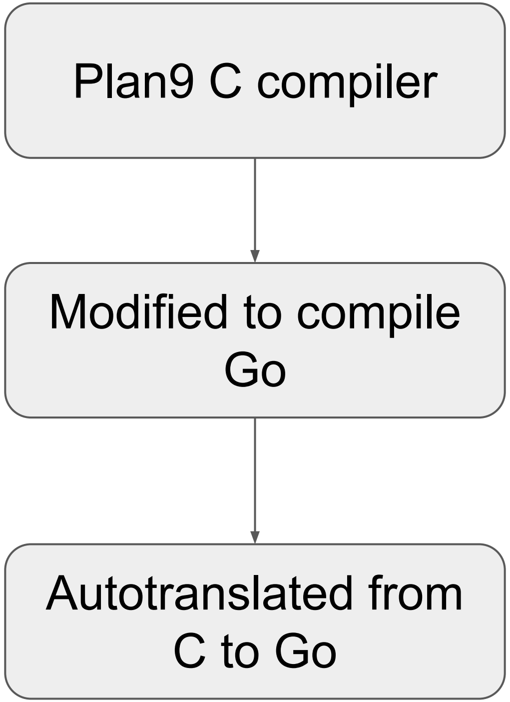
Generating better machine code with SSA

Keith Randall
@GopherCon, 2017/07/13

Generating better machine code with SSA

SSA is a technique used by most modern compilers (gcc, llvm, …) to optimize generated machine code.

The Go 1.5 compiler



Go 1.5

```
MOVQ    AX, BX
SHLQ    $0x3, BX
MOVQ    BX, 0x10(SP)
CALL    runtime.memmove(SB)
```

Go 1.5

```
MOVQ    AX, BX
SHLQ    $0x3, BX
MOVQ    BX, 0x10(SP)
CALL    runtime.memmove(SB)
```

Why not just:

```
SHLQ    $0x3, AX
MOVQ    AX, 0x10(SP)
CALL    runtime.memmove(SB)
```

Go 1.5

```
IMULQ    $0x10, R8, R8
```

Why not just:

```
SHLQ     $0x4, R8
```

Go 1.5

```
MOVQ    R8, 0x20(CX)
MOVQ    0x20(CX), R9
```

Why not just:

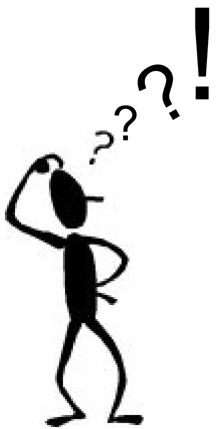
```
MOVQ    R8, 0x20(CX)
MOVQ    R8, R9
```

Go 1.5

```
LEAQ        0x10(SP), BX
MOVQ        BX, SI
```

Why not just:

```
LEAQ        0x10(SP), SI
```

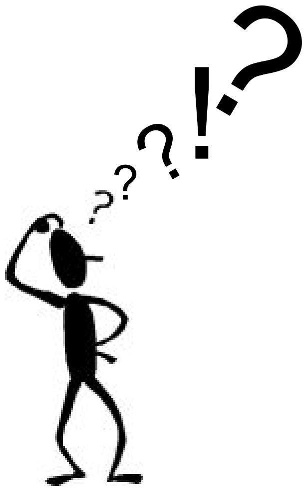


Go 1.5

```
ANDL    R8, BX
CMPL    $0x0, BX
```

Why not just:

```
ANDL    R8, BX
```



Go 1.5

```
MOVQ    AX, CX
MOVQ    CX, R9
```

Why not just:

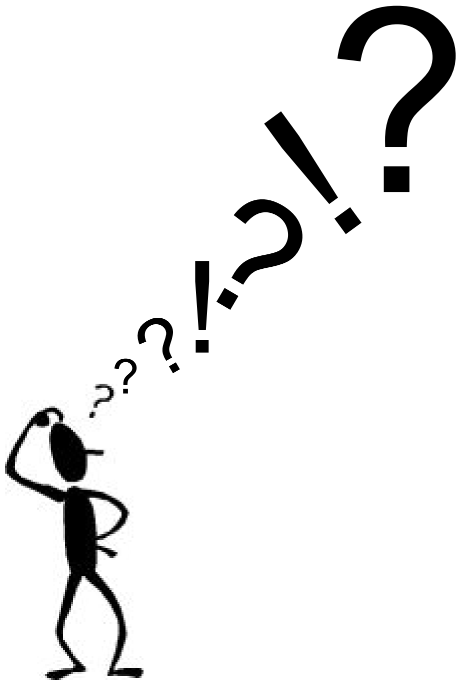
```
MOVQ    AX, R9
```

Go 1.5

```
XORL    BP, BP
CMPQ    BP, AX
JNE     ...
```

Why not just:

```
TESTQ   AX, AX
JNE     ...
```



“*I think it would be fairly easy to make the generated programs 20% smaller and 10% faster.*”

-me, Feb 2015

“*I’d like to convert from the current syntax-tree-based IR to a more modern SSA-based IR. With an SSA IR we can implement a lot of optimizations that are difficult to do in the current compiler.*”

-me, Feb 2015

"I'd like to convert from the current syntax-tree-based IR to a more modern SSA-based IR. With an SSA IR we can implement a lot of optimizations that are difficult to do in the current compiler."

