SSA and DFAs

Simone Campanoni
simonec@eecs.northwestern.edu
SSA Outline

• SSA and why?

• Reaching definitions, constant propagation with SSA forms

• Generate SSA code
Def-use chain

\[
\text{OUT[ENTRY]} = \{ \} ;
\]

for (each instruction \(i\) other than ENTRY) \(\text{OUT}[i] = \{ \} ;\)

while (changes to any OUT occur)

for (each instruction \(i\) other than ENTRY) {

\[
\text{IN}[i] = \bigcup_{\text{p a predecessor of } i} \text{OUT}[p] ;
\]

\[
\text{OUT}[i] = \text{GEN}[i] \cup (\text{IN}[i] - \text{KILL}[i]) ;
\]

}

\[
i: t \leftarrow \ldots
\]

\[
\text{GEN}[i] = \{ i \}
\]

\[
\text{KILL}[i] = \text{\textit{defs}(t)} \leftarrow \{ i \}
\]

• Given a variable \(t\),
  we need to find all definitions of \(t\) in the CFG
• How can we do it in LLVM?

\[
i: \ldots
\]

\[
\text{GEN}[i] = \{ \}
\]

\[
\text{KILL}[i] = \{ \}
\]
It’s a Static Single Assignment (SSA) representation

- A variable is set only by one instruction in the function body
  
  \[
  \%\text{myVar} = \ldots
  \]

- A static assignment can be executed more than once

\[
\text{While (\ldots)}\{
  \%\text{myVar} = \ldots
\}
\]
SSA and not SSA example

\[
\text{float myF (float par1, float par2, float par3)}\{
    \text{return (par1 * par2) + par3; }
\}
\]

\[
\text{define float @myF(float %par1, float %par2, float %par3) } \{
    \%1 = fmul float %par1, %par2
    \%1 = fadd float \%1, %par3
    \text{ret float \%1 }
\}
\]

\[
\text{define float @myF(float %par1, float %par2, float %par3) } \{
    \%1 = fmul float %par1, %par2
    \%2 = fadd float %1, %par3
    \text{ret float \%2 }
\}
\]
Consequences of SSA

• Unrelated uses of the same variable in source code become different variables in the SSA form

• Use—def chain are greatly simplified

• Data-flow analysis are simplified (... in the next slides)

• Code analysis (e.g., data flow analysis) can be designed to run faster

```
v = 5;
print(v);
v = 42;
print(v)
```

To SSA IR

```
v1 = 5
call print(v1)
v2 = 42
call print(v2)
```

No WAW, WAR data dependencies between variables!
Motivation for SSA

• Code analysis needs to represent facts at every program point

```c
define float @myF(float %par1, float %par2, float %par3) {
    %1 = fmul float %par1, %par2
    %2 = fadd float %1, %par3
    ret float %2
}
```

• What if
  • There are a lot of facts and there are a lot of program points?
  • Potentially takes a lot of space/time
    • Code analyses run slow
    • Compilers run slow
Example

\[ x := 3 \]

\[ a > b \]

\[ y := a + b \]

\[ y := a - b \]

\[ z := 2 \times y \]

\[ y := y \times 10 \]

\[ w := y + z \]

\[ w := w + y \]

\[ z := w + x \]
Sparse representation

- Instead, we’d like to use a sparse representation
  - Only propagate facts about $x$ where they’re needed

- Exploit **static single assignment** form
  - Each variable is defined (assigned to) exactly once
  - Definitions dominate their uses
Static Single Assignment (SSA)

Add **SSA edges** from definitions to uses
- No intervening statements define variable
- Safe to propagate facts about \( x \) only along SSA edges

Why can’t we do in non-SSA IRs?

In our llvm passes call->uses()
What about join nodes in the CFG?

- Add $\Phi$ functions to model joins
  - One argument for each incoming branch
- Operationally
  - Selects one of the arguments based on how control flow reach this node
- The backend needs to eliminate $\Phi$ nodes

\[
\begin{align*}
b &= c + 1 \\
b &= d + 1 \\
\text{If } (b > N) &\quad \text{Not SSA}
\end{align*}
\]

\[
\begin{align*}
b_1 &= c + 1 \\
b_2 &= d + 1 \\
\text{If } (? > N) &\quad \text{Still not SSA}
\end{align*}
\]

\[
\begin{align*}
b_1 &= c + 1 \\
b_2 &= d + 1 \\
{b_3} &= \Phi(b_1, b_2) \\
\text{If } (b_3 > N) &\quad \text{SSA}
\end{align*}
\]
Eliminating $\Phi$

- Basic idea: $\Phi$ represents facts that value of join may come from different paths
  - So just set along each possible path

\[
b_3 = \Phi(b_1, b_2)
\]

\[
\begin{align*}
b_1 &= c + 1 \\
b_2 &= d + 1
\end{align*}
\]

\[
\begin{align*}
\text{If } (b_3 > N) &: \\
\quad b_3 &= b_1 \\
\quad b_3 &= b_2
\end{align*}
\]

Not SSA
Eliminating $\Phi$ in practice

• Copies performed at $\Phi$ may not be useful
• Joined value may not be used later in the program
  (So why leave it in?)

