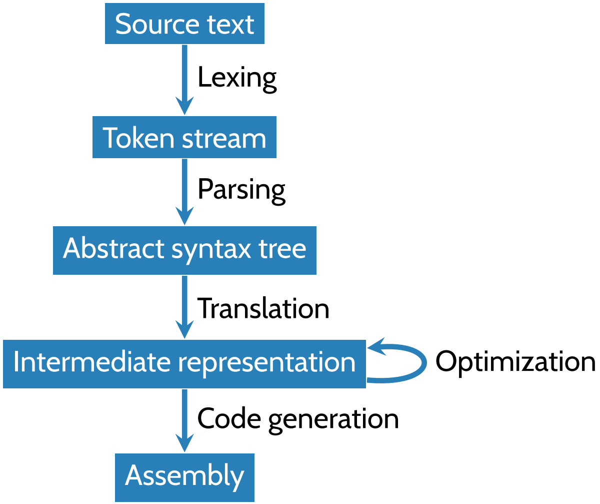
COS320: Compiling Techniques

Zak Kincaid

April 6, 2019

Optimization

Compiler phases (simplified)



Optimization

- Optimization operates as a sequeence of IR-to-IR transformations. Each transformation is expected to:
  - *improve performance* (time, space, power)
  - *not change the high-level behavior of the program*
- Optimization simplifies compiler writing
  - More modular: can translate to IR in a simple-buf-inefficient way, then optimize
- Optimization simplifies programming
  - Programmer can spend less time thinking about low-level performance issues
  - More portable: compiler can take advantage of the characteristics of a particular machine
- Already seen a few examples so far...

Algebraic simplification

Idea: replace complex expressions with simpler / cheaper ones

$$e * 1 \to e$$

$$0 + e \to e$$

$$2 * 3 \to 6$$

$$-(-e) \to e$$

$$e * 4 \to e \ll 2$$

$$\ldots$$

Loop unrolling

Idea: avoid branching by trading space for time.

```
long array_sum (long *a, long n) {
  long i;
  long sum = 0;
  for (i = 0; i < n; i++) {
    sum += *(a + i);
  }
  return sum;
}
```

→

```
long array_sum (long *a, long n) {
  long i;
  long sum = 0;
  for (i = 0; i < n % 4; i++) {
    sum += *(a + i);
  }
  for (; i < n; i += 4) {
    sum += *(a + i);
    sum += *(a + i + 1);
    sum += *(a + i + 2);
    sum += *(a + i + 3);
  }
  return sum;
}
```

Strength reduction

Idea: replace expensive operation (e.g., multiplication) w/ cheaper one (e.g., addition).

```c
long trace (long *m, long n) {
  long i;
  long result = 0;
  for (i = 0; i < n; i++) {
    result += *(m + i*n + i);
  }
  return result;
}
```

→

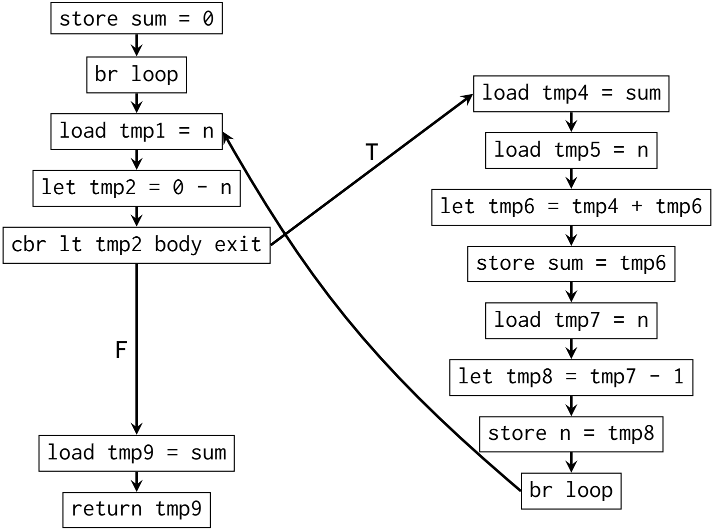
```c
long trace (long *m, long n) {
  long i;
  long result = 0;
  long *next = m;
  for (i = 0; i < n; i++) {
    result += *next;
    next += i + 1;
  }
  return result;
}
```

Optimization and Analysis

- *Program analysis*: conservatively approximate the run-time behavior of a program at compile time.
  - Type inference: find the type of value each expression will evaluate to at run time. *Conservative* in the sense that the analysis will abort if it cannot find a type for a variable, even if one exists.
  - Constant propagation: if a variable only holds on value at run time, find that value. *Conservative* in the sense that analysis may fail to find constant values for variables that have them.
- Optimization passes are typically informed by analysis
  - Analysis lets us know which transformations are safe
  - Conservative analysis $\Rightarrow$ never perform an unsafe optimization, but may miss some safe optimizations.