-me, Feb 2015

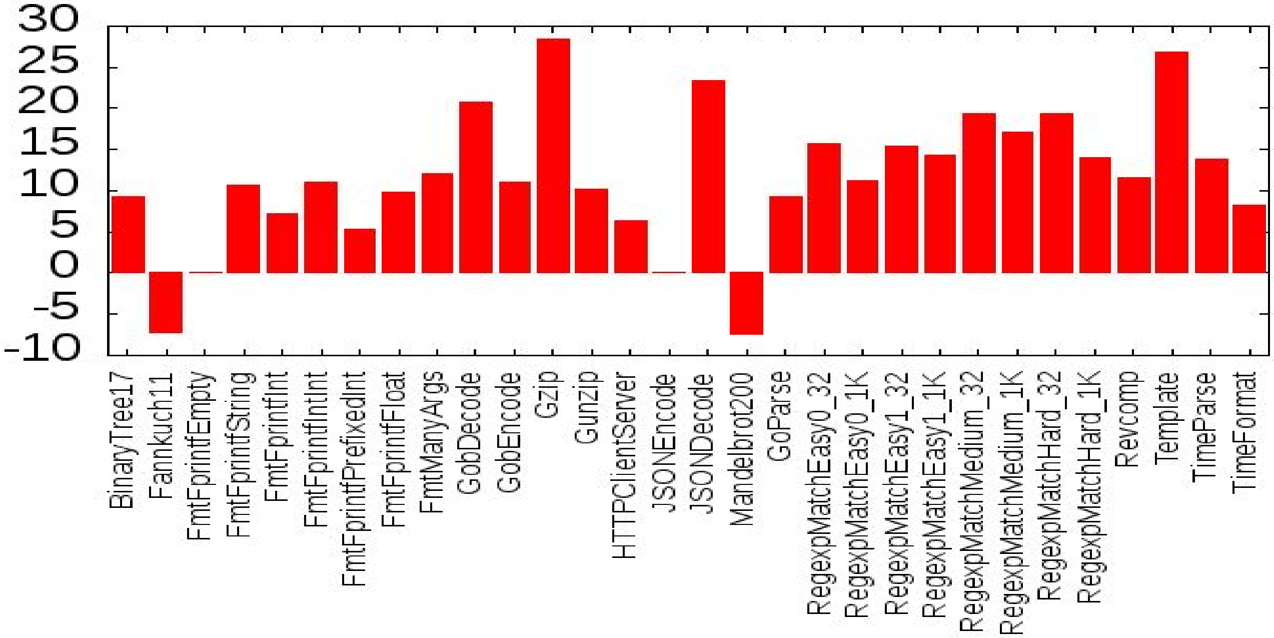
I'll explain what all of these mean.

Timeline

2015/02/10: SSA proposal mailed to golang-dev
2015/03/01: created dev.ssa branch
2016/03/01: merged dev.ssa branch into master
2016/08/15: Go 1.7 released, containing SSA for amd64
2017/02/16: Go 1.8 released, containing SSA for all other archs
(386, amd64p32, arm, arm64, mips, mips64, ppc64, s390x)

