LLVM-CANON: SHOOTING FOR CLEAR DIFFS

Presenting: Michal Paszkowski
Research: Michal Paszkowski, Radoslaw Drabinski
Special thanks to: Julia Koval
It is all about the semantic differences!

Desired

• Semantics

Distracting

• Instruction order
• Value names
• Basic Block names
• Redundant code
How could we automate that?

We needed a simple tool that would make comparing semantic differences between two modules easier.

The tool should...

• "Reduce" non-semantic differences
• Process modules independently
• Leverage existing diff tools
How could we automate that?

Therefore, the tool should transform a module into a canonical form.

How will that “canonical form” help us?

• Two semantically identical canonicalized modules should show no differences when diffed.

And more importantly…

• When the modules are not identical the semantic differences should stand out.
How do we arrive at this canonical form?

Let’s start with instruction ordering…

Assumptions for comparing an identical module after two different transformations:

• Side-Effects should be roughly the same
• Control-Flow Graphs should be similar
Instruction reordering

- *def-use* distance reduction

```assembly
%1 = ...
%2 = ...
%T1 = call float @inputV.f32(i32 15, i32 2)
%A = fsub float %T1, %T1
%B = fmul float %A, %T1
%C = fsub float %T1, %B
%X = fadd float %C, %A
store float %C, float addrspace(479623)* %1
store float %X, float addrspace(283111)* %2
ret void
```
Instruction reordering

• *def-use* distance reduction

1. Collect instructions with *side-effects* and "ret" instructions

\[
%1 = \ldots \\
%2 = \ldots \\
\%T1 = \text{call float} @\text{inputV.f32}(\text{i32 } 15, \text{i32 } 2) \\
\%A = \text{fsub float} \%T1, \%T1 \\
\%B = \text{fmul float} \%A, \%T1 \\
\%C = \text{fsub float} \%T1, \%B \\
\%X = \text{fadd float} \%C, \%A \\
\text{store float} \%C, \text{float addrspace}(479623)* \%1 \\
\text{store float} \%X, \text{float addrspace}(283111)* \%2 \\
\text{ret void}
\]
Instruction reordering

- *def-use* distance reduction

  1. Collect instructions with side-effects and "ret" instructions

  2. Walk the instructions with side-effects (top-down) and on each instruction their operands (left-right)

```plaintext
%1 = ...
%2 = ...
%T1 = call float @inputV.f32(i32 15, i32 2)
%A = fsub float %T1, %T1
%B = fmul float %A, %T1
%C = fsub float %T1, %B
%X = fadd float %C, %A

store float %C, float addrspace(479623)* %1
store float %X, float addrspace(283111)* %2
ret void
```
Instruction reordering

- **def-use distance reduction**
  1. Collect instructions with side-effects and "ret" instructions
  2. Walk the instructions with side-effects (top-down) and on each instruction their operands (left-right)
  3. For each operand, bring their definition as close as possible to the using instruction

```assembly
%1 = ...
%2 = ...
%T1 = call float @inputV.f32(i32 15, i32 2)
%A = fsub float %T1, %T1
%B = fmul float %A, %T1
%C = fsub float %T1, %B
%X = fadd float %C, %A
store float %C, float addrspace(479623)* %1
store float %X, float addrspace(283111)* %2
ret void
```
Step-by-step reorder walkthrough

%T1 = call float @inputV.f32(i32 15, i32 2)

%A = fsub float %T1, %T1

%B = fmul float %A, %T1

%C = fsub float %T1, %B
%X = fadd float %C, %A

store float %C, float addrspace(479623)* %1

store float %X, float addrspace(283111)* %2

ret void
Step-by-step reorder walkthrough

%T1 = call float @inputV.f32(i32 15, i32 2)

%A = fsub float %T1, %T1

%B = fmul float %A, %T1

%C = fsub float %T1, %B
%X = fadd float %C, %A

store float %C, float addrspace(479623)* %1
store float %X, float addrspace(283111)* %2

ret void

Select the 1st side-effecting instruction.
Don't move it, otherwise semantics may not be preserved!
Step-by-step reorder walkthrough

%T1 = call float @inputV.f32(i32 15, i32 2)

%A = fsub float %T1, %T1

%B = fmul float %A, %T1

%C = fsub float %T1, %B
%X = fadd float %C, %A

store float %C, float addrspace(479623)* %1
store float %X, float addrspace(283111)* %2

ret void

Take the 1st operand of the side-effecting instruction.

