SSA and DFAs

Simone Campanoni
simonec@eecs.northwestern.edu
SSA Outline

• SSA and why?

• Reaching definitions, constant propagation with SSA forms

• SSA in LLVM

• Generate SSA code
Def-use chains

Within your CAT: you can follow def-use chains e.g., i->getUses()

in both directions e.g., i->getDefinitions()
Def-use chains

Within your CAT: you can follow def-use chains e.g., i->getUses()

in both directions e.g., i->getDefinitions()

- An use can get data from multiple definitions depending on the control flow executed
- This is why we need to propagate data-flow values through all possible control flows
Def-use chain and DFA

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

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

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

\[ i: t \leftarrow \ldots \]
\[ \text{GEN}[i] = \{ i \} \]
\[ \text{KILL}[i] = \text{defs}(t) - \{ i \} \]

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

- Given a variable \( t \), we need to find all definitions of \( t \) in the CFG
- How can we do it in LLVM?
LLVM IR (4)

It’s a Static Single Assignment (SSA) representation

- A variable is set only by one instruction in the function body
  \%
  myVar = ...

- A static assignment can be executed more than once
  While (...){
    \%
    myVar = ...
  }
SSA and not SSA example

```c
float myF (float par1, float par2, float par3){
    return (par1 * par2) + par3; }
```

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

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

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

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

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

  No WAW, WAR data dependencies between variables!

• 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
Motivation for SSA

- Code analysis needs to represent facts at every program point

```
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: reaching definition

We iterate over instructions and if a new instruction doesn’t redefine \( x \), then, we keep propagating “\( x=3 \)”

This is needed to know whether this \( x \) can/must/cannot be equal to 3
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?
- No guarantee that def dominates use
- No guarantee about which def will be the last def before an use
Variable def-use chains in LLVM

- Iterate over users of a definition:
  ```cpp
  for (auto &user : i.users()){
    if (auto j = dyn_cast<Instruction>(&user)){
      ...
    }
  }
  ```

- Iterate over uses
  ```cpp
  for (auto &use : i.uses()){
    auto user = use.getUser();
    if (auto j = dyn_cast<Instruction>(&user)){
      ...
    }
  }
  ```

   

   i is the definition
   j is a user of i
   This fact is called “use”
Basic block def-use chains in LLVM

- Def = definition of a basic block
- User = ?
Function def-use chains in LLVM

• Def = definition of a function
• User = ?
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

```
If (b > N)
  b = c + 1
  b = d + 1

b1 = c + 1
b2 = d + 1
```

```
If (?) > N)
  b1 = c + 1
  b2 = d + 1

b3=\Phi(b1, b2)
If (b3 > N)
```

Not SSA

Still not SSA

SSA
Eliminating $\Phi$

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

```
\begin{align*}
\text{If (b3 > N)} \\
\text{b1} &= c + 1 \\
\text{b2} &= d + 1 \\
\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.
SSA Outline

• SSA and why?

• Reaching definitions, constant propagation with SSA forms

• SSA in LLVM

• Generate SSA code
Consequences of SSA

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

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

To SSA IR

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

• 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

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

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

while (changes to any \(\text{OUT}\) occur)

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

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

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

}\}

\[
i: t \gets \ldots \]
\[
\text{GEN}[i] = \{i\}
\]
\[
\text{KILL}[i] = \text{defs}(t) \setminus \{i\}
\]

\[
i: \ldots \]
\[
\text{GEN}[i] = \{\}
\]
\[
\text{KILL}[i] = \{\} \]
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) {
    \(\text{IN}[i] = \bigcup_p \text{a predecessor of } i \text{ OUT}[p]:\)
    \(\text{OUT}[i] = \text{GEN}[i]\)
  }

\(i: t \gets \ldots\)
\(\text{GEN}[i] = \{i\}\)
\(\text{KILL}[i] = \{\}\)
\(i: \ldots\)
\(\text{GEN}[i] = \{\}\)
\(\text{KILL}[i] = \{\}\)
Question answered by reaching definition analysis: does the definition “i” reach “j”?
Does it mean we can always propagate constants to variable uses?

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

• SSA and why?