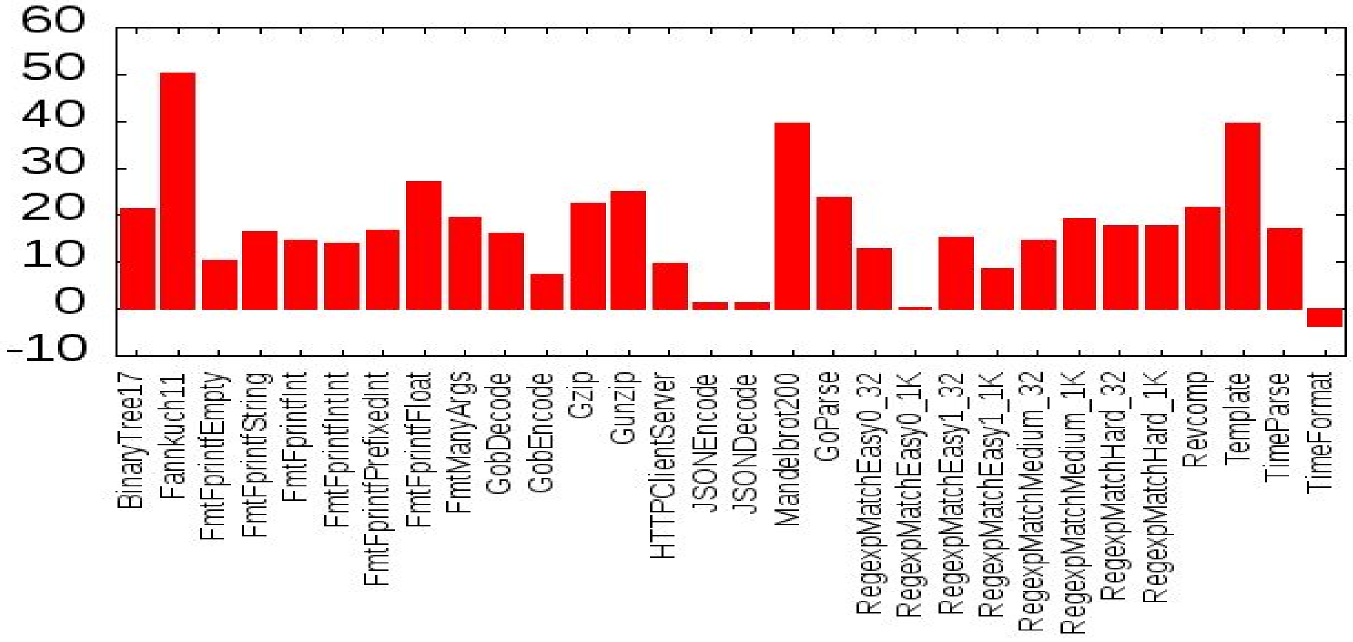
amd64 - launched in Go 1.7

12% faster on Go 1 benchmarks
13% smaller code segment



arm - launched in Go 1.8

20% faster on Go 1 benchmarks
18% smaller code segment



Community reports

- Big data workload - 15% improvement
- Convex hull - 14-24% improvement (from 1.5)
- Hash functions - 39% improvement
- Audio Processing (arm) - 48% improvement

Compiler speed

Is the new compiler faster or slower? Keep in mind:

1. The compiler has more work to do, but...
2. The compiler is compiled with the new compiler!

The amd64 compiler is 10% slower.

The extra work required by the SSA passes isn’t fully eliminated by the increase in the compiler speed.

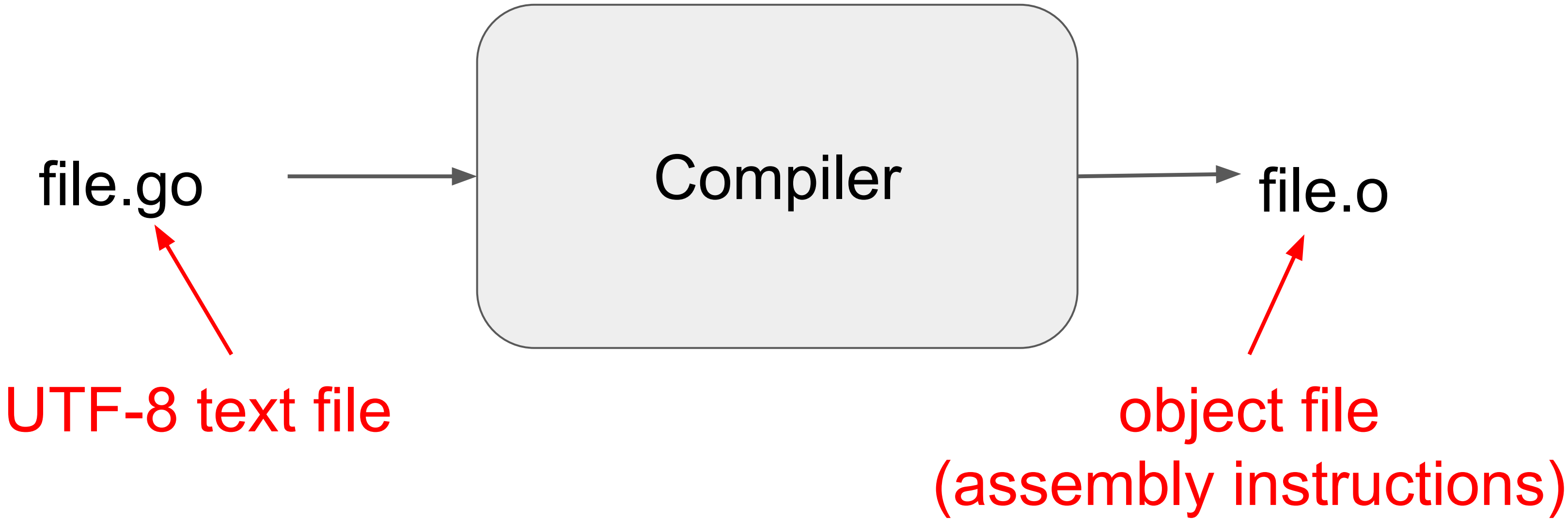
The arm compiler is 10% faster!