• Use dead code elimination to kill useless $\Phi$s
• Subsequent register allocation will map the variables onto the actual set of machine register
SSA efficiency in practice

Fig. 21. Number of $\phi$-functions versus number of program statements.
Consequences of SSA

• Unrelated uses of the same variable in source code become different variables in the SSA form

\[
\begin{align*}
v &= 5; \\
\text{print}(v); \\
v &= 42; \\
\text{print}(v)
\end{align*}
\]

\[
\begin{align*}
v1 &= 5 \\
call \text{print}(v1) \\
v2 &= 42 \\
call \text{print}(v2)
\end{align*}
\]

• Use—def chain are greatly simplified

• **Data-flow analysis are simplified**

• Code analysis (e.g., data flow analysis) can be designed to run faster
Def-use chain

\[
\begin{align*}
\text{OUT}[\text{ENTRY}] &= \{ \}; \\
\text{for each instruction } i \text{ other than } \text{ENTRY} & \quad \text{OUT}[i] = \{ \}; \\
\text{while (changes to any OUT occur) } & \\
\text{for each instruction } i \text{ other than } \text{ENTRY} & \\
\quad \text{IN}[i] &= \bigcup_{\rho \text{ a predecessor of } i} \text{OUT}[\rho]; \\
\quad \text{OUT}[i] &= \text{GEN}[i] \cup (\text{IN}[i] - \text{KILL}[i]); \\
\end{align*}
\]

\[
\begin{align*}
i: t &\leftarrow \ldots \\
\text{GEN}[i] &= \{i\} \\
\text{KILL}[i] &= \text{defs}(t) - \{i\} \\
i: \ldots & \\
\text{GEN}[i] &= \{\} \\
\text{KILL}[i] &= \{\} \\
\end{align*}
\]
Def-use chain with SSA

\[
\text{OUT}[\text{ENTRY}] = \{ \};
\]

for (each instruction \(i\) other than ENTRY) \(\text{OUT}[i] = \{ \}\); while (changes to any OUT occur) for (each instruction \(i\) other than ENTRY) { \[
\text{IN}[i] = \bigcup_{p \text{ a predecessor of } i} \text{OUT}[p];
\]
\[
\text{OUT}[i] = \text{GEN}[i] \cup (\text{IN}[i] \setminus \text{KILL}[i]);
\]
}

\[
i: t \leftarrow \ldots\\
\text{GEN}[i] = \{i\}
\]
\[
\text{KILL}[i] = \{}
\]

\[
i: \ldots\\
\text{GEN}[i] = \{}\\
\text{KILL}[i] = \{}
\]
Question answered by reaching definition analysis:

does the definition “i” reach “j”?

Code example

\[ j : b1 = b0 + 1 \]

\[ i : b0 = 1 \]

\[ ? : b0 = b0 + 2 \]
Does it mean we can always propagate constants to variable uses?

What are the definitions of \( b_3 \) that reach “z”?
SSA in LLVM

- The IR must be in SSA all the time
  - Checked at boundaries of passes
  - No time wasted converting automatically IR to its SSA form
  - CAT designed with this constraint in mind
- \( \Phi \) instructions only at the top of a basic block
- Must have exactly 1 entry for every predecessor
- Must have at least one entry
- May include `undef` values
SSA in LLVM: changing variable values

• Let’s say we have a LLVM IR variable and we want to add code to change its value

• How should we do it?

\[
\text{Builder.CreateAdd}(v, \text{const1})
\]

And then?

\[
\text{replaceAllUsesWith}
\]
SSA in LLVM: changing variable values

• Let’s say we need to generate LLVM IR for the C code
  
  ```
  int n = n+1;
  ```

• How should we do it?

• Memory isn’t in SSA, just registers (e.g., stack locations---alloca)

• Exploit already existing passes to reduce inefficiencies (mem2reg)

  ```
  opt –mem2reg mybitcode.bc –o mybitcode.bc
  ```

• mem2reg maps memory locations to registers when possible
The mem2reg LLVM pass

```c
int ssa1() {
    int z = f() + 1;
    return z;
}
```

```llvm
define i32 @ssa1() nounwind {
  entry:
    %call = call i32 @f()
    %add = add nsw i32 %call, 1
    ret i32 %add
}
```

- alloca in the entry block
- only used by load and store
mem2reg might add new instructions

```c
int ssa2() {
    int y, z;
    y = f();
    if (y < 0)
        z = y + 1;
    else
        z = y + 2;
    return z;
}
```

define i32 @ssa2() nounwind {
  entry:
    %call = call i32 @f()
    %cmp = icmp slt i32 %call, 0
    br i1 %cmp, label %if.then, label %if.else
  if.then:
    %add = add nsw i32 %call, 1
    br label %if.end
  if.else:
    %add1 = add nsw i32 %call, 2
    br label %if.end
  if.end:
    %z.0 = phi i32 [%add, %if.then], [%add1, %if.else]
    ret i32 %z.0
}
mem2reg get confused easily

```c
int ssa3() {
    int z;
    return *((&z + 1 - 1);
}
```

define i32 @ssa3() nounwind {
    entry:
    %z = alloca i32, align 4
    %add.ptr = getelementptr inbounds i32* %z, i32 1
    %add.ptr1 = getelementptr inbounds i32* %add.ptr, i32 -1
    %0 = load i32* %add.ptr1, align 4
    ret i32 %0
}

gatelemtptr abstracts away offset calculation