• Reaching definitions, constant propagation with SSA forms

• SSA in LLVM

• Generate SSA code
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: variables

• Let’s say we have the following C code:
• The equivalent bitcode is the following:

```c
3 int main (int argc, char *argv[]){
4 int v1, v2;
5 v1 = argc;
6 if (argc > 2){
7    v2 = v1 + 1;
8    return v2;
9 }
10 return v1;
11 }
```

```llvm
define dso_local i32 @main(i32, i8**) #0 {
   %3 = icmp sgt i32 %0, 2
   br i1 %3, label %4, label %6
10
11 ; <label>:4:
12 %5 = add nsw i32 %0, 1
13 br label %7
14
15 ; <label>:6:
16 br label %7
17
18 ; <label>:7:
19 %0 = phi i32 [%5, %4 ], [%0, %6 ]
20 ret i32 %0
21 }
```

• %3, %5, and %.0 are variables. How can we access them?
  E.g., Function::getVariable(%3)
  E.g., Instruction::getVariableDefined()
• It seems variables do not exist from the LLVM API!
The variable defined by an instruction is represented by the instruction itself!
This is thanks to the SSA representation

Value * Instruction::getOperand(unsigned i)
Value * CallInst::getArgOperand(unsigned i)
SSA in LLVM: variables (3)

• The variable defined by an instruction is represented by the instruction itself
• How can we find out the type of the variable defined?
  
  Type *varType = inst->getType()
  if (varType->isIntegerTy()) ...
  if (varType->isIntegerTy(32)) ...
  if (varType->isFloatingPointTy()) ...

![Diagram showing the hierarchy of type nodes: Type, IntegerType, PointerType, ...]
SSA Outline

• SSA and why?

• Reaching definitions, constant propagation with SSA forms

• SSA in LLVM

• Generate SSA code
Modify SSA code while preserving its SSA property

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

• How should we do it?
  • 2 solutions: variable renaming and variable spilling

  \[
  \begin{align*}
  \%v &= \ldots \\
  \%v1 &= \%v + 1 \\
  \%y &= \%v1 \\
  \%z &= \%v1
  \end{align*}
  \]

  Step 1: rename the new definition (%v -> %v1)

  \[
  \begin{align*}
  \%v &= \ldots \\
  \%v1 &= \%v + 1 \\
  \%y &= \%v1 \\
  \%z &= \%v1
  \end{align*}
  \]

  Step 2: rename all uses
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?

2 solutions: variable renaming and variable spilling.

Step 0: create a builder
IRBuilder<> b(I)

Step 1: create a new definition
auto newI = cast<Instruction>(b.CreateAdd(I, const1))

Step 2: rename all uses
I->replaceAllUsesWith(newI)
Modify SSA code while preserving its SSA property

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

• How should we do it?
  • 2 solutions: variable renaming and variable spilling

%pv = alloca(...)  
%v0 = load %pv  
%v1 = %v0 + 1  
store %v1, %pv  
%y = load %pv

%v = ...  
%y = %v  
%z = %v

%v = ...  
%v = %v + 1  
%y = %v  
%z = %v

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

Step 1: allocate a new variable on the stack
Step 2: use loads/stores to access it
Step 3: convert stack accesses to SSA variable accesses
SSA in LLVM: changing variable values

• Step 0: create a builder
l=f->begin()->getFirstNonPHI()
IRBuilder<> b(l)

• Step 1: allocate a new variable on the stack
auto newV = cast<Instruction>(b.createAlloca(...))

• Step 2: use loads/stores to access it
...

• Step 3: convert stack accesses to SSA variable accesses
  • Exploit already existing passes to reduce inefficiencies (mem2reg)
  • mem2reg maps memory locations to registers when possible

```sh
eopt -mem2reg mybitcode.bc -o mybitcode.bc
```
The mem2reg LLVM pass

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

```assembly
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**

```assembly
define i32 @ssa1() nounwind {
  entry:
  %z = alloca i32, align 4
  %call = call i32 @f()
  %add = add nsw i32 %call, 1
  store i32 %add, i32* %z, align 4
  %0 = load i32* %z, align 4
  ret i32 %0
}
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;
}
```

```assembly
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
}

getelementptr abstracts away offset calculation