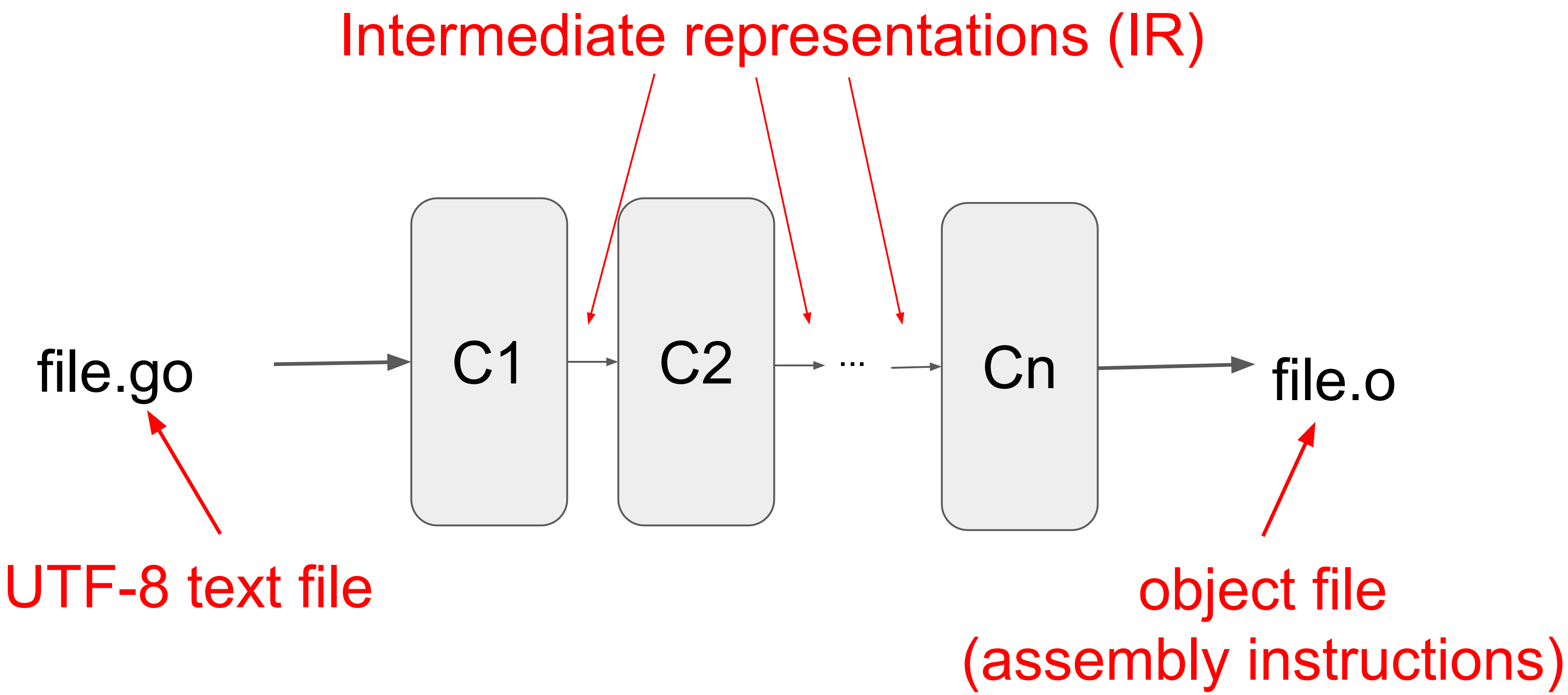
For arm, the second effect is larger than the first.

1. What is SSA?
2. Why is it useful?
3. How does Go use it?


file.go
Compiler
file.o

file.go
Compiler
file.o
UTF-8 text file


file.go
Compiler
file.o
UTF-8 text file
object file
(assembly instructions)

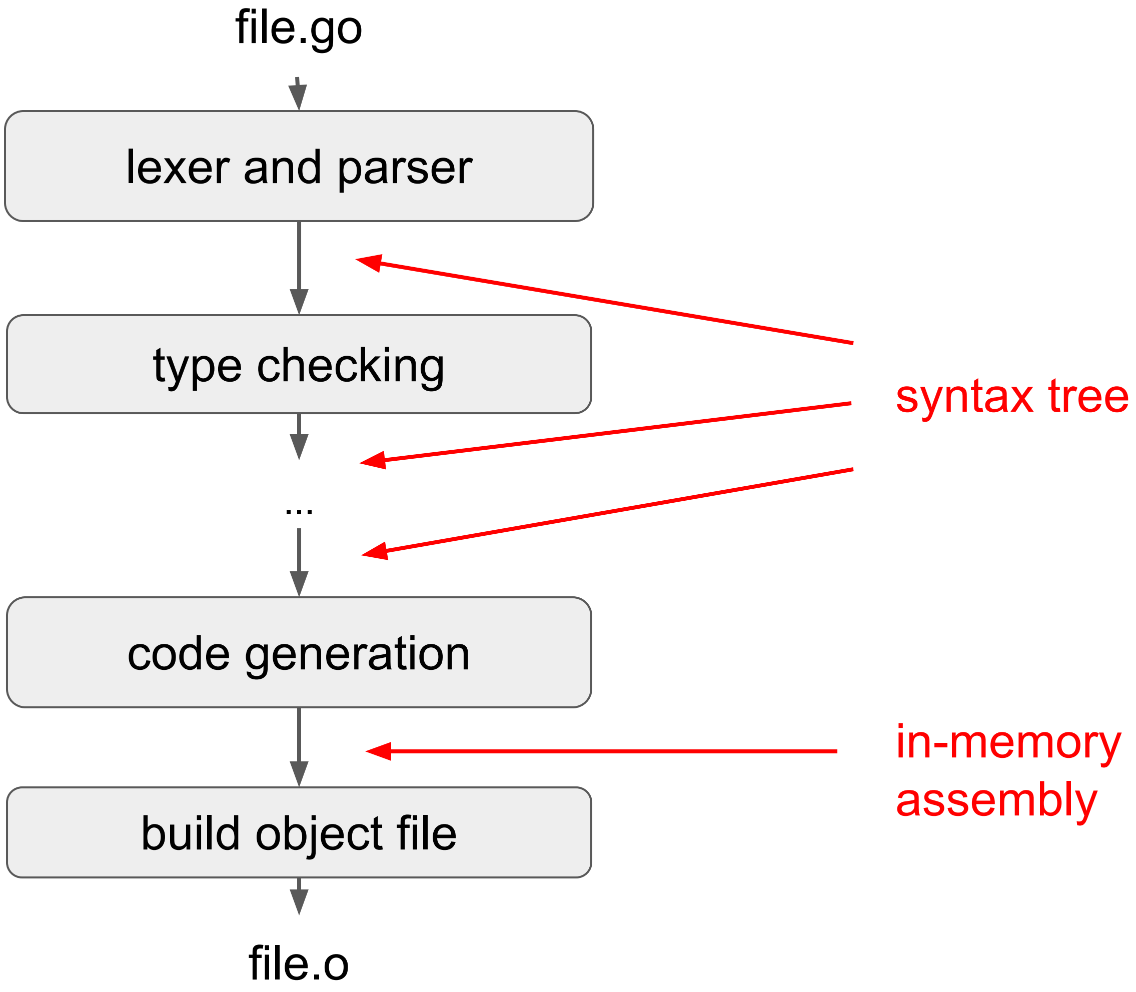

Intermediate representations (IR)
file.go
C1
C2
...
Cn
file.o
UTF-8 text file
object file
(assembly instructions)

