Flow Graphs

- A flow graph is a graphical depiction of a sequence of instructions with control flow edges.
- A flow graph can be defined at the intermediate code level or target code level.

Basic Blocks

- A basic block is a sequence of consecutive instructions with exactly one entry point and one exit point (with natural flow or a branch instruction).
Basic Blocks and Control Flow Graphs

- A control flow graph (CFG) is a directed graph with basic blocks $B_i$ as vertices and with edges $B_i \rightarrow B_j$ if $B_j$ can be executed immediately after $B_i$.

Successor and Predecessor Blocks

- Suppose the CFG has an edge $B_1 \rightarrow B_2$
  - Basic block $B_1$ is a predecessor of $B_2$
  - Basic block $B_2$ is a successor of $B_1$

Partition Algorithm for Basic Blocks

**Input:** A sequence of three-address statements

**Output:** A list of basic blocks with each three-address statement in exactly one block

1. Determine the set of leaders, the first statements in basic blocks
   - a) The first statement is the leader
   - b) Any statement that is the target of a goto is a leader
   - c) Any statement that immediately follows a goto is a leader
2. For each leader, its basic block consist of the leader and all statements up to but not including the next leader or the end of the program
Loops

- A loop is a collection of basic blocks, such that
  - All blocks in the collection are strongly connected
  - The collection has a unique entry, and the only way to reach a block in the loop is through the entry

Loops (Example)

- Strongly connected components:
  - SCC=({B2,B3}, {B4})

- Entries:
  - B3, B4

Equivalence of Basic Blocks

- Two basic blocks are (semantically) equivalent if they compute the same set of expressions

  \[
  \begin{align*}
  b &= 0 \\
  t1 &= a + b \\
  t2 &= c \cdot t1 \\
  a &= t2 \\
  b &= 0 \\
  \end{align*}
  \]

  \[
  \begin{align*}
  a &= c \cdot a \\
  b &= 0 \\
  \end{align*}
  \]

  Blocks are equivalent, assuming t1 and t2 are dead: no longer used (no longer live)
Transformations on Basic Blocks

- A **code-improving transformation** is a code optimization to improve speed or reduce code size
- **Global transformations** are performed across basic blocks
- **Local transformations** are only performed on single basic blocks
- Transformations must be safe and preserve the meaning of the code
  - A local transformation is safe if the transformed basic block is guaranteed to be equivalent to its original form

Common-Subexpression Elimination

- Remove redundant computations

```
<table>
<thead>
<tr>
<th>a := b + c</th>
<th>a := b + c</th>
</tr>
</thead>
<tbody>
<tr>
<td>b := a - d</td>
<td>b := a - d</td>
</tr>
<tr>
<td>c := b + c</td>
<td>c := b + c</td>
</tr>
<tr>
<td>d := a - d</td>
<td>d := b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t1 := b * c</th>
<th>t1 := b * c</th>
</tr>
</thead>
<tbody>
<tr>
<td>t2 := a - t1</td>
<td>t2 := a - t1</td>
</tr>
<tr>
<td>t3 := b + c</td>
<td>t3 := b + c</td>
</tr>
<tr>
<td>t4 := t2 + t3</td>
<td>t4 := t2 + t3</td>
</tr>
</tbody>
</table>
```

Dead Code Elimination

- Remove unused statements

```
| b := a + 1 |
| a := b + c |
|----------|----------|

Assuming `a` is dead (not used)

```
if true goto L2
b := a + y
```

Remove unreachable code
Renaming Temporary Variables

• Temporary variables that are dead at the end of a block can be safely renamed

```
\begin{align*}
t_1 &:= b + c \\
t_2 &:= a - t_1 \\
t_3 &:= t_1 \times d \\
d &:= t_2 + t_3 \\
\end{align*}
\Rightarrow \begin{align*}
t_1 &:= b + c \\
t_2 &:= a - t_1 \\
t_3 &:= t_1 \times d \\
d &:= t_2 + t_3 \\
\end{align*}
```

Normal-form block

Interchange of Statements

• Independent statements can be reordered

```
\begin{align*}
t_1 &:= b + c \\
t_2 &:= a - t_1 \\
t_3 &:= t_1 \times d \\
d &:= t_2 + t_3 \\
\end{align*}
\Rightarrow \begin{align*}
t_1 &:= b + c \\
t_2 &:= a - t_1 \\
t_3 &:= t_1 \times d \\
d &:= t_2 + t_3 \\
\end{align*}
```

Note that normal-form blocks permit all statement interchanges that are possible

Algebraic Transformations

• Change arithmetic operations to transform blocks to algebraic equivalent forms

