Code analysis and transformation

SSA

Simone Campanoni
simone.campanoni@northwestern.edu
Outline

• SSA and why?

• SSA in LLVM

• Generate SSA code
LLVM IR (4)

• It’s a Static Single Assignment (SSA) representation

• First constraint of an SSA representation: A variable is set only by one instruction in the whole function body
LLVM IR: SSA and not SSA example

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

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

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

NOT SSA

SSA
A direct consequence of using a SSA form

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

```
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!
Static Single Assignment (SSA) Form

• A variable is set only by one instruction in the function body
  \%myVar = ...
  A static assignment can be executed more than once
  While (...){
    \%myVar = ...  
  }

• The definition must be guaranteed to always execute before all of its uses

• Code analyses and transformations that assume SSA are (typically) faster, they use less memory, and they include less code (compared to their non-SSA versions)
Compilers using SSA

- LLVM (IR)
- Swift (SIL)
- Recent GCC (GIMPLE IR)
- Mono
- Portable.NET
- Mozilla Firefox SpiderMonkey JavaScript engine (IR)
- Chromium V8 JavaScript engine (IR)
- PyPy
- Android’s new optimizing compiler
- PHP

- Go
- WebKit
- Erlang
- LuaJit
- IBM open source JVM
- ...
Consequences of SSA

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

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

To SSA IR

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

No WAW, WAR data dependencies between variables!

• Def—use chains are greatly simplified
  • We are going to see def-use chains for a non-SSA IR
  • Then we see how def-use chains look like for an SSA IR
Def-use chains in a non-SSA IR

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

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

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

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

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

Given a variable \( t \), we need to find all definitions of \( t \) in the CFG

\[
i: \ldots \\
\text{GEN}[i] = \{\} \\
\text{KILL}[i] = \{\}
\]
Def-use chains in a non-SSA IR

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

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

Which definition was executed for a given use?
We need to run a data-flow analysis to answer it
Def-use chains in an SSA IR

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

Which definition was executed for a given use? There is only one definition for a given use.
Def-use chains in an SSA IR

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

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

Which definition was executed for a given use?
There is only one definition for a given use and it is guaranteed to be executed before all of its uses
Consequences of SSA

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

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

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

No WAW, WAR data dependencies between variables!

• Use—def chain are greatly simplified
• Data-flow analysis are simplified (... in a few 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

This is a dense representation of data-flow values

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

Is it constant?

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
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

$$b = c + 1 \quad b = d + 1$$

Not SSA

If $(b > N)$

$$b_1 = c + 1 \quad b_2 = d + 1$$

Still not SSA

If (? > N)

$$b_1 = c + 1 \quad b_2 = d + 1$$

SSA

$b_3 = \Phi(b_1, b_2)$

If $(b_3 > N)$
Eliminating $\Phi$ in the back-end

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

$$b_1 = c + 1$$
$$b_2 = d + 1$$

$$b_3 = \Phi(b_1, b_2)$$
If ($b_3 > N$)

$$b_1 = c + 1$$
$$b_2 = d + 1$$
$$b_3 = b_1$$
If ($b_3 > N$)

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?)
- Eliminate $\Phi$s that have no uses
- Subsequent register allocation will map the variables onto the actual set of machine register
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)
```

To SSA IR

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

• 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

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

i: $t$ <- ...
  GEN[$i$] = {$i$}
  KILL[$i$] = defs($t$) ─ {$i$}

i: ...
  GEN[$i$] = {}
  KILL[$i$] = {}
Def-use chain with SSA

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

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

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

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

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

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

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

\( i: \ldots \)
\( \text{GEN}[i] = \{ \} \)
\( \text{KILL}[i] = \{ \} \)
Code example

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”? 

How should we design constant propagation for SSA IRs?
Outline

• SSA and why?