“I’d like to convert from the current syntax-tree-based IR to a more modern SSA-based IR. With an SSA IR we can implement a lot of optimizations that are difficult to do in the current compiler.”

-me, Feb 2015

I’ll explain what all of these mean.

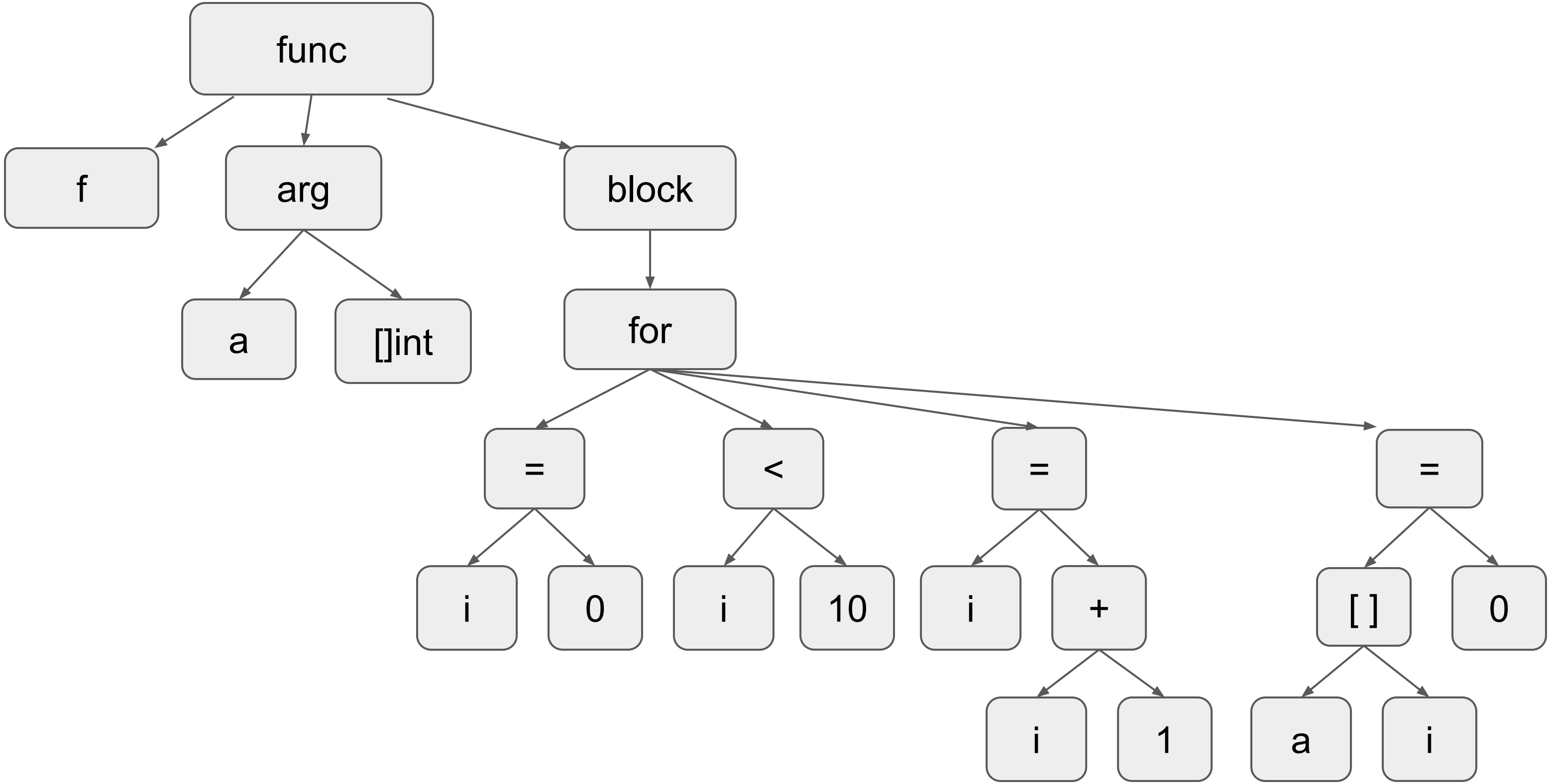
Go 1.5
compiler



Syntax tree

```
func f(a []int) {
    for i := 0; i < 10; i++ {
        a[i] = 0
    }
}
```

Syntax tree



Syntax-tree-based passes in Go 1.5

- Type checking
- Closure analysis
- Inlining
- Escape analysis
- Add temporaries where needed
- Introduce runtime calls (maps, channels, append, …)
- Code generation

“I’d like to convert from the current syntax-tree-based IR to a more modern SSA-based IR. With an SSA IR we can implement a lot of optimizations that are difficult to do in the current compiler.”