```
\begin{align*}
t_1 &:= a - a \\
t_2 &:= b + t_1 \\
t_3 &:= 2 \times t_2 \\
\end{align*}
\Rightarrow \begin{align*}
t_1 &:= 0 \\
t_2 &:= b \\
t_3 &:= t_2 \ll 1 \\
\end{align*}
```
Next-Use

- Next-use information is needed for dead-code elimination and register assignment
- Next-use is computed by a backward scan of a basic block and performing the following actions on statement
  \[ i : x := y \text{ op } z \]
  - Add liveness/next-use info on \(x\), \(y\), and \(z\) to statement \(i\)
  - Set \(x\) to "not live" and "no next use"
  - Set \(y\) and \(z\) to "live" and the next uses of \(y\) and \(z\) to \(i\)

Next-Use (Step 1)

\[i : a := b + c\]

\[j : t := a + b\]

| live(a) = true, live(b) = true, live(t) = true, nextuse(a) = none, nextuse(b) = none, nextuse(t) = none |

Attach current live/next-use information
Because info is empty, assume variables are live
(Data flow analysis Ch.10 can provide accurate information)

Next-Use (Step 2)

\[i : a := b + c\]

\[j : t := a + b\]

| live(a) = true, live(b) = true, live(t) = true, nextuse(a) = none, nextuse(b) = none, nextuse(t) = none |

Compute live/next-use information at \(j\)
Next-Use (Step 3)

\[ i: \quad a := b + c \quad \left\{ \text{live}(a) = \text{true}, \text{live}(b) = \text{true}, \text{live}(c) = \text{false}, \text{nextuse}(a) = j, \text{nextuse}(b) = j, \text{nextuse}(c) = \text{none} \right\} \]

\[ j: \quad t := a + b \quad \left\{ \text{live}(a) = \text{true}, \text{live}(b) = \text{true}, \text{live}(t) = \text{true}, \text{nextuse}(a) = \text{none}, \text{nextuse}(b) = \text{none}, \text{nextuse}(t) = \text{none} \right\} \]

Attach current live/next-use information to \( i \)

Next-Use (Step 4)

\[ i: \quad a := b + c \quad \left\{ \text{live}(a) = \text{false}, \text{nextuse}(a) = \text{none}, \text{live}(b) = \text{true}, \text{nextuse}(b) = i, \text{live}(c) = \text{false}, \text{nextuse}(c) = \text{none} \right\} \]

\[ j: \quad t := a + b \quad \left\{ \text{live}(a) = \text{false}, \text{live}(b) = \text{false}, \text{live}(t) = \text{false}, \text{nextuse}(a) = \text{none}, \text{nextuse}(b) = \text{none}, \text{nextuse}(t) = \text{none} \right\} \]

Compute live/next-use information \( i \)

A Code Generator

- Generates target code for a sequence of three-address statements using next-use information
- Uses new function \( \text{getreg} \) to assign registers to variables
- Computed results are kept in registers as long as possible, which means:
  - Result is needed in another computation
  - Register is kept up to a procedure call or end of block
- Checks if operands to three-address code are available in registers
The Code Generation Algorithm

- For each statement $x := y \ op z$
  1. Set location $L = \text{getreg}(y, z)$
  2. If $y \notin L$, then generate
     $\text{MOV} \ y', L$
     where $y'$ denotes one of the locations where the value of $y$ is available (choose register if possible)
  3. Generate
     $\text{OP} \ z', L$
     where $z'$ is one of the locations of $z$
     Update register/address descriptor of $x$ to include $L$
  4. If $y$ and/or $z$ has no next use and is stored in register, update register descriptors to remove $y$ and/or $z$

Register and Address Descriptors

- A register descriptor keeps track of what is currently stored in a register at a particular point in the code, e.g. a local variable, argument, global variable, etc: $\text{MOV} \ a, R0$  "$R0$ contains $a$"
- An address descriptor keeps track of the location where the current value of the name can be found at run time, e.g. a register, stack location, memory address, etc: $\text{MOV} \ a, R0$
  $\text{MOV} \ R0, R1$  "$a$ in $R0$ and $R1$"

The getreg Algorithm

- To compute $\text{getreg}(y, z)$
  1. If $y$ is stored in a register $R$ and $R$ only holds the value $y$, and $y$ has no next use, then return $R$;
     Update address descriptor: value $y$ no longer in $R$
  2. Else, return a new empty register if available
  3. Else, find an occupied register $R$;
     Store contents (register spill) by generating
     $\text{MOV} \ R, M$
     for every $M$ in address descriptor of $y$;
     Return register $R$
  4. Return a memory location
Code Generation Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Code Generated</th>
<th>Register Description</th>
<th>Address Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t := a - b$</td>
<td>MOV a, R0, SUB b, R0</td>
<td>R0 contains $t$ in R0</td>
<td></td>
</tr>
<tr>
<td>$u := a - c$</td>
<td>MOV a, R1, SUB c, R1</td>
<td>R1 contains $u$ in R1</td>
<td></td>
</tr>
<tr>
<td>$v := t + u$</td>
<td>ADD R1, R0</td>
<td>R0 contains $v$ in R1</td>
<td></td>
</tr>
<tr>
<td>$d := v + u$</td>
<td>ADD R1, R0, MOV R0, d</td>
<td>R0 contains $d$ in R0 and memory</td>
<td></td>
</tr>
</tbody>
</table>

Register Allocation and Assignment

- The `getreg` algorithm is simple but sub-optimal
  - All live variables in registers are stored (flushed) at the end of a block
- Global register allocation assigns variables to limited number of available registers and attempts to keep these registers consistent across basic block boundaries
  - Keeping variables in registers in looping code can result in big savings

Allocating Registers in Loops

- Suppose loading a variable $x$ has a cost of 2
- Suppose storing a variable $x$ has a cost of 2
- Benefit of allocating a register to a variable $x$ within a loop $L$ is
  $$\sum_{B \in L} (\text{use}(x, B) + 2 \times \text{live}(x, B))$$
  where $\text{use}(x, B)$ is the number of times $x$ is used in $B$ and $\text{live}(x, B) = \text{true}$ if $x$ is live on exit from $B$
Global Register Allocation with Graph Coloring

- When a register is needed but all available registers are in use, the content of one of the used registers must be stored (spilled) to free a register
- Graph coloring allocates registers and attempts to minimize the cost of spills
- Build a conflict graph (interference graph)
- Find a $k$-coloring for the graph, with $k$ the number of registers

Register Allocation with Graph Coloring: Example

