Agenda

• Overview of SSA IR
  – Constructing SSA graphs
  – Sample of SSA-based optimizations
  – Converting back from SSA form

• Sources: Appel ch. 19, also an extended discussion in Cooper-Torczon sec. 9.3, Mike Ringenburg’s CSE 401 slides
Def-Use (DU) Chains

• Common dataflow analysis problem: Find all sites where a variable is used, or find the definition site of a variable used in an expression

• Traditional solution: def-use chains – additional data structure on top of the dataflow graph
  – Link each statement defining a variable to all statements that use it
  – Link each use of a variable to its definition
DU-Chain Drawbacks

• Expensive: if a typical variable has $N$ uses and $M$ definitions, the total cost per-variable is $O(N \times M)$
  – Would be nice if cost were proportional to the size of the program

• Unrelated uses of the same variable are mixed together
  – Complicates analysis
SSA: Static Single Assignment

• IR where each variable has only one definition in the program text
  – This is a single static definition, but that definition can be in a loop that is executed dynamically many times

• Makes many analyses (and associated optimizations) more efficient

• Complementary to CFG/DFG – better for some things, but cannot do everything
SSA in Basic Blocks

Idea: for each original variable \(v\), create a new variable \(v_n\) at the \(n^{th}\) definition of the original \(v\). Subsequent uses of \(v\) use \(v_n\) until the next definition point.

- Original
  
  \[
  \begin{align*}
  a &:= x + y \\
  b &:= a - 1 \\
  a &:= y + b \\
  b &:= x \times 4 \\
  a &:= a + b
  \end{align*}
  \]

- SSA
  
  \[
  \begin{align*}
  a_1 &:= x + y \\
  b_1 &:= a_1 - 1 \\
  a_2 &:= y + b_1 \\
  b_2 &:= x \times 4 \\
  a_3 &:= a_2 + b_2
  \end{align*}
  \]
Merge Points

• The issue is how to handle merge points

if (...) 
  a = x;
else 
  a = y;
  b = a;

if (...) 
  a₁ = x;
else 
  a₂ = y;
  b₁ = ??;
Merge Points

• The issue is how to handle merge points

  \[
  a_3 := \Phi(a_1, a_2)
  \]

  • Solution: introduce a \(\Phi\)-function

  \[
  \begin{align*}
  \text{if (…)} \\
  &a = x; \\
  \text{else} \\
  &a = y; \\
  &b = a;
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{if (…)} \\
  &a_1 = x; \\
  \text{else} \\
  &a_2 = y; \\
  &a_3 = \Phi(a_1, a_2); \\
  &b_1 = a_3;
  \end{align*}
  \]

  • Meaning: \(a_3\) is assigned either \(a_1\) or \(a_2\) depending on which control path is used to reach the \(\Phi\)-function
Another Example

Original

\[
\begin{align*}
    & b := M[x] \\
    & a := 0 \\
    & \text{if } b < 4 \\
    & a := b \\
    & c := a + b
\end{align*}
\]

SSA

\[
\begin{align*}
    & b_1 := M[x] \\
    & a_{1} := 0 \\
    & \text{if } b_1 < 4 \\
    & a_2 := b_1 \\
    & a_3 := \Phi(a_{1}, a_{2}) \\
    & c_{1} := a_3 + b_1
\end{align*}
\]
How Does $\Phi$ “Know” What to Pick?

• It doesn’t

• $\Phi$-functions don’t actually exist at runtime
  – When we’re done using the SSA IR, we translate back out of SSA form, removing all $\Phi$-functions
    • Basically by adding code to copy all SSA $x_i$ values to (the single, non-SSA) $x$
  – For analysis, all we typically need to know is the connection of uses to definitions – no need to “execute” anything
Example With a Loop

Original

```
a := 0
b := a + 1
c := c + b
a := b * 2
if a < N
  return c
```

SSA

```
a_1 := 0
a_3 := \Phi(a_1, a_2)
b_1 := \Phi(b_0, b_2)
c_2 := \Phi(c_0, c_1)
b_2 := a_3 + 1
c_1 := c_2 + b_2
a_2 := b_2 * 2
if a_2 < N
  return c_1
```

