Single Static Assignment and Unboxing
Compiler Temporaries

Code Generation
- $e_1 + e_2$: Store $e_1$ while computing $e_2$.

Common Sub-Expression Elimination
- Store sub-expression for reuse later.

Loop Invariants
- Compute and store expression outside of loop.
# Storage Locations

<table>
<thead>
<tr>
<th>Memory</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pro</strong>: plentiful.</td>
<td><strong>Pro</strong>: very fast.</td>
</tr>
<tr>
<td><strong>Con</strong>: slow.</td>
<td><strong>Con</strong>: only 8 (16) available.</td>
</tr>
<tr>
<td><strong>Con</strong>: must update indices (or waste space) after code elimination.</td>
<td><strong>Con</strong>: reuse limits code motion.</td>
</tr>
</tbody>
</table>
Storage Locations

<table>
<thead>
<tr>
<th>Storage Locations</th>
<th>Memory</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pro</strong>: plentiful.</td>
<td></td>
<td><strong>Pro</strong>: very fast.</td>
</tr>
<tr>
<td><strong>Con</strong>: slow.</td>
<td></td>
<td><strong>Con</strong>: only 8 (16) available.</td>
</tr>
<tr>
<td><strong>Con</strong>: must update indices (or waste space) after code elimination.</td>
<td></td>
<td><strong>Con</strong>: reuse limits code motion.</td>
</tr>
</tbody>
</table>

We want it all: *no reuse, no indexing*, and *plentiful* storage.

- Also, fast would be nice.
Static Single Assignment (SSA)

Add “temporary locations” to intermediate representation.

- *Infinite number* (effectively) of these.
- Each location assigned *exactly once*.
- Implement these as registers later (register allocation).

3-Address IR: $x$ becomes $x_1, x_2, ...$

For Cool ASM IR: add registers $t0, t1, ...$
SSA and a Basic Block

```plaintext
mul r1 <- r1 r1
li r2 <- 1
add r0 <- r0 r2
add r0 <- r0 r1
```
SSA and a Basic Block

\[ \text{mul } r1 \leftarrow r1 \text{ r1} \]
\[ \text{li } r2 \leftarrow 1 \]
\[ \text{add } r0 \leftarrow r0 \text{ r2} \]
\[ \text{add } r0 \leftarrow r0 \text{ r1} \]

\[ \text{mul } t1 \leftarrow r1 \text{ r1} \]
\[ \text{li } t2 \leftarrow 1 \]
\[ \text{add } t3 \leftarrow r0 \text{ t2} \]
\[ \text{add } t4 \leftarrow t3 \text{ t1} \]
SSA and a Control Flow Graph (CFG)

L:  
- mul r1 <- r1 r1  
- li r2 <- 1  
- add r0 <- r0 r2  
- li r2 <- 10  
- ble r0 r2 L
SSA and a Control Flow Graph (CFG)

L: mul r1 <- r1 r1
    li r2 <- 1
    add r0 <- r0 r2
    li r2 <- 10
    bles r0 r2 L

L: mul t1 <- r1 r1
    li t2 <- 1
    add t3 <- r0 t2
    li t4 <- 10
    bles t3 t4 L
SSA and a Control Flow Graph (CFG)

This is no longer being updated!
Loop no longer terminates!
In general, one logical value may reach a basic block along more than one path.

Insert a \( \phi \)-function to “merge” those values.

Not 3-address: one argument per incoming path.

L: \( t5 \leftarrow \phi(r0,t3) \)

\begin{align*}
\text{mul } t1 & \leftarrow r1 \ r1 \\
\text{li } t2 & \leftarrow 1 \\
\text{add } t3 & \leftarrow t5 \ t2 \\
\text{li } t4 & \leftarrow 10 \\
\text{ble } t3 \ t4 & \text{ L}
\end{align*}
Converting CFG to SSA Form

General Algorithm:

1. Find *dominators* for every CFG node.
   - Every path to node $N$ goes through its dominators.

2. Compute *dominance frontiers* for every CFG node.
   - Set of nodes just barely *not* dominated by $N$.

3. Place $\phi$-functions at frontiers.

4. Rename assignments and subsequent uses.
Converting CFG to SSA Form

General Algorithm:
- Required if your language uses gotos.
- Algorithmically efficient (data-flow, plus two tree walks).

