[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])
\]
\[
\text{out}[n] = \bigcup_{n' \in \text{succ}[n]} \text{in}[n']
\]
Example Liveness Analysis

- Iteration 1:
  in[2] = x
  in[3] = e
  in[4] = x
  in[5] = e,x
  in[6] = x
  in[7] = x
  in[8] = z
  in[9] = y

\[
in[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])
\]

\[
\text{out}[n] = \bigcup_{n' \in \text{succ}[n]} \text{in}[n']
\]
Example Liveness Analysis

- Iteration 2:
  out[1] = x
  in[1] = x
  out[2] = e, x
  in[2] = e, x
  out[3] = e, x
  in[3] = e, x
  out[5] = x
  out[6] = x
  out[7] = z, y
  in[7] = x, z, y
  out[8] = x
  in[8] = x, z
  out[9] = x
  in[9] = x, y

in[n] = use[n] ∪ (out[n] – def[n])
out[n] = ∪_{n′ ∈ succ[n]} in[n′]
Example Liveness Analysis

- Iteration 3:
  out[1] = e, x
  out[6] = x, y, z
  in[6] = x, y, z
  out[7] = x, y, z
  out[8] = e, x
  out[9] = e, x

\[
\begin{align*}
in[n] &= \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \\
\text{out}[n] &= \bigcup_{n' \in \text{succ}[n]} \text{in}[n']
\end{align*}
\]
Example Liveness Analysis

- Iteration 4:
  out[5] = x, y, z
  in[5] = e, x, z

\[
in[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])
\]
\[
\text{out}[n] = \bigcup_{n' \in \text{succ}[n]} \text{in}[n']
\]
Example Liveness Analysis

- Iteration 5:
  out[3] = e, x, z

Done!

in[n] = use[n] ∪ (out[n] − def[n])
out[n] = ∪_{n' ∈ succ[n]} in[n']
Questions

Top
Dataflow Analysis: Summary

• Goal: collect some information about every statement/instruction/block in a program

• Approach: propagate information along the edges of a control flow graph
  – Set up equations describing how the information at one node relates to the information at adjacent nodes
  – Start with an initial state, and then repeatedly use the equations to update the information for each node
  – When we reach a fixpoint (no change in an iteration), we’re done!

• Questions we can answer:
  – What variables are still going to be used at this point?
  – What values can each variable have at this point?
  – What expressions have already been computed at this point?

• Result: control flow graph with information for each node
  – What good is this? Depends on the information!