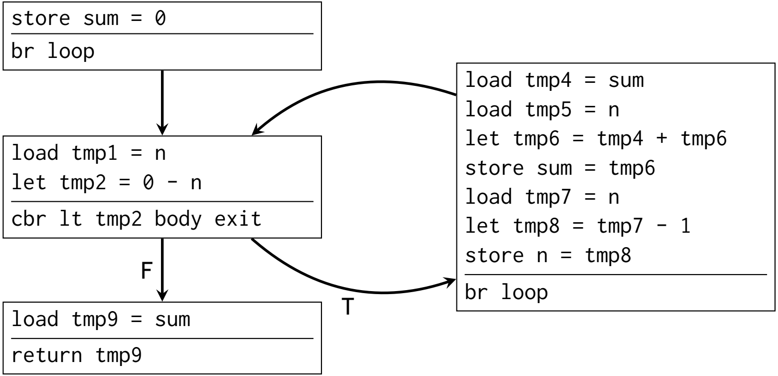
Control Flow Graphs (CFG)

```
int sum_upto(int n) {
  int sum = 0;
  while (n > 0) {
    sum += n;
    n--;
  }
  return sum;
}
```



Control Flow Graphs (CFG)

```
int sum_upto(int n) {
  int sum = 0;
  while (n > 0) {
    sum += n;
    n--;
  }
  return sum;
}
```



- Control flow graphs are one of the basic data structures used to represent programs in many program analyses
- Recall: A *control flow graph* (CFG) for a procedure P is a directed, rooted graph $G = (N, E, r)$ where
  - The nodes are basic blocks of P
  - There is an edge $n_i \to n_j \in E$ iff n_j may execute immediately after n_i
  - There is a distinguished entry block r where the execution of the procedure begins
- Some additional vocabulary:
  - Define *pred*$(n) = \{m \in N : m \to n \in E\}$ (control flow predecessors)
  - Define *succ*$(n) = \{m \in N : n \to m \in E\}$ (control flow successors)
  - Path = sequence of nodes $n_1, \ldots, n_k$ such that for each i, there is an edge from $n_i \to n_{i+1} \in E$

Simple imperative language

- Suppose that we have the following language:

```
  <instr> ::= <var> = add <opn>, <opn>
            | <var> = mul <opn>, <opn>
            | <var> = opn
    <opn> ::= <int> | <var>
  <block> ::= <instr><block> | <term>
   <term> ::= blez <opn>, <label>, <label>
<program> ::= <program> <label> : <block> | <block>
```

- Note: no uids, no SSA
  - We'll take a look at how SSA affects program analysis later

Constant propagation

- The goal of constant propagation: determine at each instruction I a *constant environment*
  - A constant environment is a symbol table mapping each variable x to one of:
    - an integer n (indicating that x's value is n whenever the program is at I)
    - $\top$ (indicating that x might take more than one value at I)
    - $\bot$ (indicating that x may take no values at run-time – I is unreachable)
  - Can place an information ordering on these values: $\bot \preceq n \preceq \top$ (most information to least information)
- Motivation: can compute expressions at compile time to save on run time

```
x = add 1, 2
y = mul x, 11
z = add x, y
```

$\longrightarrow$

```
x = 3
y = 33
z = 36
```

Propagating constants through instructions

- Goal: given a constant environment C and an instruction
  - $x = \texttt{add}, opn_1, opn_2$
  - $x = \texttt{mul}, opn_1, opn_2$
  - $x = opn$

  Assuming that constant environment C holds *before* the instruction, what is the constant environment *after* the instruction?

Propagating constants through instructions

- Goal: given a constant environment C and an instruction
  - $x = \texttt{add}, opn_1, opn_2$
  - $x = \texttt{mul}, opn_1, opn_2$
  - $x = opn$

  Assuming that constant environment C holds *before* the instruction, what is the constant environment *after* the instruction?
- Define an evaluator for operands:

$$\textit{eval}(\textit{opn}, C) = \begin{cases} C(\textit{opn}) & \text{if opn is a variable} \\ \textit{opn} & \text{if opn is an int} \end{cases}$$

- Define an evaluator for instructions

$$post(instr, C) = \begin{cases} C & \text{if } C is \bot \\ C\{x \mapsto eval(opn, C)\} & \text{if instr is } x = opn \\ C\{x \mapsto \top\} & \text{if } eval(opn_1, C) = \top \vee eval(opn_2, C) = \top \\ C\{x \mapsto eval(opn_1, C) + eval(opn_2, C)\} & \text{if instr is } x = \texttt{add } \textit{opn}_1, \textit{opn}_2 \\ C\{x \mapsto eval(opn_1, C) * eval(opn_2, C)\} & \text{if instr is } x = \texttt{mul } \textit{opn}_1, \textit{opn}_2 \end{cases}$$

Propagating constants through basic blocks

- How do we propagate a constant environment through a basic block?

Propagating constants through basic blocks

- How do we propagate a constant environment through a basic block?
- Block takes the form $instr_1, ..., instr_n, term$.
  take $post(block, C) = post(instr_n, ...post(instr_1, C))$

Propagating constants through the control flow graph

- Let $G = (N, E, s)$ be a control flow graph.
- *cp* is the *smallest*[1] function such that
  - $cp(s) = \{x_1 \mapsto \top, ..., x_n \mapsto \top\}$
  - For each $p \to n \in E$, $post(p, cp(p)) \leq cp(n)$

$cp(s) = \{x_1 \mapsto \top, ..., x_n \mapsto \top\}$;
$cp(n) = \{x_1 \mapsto \bot, ..., x_n \mapsto \bot\}$ for all other nodes;
work $\leftarrow N \setminus \{s\}$; `/* Set of nodes that may violate spec */`
while *work* $\neq \emptyset$ **do**
  Pick some n from work;
  work $\leftarrow$ *work* $\setminus \{n\}$;
  $C \leftarrow \bigsqcup_{p \in pred}$ *post*$(p,$ *cp*$(p))$;
  if $d \neq cp(n)$ **then**
    cp$(n) \leftarrow C$;
    work $\leftarrow$ *work* $\cup$ *succ*(n)

[1]Pointwise order: $f \leq g$ if for all nodes n and all variables x, $f(n)(x) \preceq g(n)(x)$