Announcements

• HW5: Oat v.2 out
  • Due Tuesday 19 Nov

• HW6 will be released Tuesday 12 Nov
  • 3 weeks to complete
Today

• Dataflow analysis
• Liveness analysis
  • Worklist algorithm
• Generalizing dataflow analysis
  • Available expressions
  • Reaching definitions
Motivating Code Analyses

- There are lots of things that might influence the safety/applicability of an optimization
  - 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?
- Today: algorithms and data structures useful for answering these questions
Moving Towards Register Allocation

- Oat compiler currently generates as many temporary variables as needed
  - The %uids that you are very familiar with...

- Current compilation strategy:
  - Each %uid maps to a stack location
  - Yields programs with many loads/stores to memory
  - Very inefficient!

- Ideally, map as many %uid’s as possible into registers.
  - Eliminate the use of the `alloca` instruction?
  - Only 16 max registers available on 64-bit X86
  - %rsp and %rbp are reserved and some have special semantics, so really only 10 or 12 available
  - This means that a register must hold more than one slot

- When is this safe?
• Observation: `%uid1` and `%uid2` can be assigned to the same register if their values will not be needed at the same time.
  - A `%uid` is “needed” if its contents will be used as a source operand in a later instruction.

• Such a variable is called “live”

• Two variables can share the same register if they are not live at the same time.
Scope vs. Liveness

- We can already get some coarse liveness information from variable scoping.
- Consider the following Oat program:

```c
int f(int x) {
    var a = 0;
    if (x > 0) {
        var b = x * x;
        a = b + b;
    }
    var c = a * x;
    return c;
}
```

- Note that due to Oat’s scoping rules, variables `b` and `c` can never be live at the same time.
  - `c`’s scope is disjoint from `b`’s scope
- So, we could assign `b` and `c` to the same `alloca`’ed slot and potentially to the same register.
But Scope is too Coarse

• Consider this program:

```c
int f(int x) {
    int a = x + 2;
    int b = a * a;
    int c = b + x;
    return c;
}
```

• The scopes of `a, b, c, x` all overlap – they’re all in scope at the end of the block.

• But `a, b, c` are never live at the same time.
  • So they can share the same stack slot / register
Live Variable Analysis

• Variable v is **live** at a program point if v is defined before the program point and used after it.
• Liveness is defined in terms of where variables are defined and where variables are used.
• Liveness analysis: Compute the live variables between each statement.
  • May be conservative (i.e., may claim a variable is live when it isn’t)
    • Safe approximation!
  • To be useful, it should be more precise than simple scoping rules.
• Liveness analysis is one example of dataflow analysis
  • Other examples: Available Expressions, Reaching Definitions, Constant-Propagation Analysis, …
Control-flow Graphs Revisited

- Recall: a basic block is a sequence of instructions such that:
  - There is a distinguished, labeled entry point (no jumps into the middle of a basic block)
  - There is a (possibly empty) sequence of non-control-flow instructions
  - The block ends with a single control-flow instruction (jump, conditional branch, return, etc.)

- In a control flow graph (CFG), nodes are basic blocks
  - 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.

- Note: the following slides are intentionally ambiguous about the exact nature of the code in the CFGs
  - CFGs and dataflow analysis work for x86 assembly, imperative C-like source, LLVM IR, ...
  - Same general idea, but the exact details differ
  - e.g. LLVM IR doesn’t have “imperative” update of %uid temporaries. SSA structure of the LLVM IR makes some of these analyses simpler.
Dataflow over CFGs

