Code: analysis, bugs, and security
supported by Bitdefender

Compiler basics. LLVM compiler infrastructure

Marius Minea

marius@cs.upt.ro

1 November 2017

Compilers: generate executable code

```
#include <disclaimer.h>
```

Not a compilers course

Basics for understanding / analyzing / reverse engineering code

Compilation steps

Preprocessing

Lexical analysis (scanner)

Syntactic Analysis (parser)

Semantic Analysis
  e.g. type checking

Intermediate Representation Generation

IR Optimization

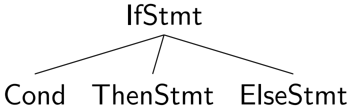
Code generation

Abstract and concrete syntax

Concrete syntax
includes representation details (keywords, punctuation)

Abstract syntax
represents conceptual structure (no keywords, but various node types, attributes, etc.)
implicit language elements (conversions,...) may appear explicitly

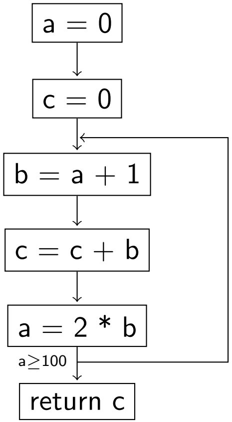
$\Rightarrow$ abstract syntax tree



AST is starting point for subsequent processing $\longrightarrow$ CFG

Control flow graph (CFG)

```
int a = 0, b, c = 0;
do {
  b = a + 1;
  c = c + b;
  a = 2 * b;
} while (a < 100);
return c;
```



nodes are *basic blocks*: straight-line code segments with single entry and exit

fundamental data structure for analysis and code generation

Dataflow analyses

obtains information about possible sets of *values* computed at various points in the program

value need not be numeric: any useful information

Reaching Definitions uninitialized variables ?

Live Variables "value assigned but never used"

what/how many registers needed ?

Available Expressions Very Busy Expressions

for code motion / optimization

Intermediate formats

Higher-level IR
preserves object structure
for dataflow and other analyses

Lower-level IR
for code generation and optimization

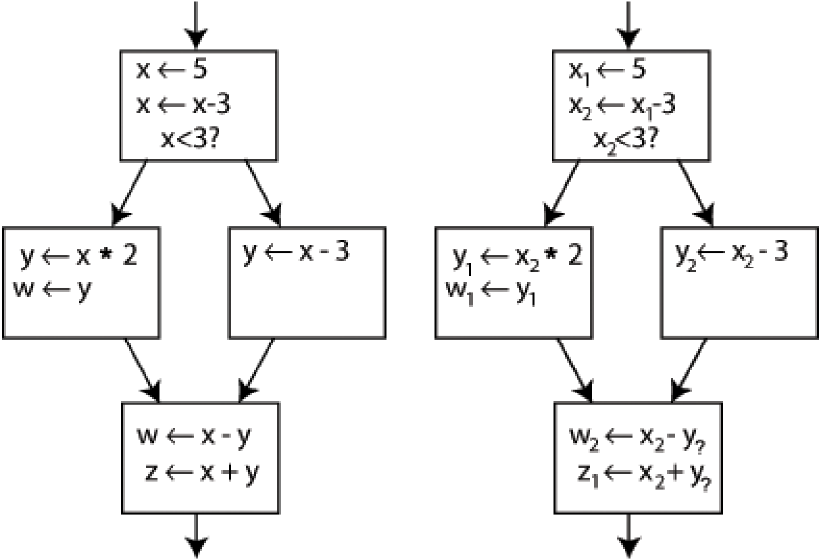
Static single assignment (SSA) form

IR in which every variable is assigned exactly once
so new auxiliary variables are introduced