Notes:
- Loop-back edges are also merge points, so require \( \Phi \)-functions
- \( a_0, b_0, c_0 \) are initial values of \( a, b, c \) on block entry
- \( b_1 \) is dead – can delete later
- \( c \) is live on entry – either input parameter or uninitialized
Converting To SSA Form

• Basic idea
  – First, add $\Phi$-functions
  – Then, rename all definitions and uses of variables by adding subscripts
Inserting $\Phi$-Functions

• Could simply add $\Phi$-functions for every variable at every join point(!)

• But
  – Wastes *way* too much space and time
  – Not needed
Path-convergence criterion

• Insert a $\Phi$-function for variable $a$ at point $z$ when:
  – There are blocks $x$ and $y$, both containing definitions of $a$, and $x \neq y$
  – There are nonempty paths from $x$ to $z$ and from $y$ to $z$
  – These paths have no common nodes other than $z$
Details

• The start node of the flow graph is considered to define every variable (even if “undefined”)
• Each $\Phi$-function itself defines a variable, which may create the need for a new $\Phi$-function
  – So we need to keep adding $\Phi$-functions until things converge
• How can we do this efficiently?
  Use a new concept: dominance frontiers


Dominators

• Definition: a block $x$ *dominates* a block $y$ iff every path from the entry of the control-flow graph to $y$ includes $x$

• So, by definition, $x$ dominates $x$

• We can associate a Dom(inator) set with each CFG node $x$ – set of all blocks dominated by $x$
  
  \[ |\text{Dom}(x)| \geq 1 \]

• Properties:
  – Transitive: if $a$ dom $b$ and $b$ dom $c$, then $a$ dom $c$
  – There are no cycles, thus can represent the dominator relationship as a tree
Example
Dominators and SSA

• One property of SSA is that definitions dominate uses; more specifically:
  – If $x := \Phi(..., x_i, ...) \text{ is in block } b$, then the definition of $x_i$ dominates the $i^{\text{th}}$ predecessor of $b$
  – If $x$ is used in a non-$\Phi$ statement in block $b$, then the definition of $x$ dominates block $b$
Dominance Frontier (1)

• To get a practical algorithm for placing $\Phi$-functions, we need to avoid looking at all combinations of nodes leading from $x$ to $y$

• Instead, use the dominator tree in the flow graph
Dominance Frontier (2)

• Definitions
  – $x$ strictly dominates $y$ if $x$ dominates $y$ and $x \neq y$
  – The dominance frontier of a node $x$ is the set of all nodes $w$ such that $x$ dominates a predecessor of $w$, but $x$ does not strictly dominate $w$
    • This means that $x$ can be in it’s own dominance frontier! That can happen if there is a back edge to $x$ (i.e., $x$ is the head of a loop)

• Essentially, the dominance frontier is the border between dominated and undominated nodes
Example

= x

= DomFrontier(x)

= StrictDom(x)
Example

\[ x = \text{DomFrontier}(x) \]

\[ y = \text{StrictDom}(x) \]
Example

= x

= DomFrontier(x)

= StrictDom(x)
Example

\[
\begin{array}{c}
\text{DomFrontier}(x) = x \\
\text{StrictDom}(x) = \text{DomFrontier}(x)
\end{array}
\]
Example

1 = DomFrontier(x)
2 = x
3 = StrictDom(x)

UW CSE 401 Winter 2017
Example

\[
\begin{align*}
\text{DomFrontier}(x) &= \text{StrictDom}(x) \\
= x &= 5
\end{align*}
\]

UW CSE 401 Winter 2017
Example

\[ x = \text{DOMFrontier}(x) \]
\[ = \text{StrictDom}(x) \]
Example

= \DomFrontier(x)

= \StrictDom(x)
Example

= x

= \text{DomFrontier}(x)