- For precision, it is helpful to think of the “fall through” between sequential instructions as an edge of the control-flow graph too.
- Different implementation tradeoffs in practice…
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
  ```

  Register Allocate:
  - \( a \rightarrow \text{eax}, b \rightarrow \text{eax} \)
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:
  • use[s] : set of variables used by s
  • 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

• Variable \( v \) is **live** on edge \( e \) if:
  • (1) there is a node \( n \) in the CFG such that \( \text{use}[n] \) contains \( v \), and
  • (2) there is a directed path from \( e \) to \( n \) such that for every statement \( s' \) on the path, \( \text{def}[s'] \) does not contain \( v \)

• Clause (1) says that \( v \) will be used on some path starting from edge \( e \)
• Clause (2) says that \( v \) won’t be redefined on that path before the use

• Questions:
  • How to compute this efficiently?
  • How to use this information (e.g., for register allocation)?
  • How does the choice of IR affect this?
    (e.g. LLVM IR uses SSA, so it doesn’t allow redefinition, which simplifies liveness analysis)
Simple, inefficient algorithm

• “A variable v is live on an edge e if there is a node n in the CFG using it and a directed path from e to n that does not define v”

• Backtracking Algorithm:
  • For each variable v…
    • Try all paths from each use of v, tracing backwards through the control-flow graph until either v is defined or a previously visited node has been reached.
    • Mark the variable v live on 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 must be satisfied by any liveness determination
  - Equations based on “obvious” constraints.

• **Solve the equations** by iteratively converging on a solution.
  - Start with a “rough” approximation to the answer
  - Refine the answer at 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, for each \( n \), define the following sets:
  • \( \text{use}[n] \) : set of variables used by \( n \)
  • \( \text{def}[n] \) : set of variables defined by \( n \)
  • \( \text{in}[n] \) : set of variables live on entry to \( n \)
  • \( \text{out}[n] \) : set of variables live on exit from \( n \)

• Associate \( \text{in}[n] \) and \( \text{out}[n] \) with the “collected” information about incoming/outgoing edges
  • i.e., \( \text{out}[n] \) is union of all liveness information on outgoing edges of \( n \)

• For liveness, what constraints are there among these sets?
**Liveness Dataflow Constraints**

- We have: \( \text{in}[n] \supseteq \text{use}[n] \)
  
  - “A variable must be live on entry to \( n \) if it is used by \( n \)”

- Also: \( \text{in}[n] \supseteq \text{out}[n] - \text{def}[n] \)
  
  - “If a variable is live on exit from \( n \), and \( n \) doesn’t define it, then it is live on entry to \( n \)”
  
  - Note: here ‘–’ means “set difference”

- And: \( \text{out}[n] \supseteq \text{in}[n'] \) if \( n' \in \text{succ}[n] \)
  
  - “If a variable is live on entry to a successor node of \( n \), it must be live on exit from \( n \).”
Iterative Dataflow Analysis

• Find a solution to those constraints by starting from a rough guess.
• Start with: \( \text{in}[n] = \emptyset \) and \( \text{out}[n] = \emptyset \)
• They don’t satisfy the constraints:
  • \( \text{in}[n] \supseteq \text{use}[n] \)
  • \( \text{in}[n] \supseteq \text{out}[n] - \text{def}[n] \)
  • \( \text{out}[n] \supseteq \text{in}[n'] \) if \( n' \in \text{succ}[n] \)
• Idea: iteratively re-compute \( \text{in}[n] \) and \( \text{out}[n] \) where forced to by the constraints
  • Each iteration will add variables to the sets \( \text{in}[n] \) and \( \text{out}[n] \) (i.e. the live variable sets will increase monotonically)
• We stop when \( \text{in}[n] \) and \( \text{out}[n] \) satisfy these equations:
  (which are derived from the constraints above)
  • \( \text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \)
  • \( \text{out}[n] = \bigcup_{n' \in \text{succ}[n]} \text{in}[n'] \)
Complete Liveness Analysis Algorithm

- Finds a fixpoint of the in and out equations.
  - The algorithm is guaranteed to terminate… Why?
- Why do we start with $\emptyset$?
Example Liveness Analysis

- Example flow graph:

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

Each iteration update:

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

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

- **Iteration 1:**

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

  (showing only updates that make a change)
Example Liveness Analysis

Each iteration update:

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

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

- Iteration 2:
  
  \[
  \begin{align*}
  \text{out}[1] &= x \\
  \text{in}[1] &= x \\
  \text{out}[2] &= e, x \\
  \text{in}[2] &= e, x \\
  \text{out}[3] &= e, x \\
  \text{in}[3] &= e, x \\
  \text{out}[5] &= x \\
  \text{in}[5] &= x \\
  \text{out}[6] &= x \\
  \text{in}[6] &= x \\
  \text{out}[7] &= z, y \\
  \text{in}[7] &= x, z, y \\
  \text{out}[8] &= x \\
  \text{in}[8] &= x, z \\
  \text{out}[9] &= x \\