Canonicalized instructions
Step-by-step reorder walkthrough

%T1 = call float @inputV.f32(i32 15, i32 2)

%A = fsub float %T1, %T1

%B = fmul float %A, %T1

%C = fsub float %T1, %B

%X = fadd float %C, %A

%C = fsub float %T1, %B

store float %C, float addrspace(479623)* %1

store float %X, float addrspace(283111)* %2

ret void
Step-by-step reorder walkthrough

\%T1 = call float @inputV.f32(i32 15, i32 2)

\%A = fsub float \%T1, \%T1

\%B = fmul float \%A, \%T1

\%X = fadd float \%C, \%A

\%C = fsub float \%T1, \%B
store float \%C, float addrspace(479623)* \%1

store float \%X, float addrspace(283111)* \%2

ret void

Select the 1st operand of the moved instruction.

 Canonicalized instructions
Step-by-step reorder walkthrough

%T1 = call float @inputV.f32(i32 15, i32 2)

%A = fsub float %T1, %T1

%B = fmul float %A, %T1

%X = fadd float %C, %A
%T1 = call float @inputV.f32(i32 15, i32 2)

%C = fsub float %T1, %B
store float %C, float addrspace(479623)* %1

store float %X, float addrspace(283111)* %2

ret void

Move it closer to the user.
Def-Use sequence may be temporarily broken.

Canonicalized instructions
Step-by-step reorder walkthrough

\[
\%A = \text{fsub float} \ \%T1, \ %T1
\]

\[
\%B = \text{fmul float} \ %A, \ %T1
\]

\[
\%X = \text{fadd float} \ %C, \ %A
\]
\[
\%T1 = \text{call float} \ \text{@inputV.f32(i32 15, i32 2)}
\]

\[
\%C = \text{fsub float} \ %T1, \ %B
\]
\[
\text{store float} \ %C, \ \text{float addrspace}(479623)* \ %1
\]
\[
\text{store float} \ %X, \ \text{float addrspace}(283111)* \ %2
\]

\text{ret void}

Select the 2nd operand of the previously moved instruction.

Canonicalized instructions
Step-by-step reorder walkthrough

%A = fsub float %T1, %T1

%B = fmul float %A, %T1

%X = fadd float %C, %A

%T1 = call float @inputV.f32(i32 15, i32 2)

%B = fmul float %A, %T1

%C = fsub float %T1, %B

store float %C, float addrspace(479623)* %1

store float %X, float addrspace(283111)* %2

ret void

Move it closer to the user.
Def-Use sequence is being repaired.

Canonicalized instructions
Step-by-step reorder walkthrough

\[%A = \text{fsub float} \ %T1, \ %T1\%

\[%X = \text{fadd float} \ %C, \ %A\%

\[%T1 = \text{call float} \ @\text{inputV.f32(i32 15, i32 2)}\%

\[%B = \text{fmul float} \ %A, \ %T1\%

\[%C = \text{fsub float} \ %T1, \ %B\%

\text{store float} \ %C, \ \text{float addrspace(479623)}* \ %1

\text{store float} \ %X, \ \text{float addrspace(283111)}* \ %2

\text{ret void}\]
Step-by-step reorder walkthrough

%A = fsub float %T1, %T1

%X = fadd float %C, %A
%T1 = call float @inputV.f32(i32 15, i32 2)
%A = fsub float %T1, %T1
%B = fmul float %A, %T1
%C = fsub float %T1, %B
store float %C, float addrspace(479623)* %1

store float %X, float addrspace(283111)* %2

ret void
Step-by-step reorder walkthrough

Select the 1st operand of the moved instruction. Don't move it! %T1 has been already moved!

Select the 2nd operand of the moved instruction. Don't move it! %T1 has been already moved!

%X = fadd float %C, %A
%T1 = call float @inputV.f32(i32 15, i32 2)
%A = fsub float %T1, %T1
%B = fmul float %A, %T1
%C = fsub float %T1, %B
store float %C, float addrspace(479623)* %1

store float %X, float addrspace(283111)* %2

ret void
Step-by-step reorder walkthrough

%X = fadd float %C, %A
%T1 = call float @inputV.f32(i32 15, i32 2)
%A = fsub float %T1, %T1
%B = fmul float %A, %T1
%C = fsub float %T1, %B
store float %C, float addrspace(479623)* %1

store float %X, float addrspace(283111)* %2

ret void

Select the 2nd operand of the previously moved instruction. Don’t move it, %T1 has been moved before.

Canonicalized instructions
Step-by-step reorder walkthrough

We repeat the process for all operands in all side-effecting instructions.

Canonicalized instructions
How do we arrive at this canonical form?