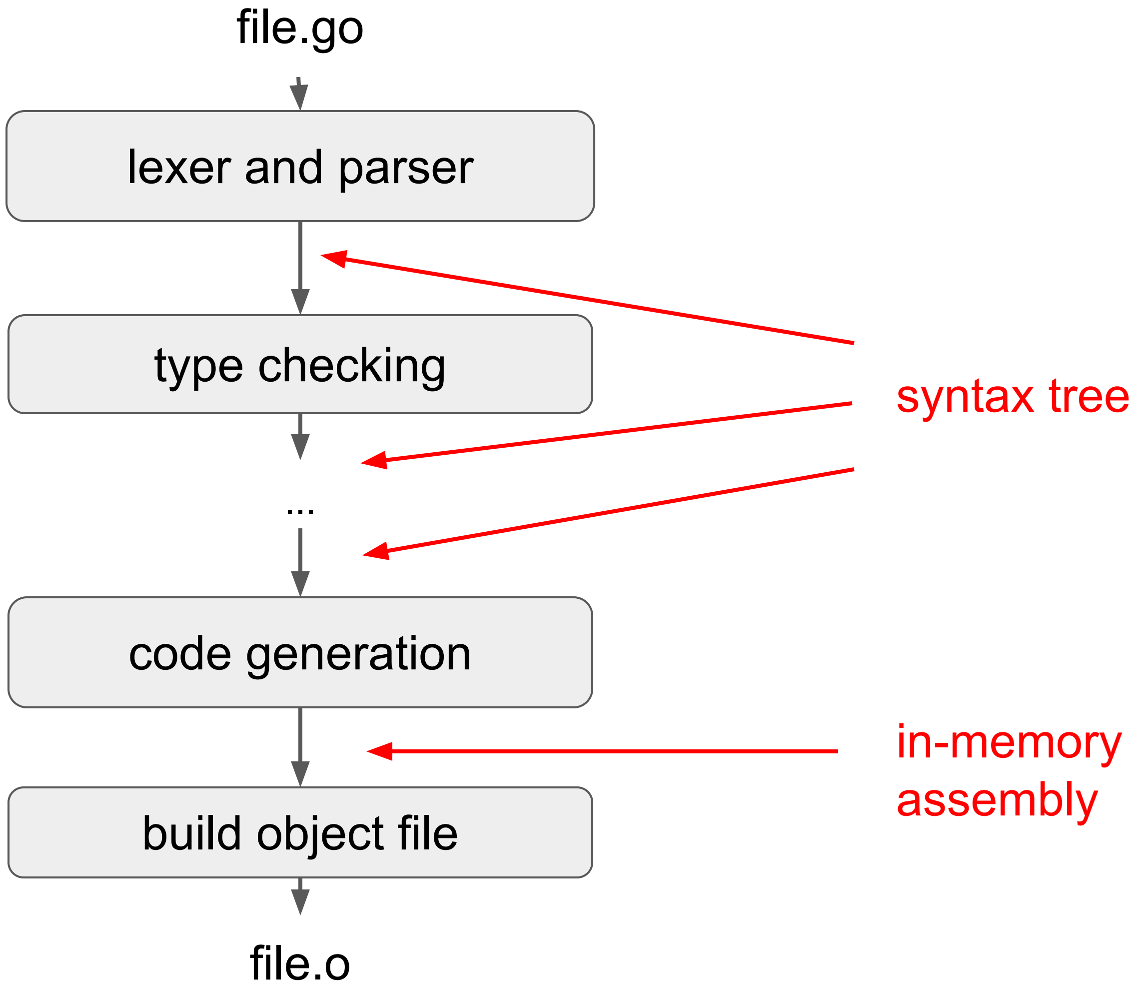
-me, Feb 2015

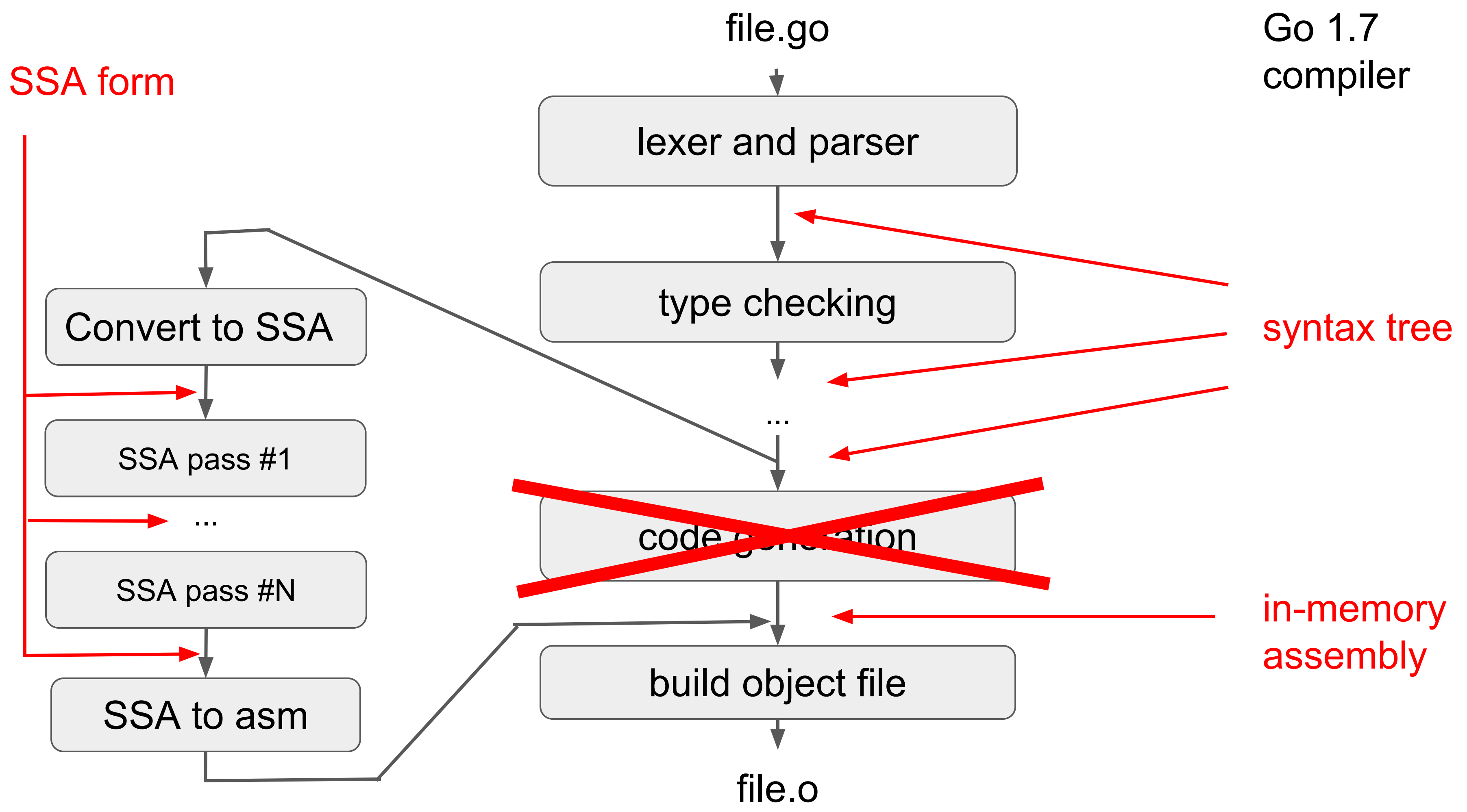
I’ll explain what all of these mean.

Syntax-tree-based passes in Go 1.~~5~~ 7

- Type checking
- Closure analysis
- Inlining
- Escape analysis
- Add temporaries where needed
- Introduce runtime calls (maps, channels, append, …)
- ~~Code generation~~ replace with SSA

Go 1.5 compiler




file.go
Go 1.7
compiler
SSA form
lexer and parser
type checking
syntax tree
...
Convert to SSA
SSA pass #1
...
SSA pass #N
~~code generation~~
in-memory
assembly
SSA to asm
build object file
file.o

S tatic
S ingle
A ssignment

One assignment per variable in the program.

```
x = 5
y = 7
z = x + y
x = y * 5
y = z - 7
z = x + 3
```