= \text{StrictDom}(x)
Example

\[ = x \]
\[ = \text{DomFrontier}(x) \]
\[ = \text{StrictDom}(x) \]
Example

\[ x = \text{DomFrontier}(x) \]

\[ x = \text{StrictDom}(x) \]
Dominance Frontier Criterion for Placing $\Phi$-Functions

• If a node $x$ contains the definition of variable $a$, then every node in the dominance frontier of $x$ needs a $\Phi$-function for $a$
  
  — Idea: Everything dominated by $x$ will see $x$’s definition of $a$. The dominance frontier represents the first nodes we could have reached via an alternative path, which will have an alternate reaching definition (recall we say the entry node defines everything)
    • Why is this right for loops? Hint: strict dominance...
  
  — Since the $\Phi$-function itself is a definition, this placement rule needs to be iterated until it reaches a fixed-point

• Theorem: this algorithm places exactly the same set of $\Phi$-functions as the path criterion given previously
Placing $\Phi$-Functions: Details

- See the book for the full construction, but the basic steps are:
  1. Compute the dominance frontiers for each node in the flowgraph
  2. Insert just enough $\Phi$-functions to satisfy the criterion. Use a worklist algorithm to avoid reexamining nodes unnecessarily
  3. Walk the dominator tree and rename the different definitions of each variable $a$ to be $a_1, a_2, a_3, ...$
SSA Optimizations

• Why go to the trouble of translating to SSA?
• The advantage of SSA is that it makes many optimizations and analyses simpler and more efficient
  – We’ll give a couple of examples
• But first, what do we know? (i.e., what information is kept in the SSA graph?)
SSA Data Structures

• Statement: links to containing block, next and previous statements, variables defined, variables used.

• Variable: link to its (single) definition statement and (possibly multiple) use sites

• Block: List of contained statements, ordered list of predecessors, successor(s)
Dead-Code Elimination

• A variable is live iff its list of uses is not empty(!)
  – That’s it! Nothing further to compute

• Algorithm to delete dead code:
  while there is some variable \( v \) with no uses
    if the statement that defines \( v \) has no other side effects, then delete it
  – Need to remove this statement from the list of uses for its operand variables – which may cause those variables to become dead
Sparse Simple Constant Propagation

• If $c$ is a constant in $v := c$, any use of $v$ can be replaced by $c$
  - Then update every use of $v$ to use constant $c$
• If the $c_i$’s in $v := \Phi(c_1, c_2, \ldots, c_n)$ are all the same constant $c$, we can replace this with $v := c$
• Can also incorporate copy propagation, constant folding, and others in the same worklist algorithm
Simple Constant Propagation

\[
W := \text{list of all statements in SSA program}
\]

while \( W \) is not empty

remove some statement \( S \) from \( W \)

if \( S \) is \( v := \Phi(c, c, \ldots, c) \), replace \( S \) with \( v := c \)

if \( S \) is \( v := c \)

delete \( S \) from the program

for each statement \( T \) that uses \( v \)

substitute \( c \) for \( v \) in \( T \)

add \( T \) to \( W \)
Converting Back from SSA

- Unfortunately, real machines do not include a Φ instruction
- So after analysis, optimization, and transformation, need to convert back to a “Φ-less” form for execution
Translating $\Phi$-functions

• The meaning of $x := \Phi(x_1, x_2, ..., x_n)$ is “set $x := x_1$ if arriving on edge 1, set $x := x_2$ if arriving on edge 2, etc.”

• So, for each $i$, insert $x := x_i$ at the end of predecessor block $i$

• Rely on copy propagation and coalescing in register allocation to eliminate redundant moves
SSA Wrapup

- There are many details needed to fully and efficiently implement SSA, but these are the main ideas
- Not a silver bullet – some optimizations still need non-SSA forms – but it allows efficient implementation of many optimizations
- SSA is used in most modern optimizing compilers (llvm is based on it) and has been retrofitted into many older ones (gcc is a well-known example)