```plaintext
a := read();
b := read();
c := read();
a := a + b + c;
if (a < 10) {
    d := c + 8;
    write(d);
} else if (a < 20) {
    e := 10;
    d := e + a;
    write(e);
} else {
    f := 12;
    d := f + a;
    write(f);
}
write(d);
```

Register Allocation with Graph Coloring: Live Ranges

Interference graph: connected vars have overlapping ranges

Live range of $b$
Register Allocation with Graph Coloring: Solution

Interference graph

Solve

Three registers:

\[ a = r_2 \]
\[ b = r_3 \]
\[ c = r_1 \]
\[ d = r_2 \]
\[ e = r_1 \]
\[ f = r_1 \]

\[
x_2 := \text{read();}
x_3 := \text{read();}
x_1 := \text{read();}
x_2 := x_2 + x_3 + x_1;
\]

if \( x_2 < 10 \) {
\[
x_2 := x_1 + 8;
write(x_1);
\]
} else if \( x_2 < 20 \) {
\[
x_1 := 10;
x_2 := x_1 + x_2;
write(x_1);
\]
} else {
\[
x_1 := 12;
x_2 := x_1 + x_2;
write(x_1);
\]
write(x_2);

Peephole Optimization

- Examines a short sequence of target instructions in a window (peephole) and replaces the instructions by a faster and/or shorter sequence when possible
- Applied to intermediate code or target code
- Typical optimizations:
  - Redundant instruction elimination
  - Flow-of-control optimizations
  - Algebraic simplifications
  - Use of machine idioms

Peephole Opt: Eliminating Redundant Loads and Stores

- Consider

\[
\begin{align*}
\text{MOV R0,} & \ a \\
\text{MOV a,} & \ R0
\end{align*}
\]

- The second instruction can be deleted, but only if it is not labeled with a target label
  - Peephole represents sequence of instructions with at most one entry point
- The first instruction can also be deleted if \( \text{live(a)} = \text{false} \)
Peephole Optimization: Deleting Unreachable Code

- Unlabeled blocks can be removed

\[
\begin{align*}
\text{if } 0 &= 0 \text{ goto } L2 \\
\text{b := } x + y \\
\end{align*}
\]

Peephole Optimization: Branch Chaining

- Shorten chain of branches by modifying target labels

\[
\begin{align*}
\text{if } a &= 0 \text{ goto } L2 \\
\text{b := } x + y \\
L2: \text{ goto } L3 \\
\end{align*}
\]

Peephole Optimization: Other Flow-of-Control Optimizations

- Remove redundant jumps

\[
\begin{align*}
\text{goto } L1 \\
L1: \\
\end{align*}
\]
Other Peephole Optimizations

- **Reduction in strength**: replace expensive arithmetic operations with cheaper ones
  
  \[
  a := x^2 \\
  b := y / 8 \\
  \text{\rightarrow} \\
  a := x \cdot x \\
  b := y \gg 3
  \]

- Utilize machine idioms
  
  \[
  a := a + 1 \\
  \text{\rightarrow} \\
  \text{inc a}
  \]

- Algebraic simplifications
  
  \[
  a := a + 0 \\
  b := b + 1 \\
  \text{\rightarrow}
  \]

}\]