$\longrightarrow$

$$
\begin{aligned}
x_1 &= 5\\
y_1 &= 7\\
z_1 &= x_1 + y_1\\
x_2 &= y_1 * 5\\
y_2 &= z_1 - 7\\
z_2 &= x_2 + 3
\end{aligned}
$$

```
x = 7
if b {
  x = 8
}
fmt.Println(x)
```

$$\begin{array}{l} \texttt{x}_1\ \texttt{= 7} \\ \texttt{if b \{} \\ \quad \texttt{x}_2\ \texttt{= 8} \\ \texttt{\}} \\ \texttt{fmt.Println(x}_?\texttt{)} \end{array}$$

What number do we put here?

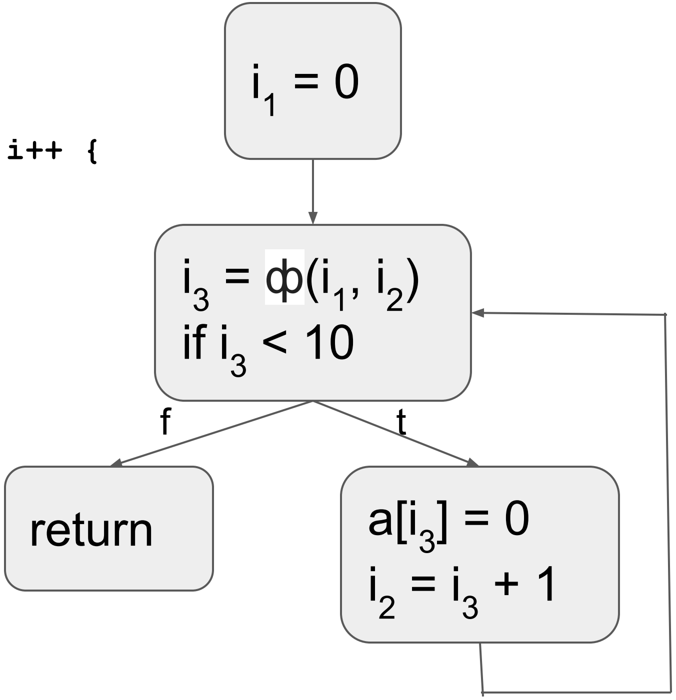
```
x = 7
if b {
  x = 8
}
fmt.Println(x)
```

$$\begin{array}{l} \mathtt{x}_1 \ \mathtt{=}\ \mathtt{7} \\ \mathtt{if\ b\ \{} \\ \quad \mathtt{x}_2 \ \mathtt{=}\ \mathtt{8} \\ \mathtt{\}} \\ \mathtt{x}_3 \ \mathtt{=}\ \phi(\mathtt{x}_1, \mathtt{x}_2) \\ \mathtt{fmt.Println}(\mathtt{x}_3) \end{array}$$