 Dominators:
- Also useful for identifying natural loops (gotos again).
Generating SSA Form Directly

\[
\begin{align*}
\text{program} &::= \ [\text{class}]^{+} \\
\text{class} &::= \text{class} \ \text{TYPE} \ \text{[inherits} \ \text{TYPE]} \ \{ \ \text{[feature;]}^{*} \} \\
\text{feature} &::= \text{ID}( \ [\text{formal}[,\text{formal}]^{*}] \ ) : \ \text{TYPE} \ \{ \ \text{expr} \} \\
& \quad \mid \ \text{ID} : \ \text{TYPE} \ [\ < \ \text{expr} \ ] \\
\text{formal} &::= \ \text{ID} : \ \text{TYPE} \\
\text{expr} &::= \ \text{ID} \ < \ \text{expr} \\
& \quad \mid \ \text{expr}[@\text{TYPE}].\text{ID}( \ [\ \text{expr}[,\text{expr}]^{*}] \ ) \\
& \quad \mid \ \text{ID}( \ [\ \text{expr}[,\text{expr}]^{*}] \ ) \\
& \quad \mid \ \text{if} \ \text{expr} \ \text{then} \ \text{expr} \ \text{else} \ \text{expr} \ \text{fi} \\
& \quad \mid \ \text{while} \ \text{expr} \ \text{loop} \ \text{expr} \ \text{pool} \\
& \quad \mid \ \{ \ [\text{expr;}^{+}] \} \\
& \quad \mid \ \text{let} \ \text{ID} : \ \text{TYPE} \ [\ < \ \text{expr} \ ] \ {[\text{ID} : \ \text{TYPE} \ [\ < \ \text{expr} \ ]^{*}} \ \text{in} \ \text{expr} \\
& \quad \mid \ \text{case} \ \text{expr} \ \text{of} \ \{\ \text{ID} : \ \text{TYPE} \ => \ \text{expr;}^{+} \text{esac} \\
& \quad \mid \ \text{new} \ \text{TYPE} \\
& \quad \mid \ \text{isvoid} \ \text{expr}
\end{align*}
\]
Generating SSA Form Directly

Most expressions: store result in temporary.
Generating SSA Form Directly

Control-flow: insert $\phi$-function for assignments in body.
SSA Code Generation of Control-Flow

(*local x*)

if b then
  x <- 1
else
  x <- 2
fi
SSA Code Generation of Control-Flow

(*local x*)
if b then
  x <- 1
else
  x <- 2
fi

brz t1 L1 ; true branch
call Int..new
mov t2 <- r1
li t3 <- 1
st t2[2] <- t3
jmp L2

L1:
; false branch
call Int..new
mov t4 <- r1
li t5 <- 2
st t4[2] <- t5
L2:
t6 <- \phi(t2,t4)
SSA Code Generation of Control-Flow

if and case expressions:
◦ Also insert $\phi$-functions for the expression value.
◦ $x \leftarrow \text{if } b \text{ then new Int else new String fi}$

while expressions:
◦ Insert $\phi$-function at top for variables used then modified.
◦ $\text{while } x < 10 \text{ loop } x \leftarrow x + 1 \text{ pool}$
Auto-Unboxing
Boxed Types

All Cool values are objects.

- Fewer special cases in language.
  - Handle for any value of any type can be stored in a (integer) register.
  - No need for void methods: everything satisfies type Object.

- Simpler generic data structures and methods.
  - Java7 has 9 System.out.print() methods.
  - java.util.List holds Integers, but not ints.
Unboxed Types

Many uses of Int (or Bool or String) do not use objects.
  ◦ Paying for flexibility we do not use!