  \text{in}[9] &= x, y \\
  \end{align*}
  \]
Example Liveness Analysis

Each iteration update:

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

• Iteration 3:
  \[
  \text{out}[1] = e, x
  \]
  \[
  \text{out}[6] = x, y, z
  \]
  \[
  \text{in}[6] = x, y, z
  \]
  \[
  \text{out}[7] = x, y, z
  \]
  \[
  \text{out}[8] = e, x
  \]
  \[
  \text{out}[9] = e, x
  \]
Example Liveness Analysis

Each iteration update:

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

- **Iteration 4:**
  
  \[
  \begin{align*}
  \text{out}[5] &= x, y, z \\
  \text{in}[5] &= e, x, z
  \end{align*}
  \]
Example Liveness Analysis

Each iteration update:

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

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

- **Iteration 5:**
  \[ \text{out}[3] = e, x, z \]

Done!
Improving the Algorithm

• Can we do better?

• Observe: the only way information propagates from one node to another is using:

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

• This is the only rule that involves more than one node

• If the in sets of a node’s successors haven’t changed, then the node itself won’t change!

• Idea for an improved version of the algorithm:
  • Keep track of which node’s successors have changed
A Worklist Algorithm

• Use a FIFO queue of nodes that might need to be updated.

for all n, in[n] := Ø, out[n] := Ø
w = new queue with all nodes
repeat until w is empty
   let n = w.pop()       // pull a node off the queue
   old_in = in[n]       // remember old in[n]
   out[n] := \bigcup_{n' \in \text{succ}[n]} in[n']
   in[n] := use[n] \cup (out[n] – \text{def}[n])
   if (old_in != in[n]),       // if in[n] has changed
      for all m in pred[n], w.push(m) // add to worklist
end
Generalizing Dataflow Analyses

• The kind of iterative constraint solving used for liveness analysis applies to other kinds of analyses as well
  • Available expressions analysis
  • Reaching definitions analysis
  • Alias Analysis
  • Constant Propagation
Available Expressions

• An expression $e$ is available at program point $p$ if on all paths from the entry to $p$, expression $e$ is computed at least once, and there are no intervening assignment to $x$ or to the free variables of $e$

• If $e$ is available at $p$, we do not need to re-compute $e$
  • (i.e., for common sub-expression elimination)

• How do we compute the available expressions at each program point?
Available Expressions Example

1. $\emptyset$
2. $\emptyset$
3. $\{a+b\}$
4. $\{a+b, a*b\}$
5. $\{a+b, a*b\}$
6. $\{a+b, a*b\}$
7. $\{a+b, a*b\}$
8. $\emptyset$
9. $\{a+b\}$

$\begin{align*}
x & := a + b; \\
y & := a \times b;
\end{align*}$

$y > a$

$a := a + 1;$

$x := a + b$

$\emptyset$

$\{a+b\}$

$\{a+b\}$

$\{a+b\}$

$\{a+b\}$

$\{a+b\}$

$\{a+b\}$

$\{a+b\}$

$\{a+b\}$

$\{a+b\}$

$\{a+b\}$

$\{a+b\}$

$\{a+b\}$

$\{a+b\}$

$\{a+b\}$

(Numbers indicate the order that the facts are computed in this example.)
Reaching definitions

• A definition of a variable $v$ is an assignment to $v$
• A definition of variable $v$ reaches point $p$ if
  • There is a path from the definition of $v$ to $p$
  • There is no intervening assignment to $v$ on that path
  • Also called def-use information
Common Framework: Gen-Kill

• Can think of all these dataflow analysis as computing facts at program points
  • \text{in}[n] \text{ is set of facts that hold immediately before before } n
  • \text{out}[n] \text{ is set of facts that hold immediately before before } n

• Each instruction \text{n generates some facts, and kills some facts}
  • E.g., liveness: \text{in}[n] := \text{use}[n] \cup (\text{out}[n] – \text{def}[n])
  • Generates \text{use}[n] and kills \text{def}[n]

• Analyses differ on:
  • Which facts we are computing and which facts instructions gen and kill
  • Forward or backwards
    • Forwards: compute \text{out}[n] using \text{in}[n]
    • Backwards: compute \text{in}[n] using \text{out}[n]
  • How to combine facts: may or must
    • Must: compute facts which must be true, by intersect-ing facts
    • May: compute facts that may be true, by union-ing facts
Comparing Dataflow Analyses

**Liveness:**
backward may analysis

- Facts = variables that are live
- \( \text{gen}[n] = \text{use}[n] \)
  \( \text{kill}[n] = \text{def}[n] \)
- \( \text{out}[n] := \bigcup_{n' \in \text{succ}[n]} \text{in}[n'] \)
- \( \text{in}[n] := \text{gen}[n] \cup (\text{out}[n] - \text{kill}[n]) \)

**Available Expressions:**
forward must analysis

- Facts = expressions that are available
- \( \text{gen}[n] = \text{expressions evaluated} \)
  \( \text{kill}[n] = \text{expressions containing a variable in } \text{def}[n] \)
- \( \text{in}[n] := \bigcap_{n' \in \text{pred}[n]} \text{out}[n'] \)
- \( \text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n]) \)

**Reaching Definitions:**
forward may analysis

- Facts = definitions (i.e., instructions that assign)
- \( \text{gen}[n] = \{ n \} \text{ if } n \text{ defines variables} \)
  \( \text{kill}[n] = \{ n' \mid n' \text{ defines a variable in } \text{def}[n] \} \)
- \( \text{in}[n] := \bigcup_{n' \in \text{pred}[n]} \text{out}[n'] \)
- \( \text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n]) \)