Optimization
Compiler phases (simplified)

- Source text
  - Lexing
- Token stream
  - Parsing
- Abstract syntax tree
  - Translation
  - Optimization
  - Intermediate representation
  - Code generation
  - Assembly
Optimization

- Optimization operates as a sequence 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

\[ e \times 1 \rightarrow e \]
\[ 0 + e \rightarrow e \]
\[ 2 \times 3 \rightarrow 6 \]
\[ -(e) \rightarrow e \]
\[ e \times 4 \rightarrow e \ll 2 \]

...
Loop unrolling

Idea: avoid branching by trading space for time.

```c
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;
}
```
Idea: replace expensive operation (e.g., multiplication) with cheaper one (e.g., addition).

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

```
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 ⇒ never perform an unsafe optimization, but may miss some safe optimizations.
```c
int sum_up_to(int n) {
    int sum = 0;
    while (n > 0) {
        sum += n;
        n--;
    }
    return sum;
}
```
Control Flow Graphs (CFG)

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

---

```
store sum = 0
br loop

load tmp1 = n
let tmp2 = 0 - n
cbr lt tmp2 body exit

load tmp4 = sum
load tmp5 = n
let tmp6 = tmp4 + tmp6
store sum = tmp6
load tmp7 = n
let tmp8 = tmp7 - 1
store n = tmp8
br loop

load tmp9 = sum
return tmp9
```
• 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 $n_i \rightarrow n_j \in E$ 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 $\text{pred}(n) = \{ m \in N : m \rightarrow n \in E \}$ (control flow predecessors).
  - Define $\text{succ}(n) = \{ m \in N : n \rightarrow m \in E \}$ (control flow successors).
  - Path = sequence of nodes $n_1, ..., n_k$ such that for each $i$, there is an edge from $n_i \rightarrow n_{i+1} \in E$. 
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
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

\begin{align*}
x &= \text{add} \ 1, \ 2 \\
y &= \text{mul} \ x, \ 11 \\
z &= \text{add} \ x, \ y
\end{align*}

\begin{align*}
x &= 3 \\
y &= 33 \\
z &= 36
\end{align*}
Propagating constants through instructions

• Goal: given a constant environment \( C \) and an instruction
  • \( x = \text{add}, \; opn_1, \; opn_2 \)
  • \( x = \text{mul}, \; opn_1, \; opn_2 \)
  • \( x = opn \)

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
  • \( x = \text{add}, \ opn_1, \ opn_2 \)
  • \( x = \text{mul}, \ opn_1, \ opn_2 \)
  • \( x = \ opn \)

Assuming that constant environment \( C \) holds before the instruction, what is the constant environment after the instruction?

• Define an evaluator for operands:

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

• Define an evaluator for instructions

\[
\text{post}(\text{instr}, \ C) = \begin{cases} 
C & \text{if } C \text{ is } \perp \\
C[x \mapsto \text{eval}(\ opn, \ C)] & \text{if instr is } x = \ opn \\
C[x \mapsto \top] & \text{if eval}(\ opn_1, \ C) = \top \lor \text{eval}(\ opn_2, \ C) = \top \\
C[x \mapsto \text{eval}(\ opn_1, \ C) + \text{eval}(\ opn_2, \ C)] & \text{if instr is } x = \text{add} \ \ opn_1, \ opn_2 \\
C[x \mapsto \text{eval}(\ opn_1, \ C) \times \text{eval}(\ opn_2, \ C)] & \text{if instr is } x = \text{mul} \ \ opn_1, \ opn_2
\end{cases}
\]
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 $\text{instr}_1, ..., \text{instr}_n, \text{term}$.
  take $\text{post}(\text{block}, C) = \text{post}(\text{instr}_n, ... \text{post}(\text{instr}_1, C))$
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
  - $cp(s) = \{x_1 \mapsto \top, \ldots, x_n \mapsto \top\}$
  - For each $p \rightarrow n \in E$, $post(p, cp(p)) \leq cp(n)$

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

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