Developed by IBM researchers in 1980s

significantly simplifies dataflow analysis

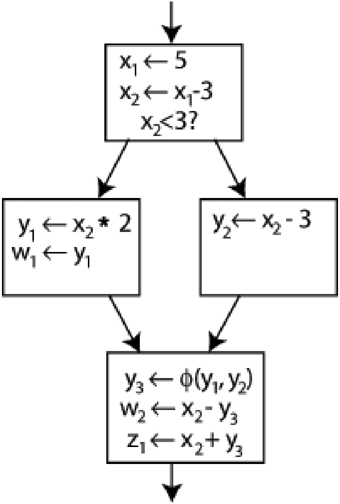
SSA by example



https://en.wikipedia.org/wiki/Static_single_assignment_form

SSA by example (cont'd)

For joins of *then* and *else* branches, introduce special Φ-node, depending on one of two variables
(like conditional expression)



https://en.wikipedia.org/wiki/Static_single_assignment_form

LLVM Intermediate Representation

LLVM: collection of compiler & toolchain technologies

- started by Chris Lattner (later: Apple) & Vikram Adve at UIUC
- now open-source, very widely used

Clang: compiler in the LLVM framework, alternative to gcc

- many analyses & security add-ons done for `clang`

Can write our own:

- analysis passes (do not change code)
- transformation passes (keep or *change* behavior)

Sample LLVM intermediate code

```
define i32 @delta(i32 %a, i32 %b, i32 %c) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 %a, i32* %1, align 4
  store i32 %b, i32* %2, align 4
  store i32 %c, i32* %3, align 4
  %4 = load i32* %2, align 4
  %5 = load i32* %2, align 4
  %6 = mul nsw i32 %4, %5
  %7 = load i32* %1, align 4
  %8 = mul nsw i32 4, %7
  %9 = load i32* %3, align 4
  %10 = mul nsw i32 %8, %9
  %11 = sub nsw i32 %6, %10
  ret i32 %11
  }
```

Same code optimized

```
define i32 @delta(i32 %a, i32 %b, i32 %c) #0 {
  %1 = mul nsw i32 %b, %b
  %2 = shl i32 %a, 2
  %3 = mul nsw i32 %2, %c
  %4 = sub nsw i32 %1, %3
  ret i32 %4
}
```

Notice: 3-address code, in SSA form

Same code optimized

```
define i32 @delta(i32 %a, i32 %b, i32 %c) #0 {
  %1 = mul nsw i32 %b, %b
  %2 = shl i32 %a, 2
  %3 = mul nsw i32 %2, %c
  %4 = sub nsw i32 %1, %3
  ret i32 %4
}
```

Notice: 3-address code, in SSA form

C source:

```
int delta(int a, int b, int c) {
  return b * b - 4 * a * c;
}
```

Register allocation

first: live variable analysis
    determines when a value no longer needed

Common technique: *graph coloring*

one node for each temp variable
connect two temps if live at same time
    ⇒ cannot be in same register find minimum coloring, no edge with same node colors

If insufficient ⇒ register *spilling*
    must be save in memory / on stack

Compiler optimizations

Many different types, from early to late in code generation

Local optimizations – within basic blocks
  peephole optimizations – a few instructions, assembly level

Global optimizations – within function body

Interprocedural optimizations – expensive analysis

Common optimizations

Constant folding
precompute value of constant expression

Dead Code Elimination
needs live variable analysis

Algebraic Simplification
`pow(x, 2)` $\rightarrow$ `x * x`
`x * 4` $\rightarrow$ `x << 2`

Common Subexpression Elimination
compute once into temp, use several times

Loop optimizations

Strength reduction

replace expensive with simpler operations (esp. in loops)

need to know:

loop *invariants*

loop *induction variables*

```
for (int i=0; i<n; ++i)
  a[i] = k * i;
```

```
for (int i=0, s=0; i<n; ++i) {
  a[i] = s;
  s += k;
}
```

Scalar evolution

for loops with regular computations (e.g. polynomials)

can compute loop counts, summarize loops, etc.

Loop unrolling

Branch instructions are expensive

⇒ copy loop body several times, reducing loop count

count may be statically known or not

Classic example: Duff's device

```
void send(int *to, int *from, unsigned count)
{
    register int n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
    case 5:      *to = *from++;
    case 4:      *to = *from++;
    case 3:      *to = *from++;
    case 2:      *to = *from++;
    case 1:      *to = *from++;
            } while (--n > 0);
    }
}
```