<table>
<thead>
<tr>
<th>$x + y$</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unboxed</td>
<td>1 ALU</td>
</tr>
<tr>
<td>Boxed</td>
<td>1 ALU + 3 Memory + 1 constructor call*</td>
</tr>
</tbody>
</table>

We want to work with raw values wherever possible.
Unboxing: Intuition

Define type UnboxedInt
  ◦ Integer literals have type UnboxedInt
  ◦ Instruction \texttt{boxi} : UnboxedInt \rightarrow \texttt{Int}
  ◦ Instruction \texttt{unboxi} : \texttt{Int} \rightarrow UnboxedInt
  ◦ Arithmetic operators apply to UnboxedInt

\[
x + y \Rightarrow \text{box}(\text{unbox}(x) + \text{unbox}(y))
\]
(a + b) + c
(a + b) + c

unboxi t1 <- r2
unboxi t2 <- r3
add t3 <- t1 t2
boxi t4 <- t3
unboxi t5 <- t4
unboxi t6 <- r4
add t7 <- t5 t6
boxi t8 <- r7
(a + b) + c

unboxi t1 <- r2
unboxi t2 <- r3
add t3 <- t1 t2
boxi t4 <- t3
unboxi t5 <- t4
unboxi t6 <- r4
add t7 <- t5 t6
boxi t8 <- r7

Peephole optimization:
Can save constructor call, plus 2 memory ops.
(a + b) + c

unboxi t1 <- r2
unboxi t2 <- r3
add t3 <- t1 t2
mov t5 <- t3
unboxi t6 <- r4
add t7 <- t5 t6
boxi t8 <- r7

Was that safe?
Something a Little More Complicated

```
let i : Int <- 1 in
while i < 10 loop {
  x <- x * 2;
  i <- i + 1;
} pool;

x
```
Something a Little More Complicated

li t1 <- 1
boxi t2 <- t1
L1:
unboxi t3 <- t2
li t4 <- 10
boxi t5 <- t4
unboxi t6 <- t5
ble t6 t3 L2

unboxi t7 <- r0
li t8 <- 2
boxi t9 <- t8
unboxi t10 <- t9
mul t11 <- t7 t10
boxi t12 <- t11
unboxi t13 <- t2
li t14 <- 1
boxi t15 <- t14
unboxi t16 <- t15
add t17 <- t13 t16
boxi t18 <- t17
jmp L1
L2:
mov r1 <- t12
### Something a Little More Complicated

<table>
<thead>
<tr>
<th>Code Block</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>li t1 &lt;- 1</code></td>
</tr>
<tr>
<td><code>boxi t2 &lt;- t1</code></td>
</tr>
<tr>
<td><code>L1:</code></td>
</tr>
<tr>
<td><code>t19 &lt;- \phi(t2, t18)</code></td>
</tr>
<tr>
<td><code>t20 &lt;- \phi(r0, t12)</code></td>
</tr>
<tr>
<td><code>unboxi t3 &lt;- t19</code></td>
</tr>
<tr>
<td><code>li t4 &lt;- 10</code></td>
</tr>
<tr>
<td><code>boxi t5 &lt;- t4</code></td>
</tr>
<tr>
<td><code>unboxi t6 &lt;- t5</code></td>
</tr>
<tr>
<td><code>ble t6 t3 L2</code></td>
</tr>
<tr>
<td><code>unboxi t7 &lt;- t20</code></td>
</tr>
<tr>
<td><code>li t8 &lt;- 2</code></td>
</tr>
<tr>
<td><code>boxi t9 &lt;- t8</code></td>
</tr>
<tr>
<td><code>unboxi t10 &lt;- t9</code></td>
</tr>
<tr>
<td><code>mul t11 &lt;- t7 t10</code></td>
</tr>
<tr>
<td><code>boxi t12 &lt;- t11</code></td>
</tr>
<tr>
<td><code>unboxi t13 &lt;- t2</code></td>
</tr>
<tr>
<td><code>li t14 &lt;- 1</code></td>
</tr>
<tr>
<td><code>boxi t15 &lt;- t14</code></td>
</tr>
<tr>
<td><code>unboxi t16 &lt;- t15</code></td>
</tr>
<tr>
<td><code>add t17 &lt;- t13 t16</code></td>
</tr>
<tr>
<td><code>boxi t18 &lt;- t17</code></td>
</tr>
<tr>
<td><code>jmp L1</code></td>
</tr>
<tr>
<td><code>L2:</code></td>
</tr>
<tr>
<td><code>mov r1 &lt;- t12</code></td>
</tr>
</tbody>
</table>
Something a Little More Complicated

```
li t1 <- 1
boxi t2 <- t1
L1:
t19 <- \phi(t2, t18)
t20 <- \phi(r0, t12)
unboxi t3 <- t19
li t4 <- 10
boxi t5 <- t4
unboxi t6 <- t5
ble t6 t3 L2
```

```
unboxi t7 <- t20
li t8 <- 2
boxi t9 <- t8
unboxi t10 <- t9
mul t11 <- t7 t10
boxi t12 <- t11
unboxi t13 <- t2
li t14 <- 1
boxi t15 <- t14
unboxi t16 <- t15
```

```
add t17 <- t13 t16
boxi t18 <- t17
jmp L1
L2:
mov r1 <- t12
```
Something a Little More Complicated

```
li t1 <- 1
boxi t2 <- t1
L1:
t19 <- \phi(t2, t18)
t20 <- \phi(r0, t12)
unboxi t3 <- t19
li t4 <- 10
mov t6 <- t4
ble t6 t3 L2

unboxi t7 <- t20
li t8 <- 2
mov t10 <- t8
mul t11 <- t7 t10
boxi t12 <- t11
unboxi t13 <- t2
li t14 <- 1
mov t16 <- t14
add t17 <- t13 t16
boxi t18 <- t17
jmp L1
L2:
mov r1 <- t12
```