• SSA in LLVM

• Generate SSA code
SSA in LLVM

• The IR is assumed to be always in SSA
  • Checked at boundaries of passes
  • No time wasted converting automatically IR to its SSA form
  • CAT designed with this constraint in mind

• Φ instructions only at the top of a basic block
SSA in LLVM: Φ instructions

When the predecessor just executed is %4
store the constant 1 to %.0
SSA in LLVM: Φ instructions

When the predecessor just executed is %5
store %6 to %0
SSA in LLVM: $\Phi$ (PHI) instructions

- A PHI instruction can have many [predecessor, value] pairs as inputs.
- A PHI instruction must have one pair per predecessor.
- A PHI instruction must have at least one pair.
- A PHI instruction is a definition.
  - Hence, it must dominate all of its uses.
  - PHI uses are defined to happen on the incoming edge, not at the instruction.

```asm
%v2 = phi [%L0, %v0], [%L1, %v1]
...
%v3 = add %v2, 1
```

This use must be dominated by.
SSA in LLVM: Φ (PHI) instructions

• PHI must dominate all of its uses
• PHI uses are defined to happen on the incoming edge, not at the instruction

```
L0:
  %v2 = phi [...], [%L1, %v1]
  ...

L1:
  %v3 = ...

%v1 = phi [%L1, %v3], [%L0, %v2]
  ...
```
SSA in LLVM: Φ (PHI) instructions

L0:
  %v2 = phi [...], [%L1, %v1]
  ...

L1:
  %v3 = ...

%v1 = phi [%L1, %v3], [%L0, %v2]
  ...

%v1 = phi [...], [%L1, %v1]
SSA in LLVM: Variable def-use chains

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

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

Why do we need Use?

i is the definition of %v
j is a user of i
This fact is called “use”
SSA in LLVM: Variable def-use chains

Use differentiates between **and** **, User does not**

- Replace only a specific operand:
  - From: `call @myF (%v0, %v1, %v0)`
  - To: `call @myF (%w0, %v1, %v0)`

- If `i` is the instruction that defines `%v0`
  - `i` has different uses in the call above
  - An `Use` holds information about it

- Iterate over uses

```
for (auto &use : i.uses()){
  User *user = use.getUser();
  if (auto j = dyn_cast<Instruction>(user)){
    ...
  }
}
```

- **i** is the definition of `%v`
- **j** is a user of `i`

This fact is called “use”
Def-use chains

• So far we saw def-use chains for variables

• But LLVM has def-use chains for other compiler concepts
SSA in LLVM: Basic block def-use chains

• Def = definition of a basic block
• User = ?

```cpp
bool runOnFunction (Function &F){
  for (auto &BB : F){
    for (auto &user : BB.users()){
      ...
    }
  }
}
```
SSA in LLVM: Function def-use chains

• Def = definition of a function
• User = ?

```cpp
bool runOnFunction (Function &F){
  for (auto &user : F.users()){
    ...
  }
}
```
SSA in LLVM: variables

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

```c
#define dso_local i32 @main(i32, i8**) #0 {
  %3 = icmp sgt i32 %0, 2
  br i1 %3, label %4, label %6
  ; <label>:4: ; preds = %2
  %5 = add nsw i32 %0, 1
  br label %7
  ; <label>:6: ; preds = %2
  br label %7
  ; <label>:7: ; preds = %6, %4
  %.0 = phi i32 [ %.5, %4 ], [ %.0, %6 ]
  ret i32 %.0
}
```

- %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!
Variables do not exist
SSA in LLVM: variables (2)

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()) ... 
  ```
LLVM class hierarchies we saw so far
LLVM class hierarchies we saw so far

Value
  - Argument
  - User
    - Instruction
      - BinaryOperator
      - ReturnInst
      - Constant
    - ...
Outline

• SSA and why?

• 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

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

Step 1: rename the new definition (\%v -> \%v1)
Step 2: rename all uses

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

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

%pv = alloca(...)%
v = ...
y = %v
z = %v
%y = %v + 1
%z = %v

%pv0 = load %pv
%v1 = %v0 + 1
store %v1, %pv
%y = load %pv
Modify SSA code while preserving its SSA property

• Step 0: create a builder

```cpp
auto I = f->begin()->getFirstNonPHI();
IRBuilder<> b(I)
```

• Step 1: allocate a new variable on the stack

```cpp
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 variables when possible

```
opt --mem2reg mybitcode.bc -o mybitcode.bc
```
The mem2reg LLVM pass

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

Stack allocation in the entry block

Only used by loads and stores
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;
}
```

```c
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);
}
```

```c
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

*Be careful at generating accesses to* alloca *objects*

*if you want* mem2reg *to automatically map them to SSA variables*
Always have faith in your ability

Success will come your way eventually

Best of luck!