- Desired
  - Semantics

- Distracting
  - Instruction order ✓
  - Value names
  - Basic Block names
  - Redundant code
Naming instructions: Linear

- Numbers all instructions top-down after reordering $v_n$
- We were hoping that maybe the reordering mechanism could be used as a ‘seed’ for instruction naming

```asm
%T1 = @inputV.f32(i32 15, i32 2)
%A = fsub float %T1, %T1
%B = fmul float %A, %T1
%C = fsub float %T1, %T1
%X = fadd float %C, %A
store float %C, float addrspace(479623)* %1
store float %X, float addrspace(283111)* %2
ret void
```

```asm
%v0 = @inputV.f32(i32 15, i32 2)
%v1 = fsub float %v0, %v0
%v2 = fmul float %v1, %v0
%v3 = fsub float %v0, %v0
%v4 = fadd float %v3, %v1
store float %v3, float addrspace(479623)* %a3
store float %v4, float addrspace(283111)* %a4
ret void
```
Naming instructions: Linear

- Numbers all instructions top-down after reordering $vn$
- We were hoping that maybe the reordering mechanism could be used as a ‘seed’ for instruction naming

```assembly
%T1 = @inputV.f32(i32 15, i32 2)
%A = fsub float %T1, %T1
%B = fmul float %A, %T1
%C = fsub float %T1, %T1
%X = fadd float %C, %A
store float %C, float addrspace(479623)* %1
store float %X, float addrspace(283111)* %2
ret void

%v0 = @inputV.f32(i32 15, i32 2)
%v1 = fsub float %v0, %v0
%v2 = fmul float %v1, %v0
%v3 = fsub float %v0, %v0
%v4 = fadd float %v3, %v1
store float %v3, float addrspace(479623)* %a3
store float %v4, float addrspace(283111)* %a4
ret void
```
Two types of instructions:

1. Initial instructions
   - Instructions with only immediate operands
   - Numbered according to positions of outputs using that instruction after sorting

2. Regular instructions
   - “Graph naming”: Differences in defs are reflected in uses
Two types of instructions:

1. Initial instructions
   - Instructions with only immediate operands
   - Numbered according to positions of outputs using that instruction after sorting

2. Regular instructions
   - “Graph naming”: Differences in defs are reflected in uses
Naming instructions: “Graph naming”

Two types of instructions:

1. Initial instructions
   • Instructions with only immediate operands
   • Numbered according to positions of outputs using that instruction after sorting

2. Regular instructions
   • “Graph naming”: Differences in defs are reflected in uses

Instructions with different opcodes but same users got the same names.

Extremely long names! Differences in defs should be reflected only in outputs.
Naming instructions: Current version

1. Initial instructions (those with only immediate operands)

\[
\text{%"v1\{12345\}\text{Foo}(\{2, 5\})" = \ldots}
\]

- Hash calculated considering instruction’s opcode and the “output footprint”
- Called function name only included in case of a CallInst
- Immediate operands list (sorted in case of commutative instructions)
Naming instructions: Current version

2. Regular instructions

```
%"op12345_Foo(op54321, ...)"
```

- Hash calculated considering instruction’s and its operands’ opcodes
- Called function name only included in case of a CallInst
- Short operand names
Naming instructions: Current version

3. Output instructions (instructions with side-effects and their relative operands)

`%"op 12345 Foo ( op54321(...), ... )"`

- Same as regular instructions, but…
- recursively generated long operand list is kept, so…
- by just looking at an output we see what impacts its semantics in the diff.
Naming instructions: Current version

"vl20713gfx_input(0, 2)" = % Initial instructions
"vl15160gfx_input(1, 2)" = %
"op46166(vl15160)" = % Regular instruction
"op11867(vl20713gfx_input(0, 2), op46166(vl15160...))" = % Output instruction
null
Other little things…

• Naming basic blocks
• Numbering function arguments
• Sorting values in PHI nodes
Why not integrate with llvm-diff?

- We wanted to use this tool to also spot differences in just one file.
- We wanted to leverage existing diff tools.

However, llvm-canon could be used as a prepass before using llvm-diff:
llvm-canon vs llvm-diff

define double @foo(double %a0, double %a1) {
  entry:
    %a = fmul double %a0, %a1
    %b = fmul double %a0, 2.000000e+00
    %c = fmul double %a, 6.000000e+00
    %d = fmul double %b, 6.000000e+00
  ret double %d
}

define double @foo(double %a0, double %a1) {
  entry:
    %a = fmul double %a0, %a1
    %c = fmul double 6.000000e+00, %a
    %b = fmul double %a0, 2.000000e+00
    %d = fmul double 6.000000e+00, %b
  ret double %d
}
Special thanks

Thanks to Puyan Lotfi for his talk on MIR-Canon during 2018 EuroLLVM and a head start on canonicalization techniques.

Special thanks to Radoslaw Drabinski and Julia Koval.

I would like to also thank the LLVM community for excellent code review and coming to this talk.