A Data-Flow for Unboxing

Direction: Backward  Meet operator: Diamond

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊤</td>
<td>Unknown</td>
</tr>
<tr>
<td>used</td>
<td>Value is used (arithmetic or method call)</td>
</tr>
<tr>
<td>converted</td>
<td>Value is eventually converted.</td>
</tr>
<tr>
<td>⊥</td>
<td>Value is used and converted.</td>
</tr>
</tbody>
</table>
Something a Little More Complicated

li t1 <- 1
boxi t2 <- t1
L1:
t19 <- \phi(t2, t18)
t20 <- \phi(r0, t12)
unboxi t3 <- t19
li t4 <- 10
mov t6 <- t4
ble t6 t3 L2

unboxi t7 <- t20
li t8 <- 2
mov t10 <- t8
mul t11 <- t7 t10
boxi t12 <- t11
unboxi t13 <- t2
li t14 <- 1
mov t16 <- t14
add t17 <- t13 t16
boxi t18 <- t17
jmp L1

L2:
mov r1 <- t12
Something a Little More Complicated

li t1 <- 1
mov t2 <- t1
L1:
t19 <- φ(t2, t18)
t20 <- φ(r0, t12)
mov t3 <- t19
li t4 <- 10
mov t6 <- t4
ble t6 t3 L2

unboxi t7 <- t20
li t8 <- 2
mov t10 <- t8
mul t11 <- t7 t10
boxi t12 <- t11
unboxi t13 <- t2
li t14 <- 1
mov t16 <- t14
add t17 <- t13 t16
mov t18 <- t17
jmp L1
L2:
mov r1 <- t12
Something a Little More Complicated

li t1 <- 1
mov t2 <- t1
L1:
  t19 <- \phi(t2, t18)
t20 <- \phi(r0, t12)
mov t3 <- t19
li t4 <- 10
mov t6 <- t4
ble t6 t3 L2

unboxi t7 <- t20
li t8 <- 2
mov t10 <- t8
mul t11 <- t7 t10
boxi t12 <- t11
unboxi t13 <- t2
li t14 <- 1
mov t16 <- t14
add t17 <- t13 t16
mov t18 <- t17
jmp L1
L2:
mov r1 <- t12
Unboxing Summary

1. Insert type-casts to represent boxing and unboxing.

2. Use data-flow analysis to identify wasteful boxing.
   - *Argument values* must be boxed (for now).
   - *Return values* must be boxed (for now).
   - *Self objects* must be boxed.
Dominance

X *dominates* Y \((X \geq Y)\)
- If *every* path to Y goes through X.
- Note: \(X \geq X\)

X *strictly dominates* Y \((X > Y)\)
- If \(X \geq Y\), but \(X \neq Y\).

Find dominators with data-flow algorithm
- (Dragon Book, p658).
Dominance Trees
Dominance and $\phi$-Functions

Place $\phi$-function at node $N$ if
- 2 non-empty CFG paths define $v$,
- And both paths meet at $N$.

Note: $\phi$-function defines $v$.

I.e., place $\phi$-function along dominance frontier.

DF(1) = {1}
DF(2) = {7}
DF(3) = {6}
DF(4) = {6}
DF(5) = {1,7}
DF(6) = {7}
DF(7) = { }