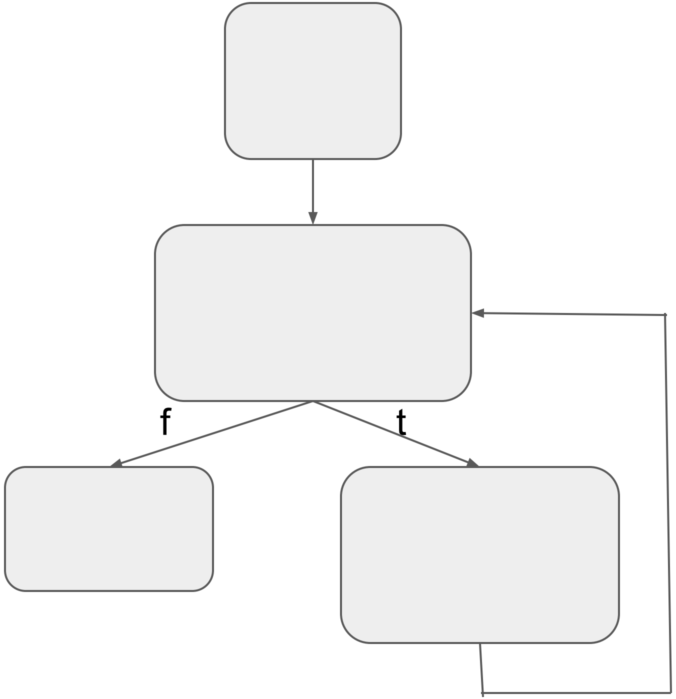
Φ functions represent explicitly what is otherwise an implicit merge point.

SSA Intermediate Representation

```
func f(a []int) {
	for i := 0; i < 10; i++ {
		a[i] = 0
	}
}
```



CFG - Control Flow Graph



= Basic Block

“I’d like to convert from the current syntax-tree-based IR to a more modern SSA-based IR. With an SSA IR we can implement a lot of optimizations that are difficult to do in the current compiler.”

-me, Feb 2015

I’ll explain what all of these mean.

SSA enables fast, accurate optimization algorithms for:

- Common Subexpression Elimination
- Dead Code Elimination
- Dead Store Elimination
- Nil Check Elimination
- Bounds Check Elimination
- Register allocation
- Instruction scheduling
- ...and more!

Common Subexpression Elimination

```
y = x + 5
…
z = x + 5
```

?

```
y = x + 5
…
z = y
```

Well, maybe. Let me look at all the code between the two assignments to see if x might be reassigned...

Yes!

SSA

Dead Store Elimination

```
*p = 0
…
*p = 1
```

?

```
~~*p = 0~~
…
*p = 1
```

Need to check that:

- **`p`** hasn't changed in …
- There is no control flow that avoids **`*p = 1`** in …
- There are no reads of **`*p`** in …

Dead Store Elimination

```
*p = 0
…
*p = 1
```

?

→

```
~~*p = 0~~
…
*p = 1
```

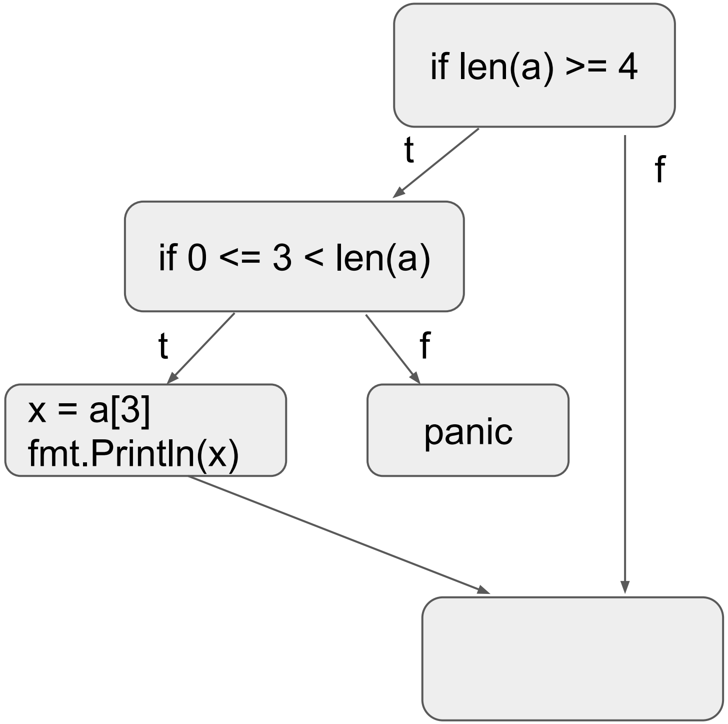
Need to check that:

SSA →
- **`p`** hasn't changed in …

CFG →
- There is no control flow that avoids **`*p = 1`** in …
- There are no reads of **`*p`** in …

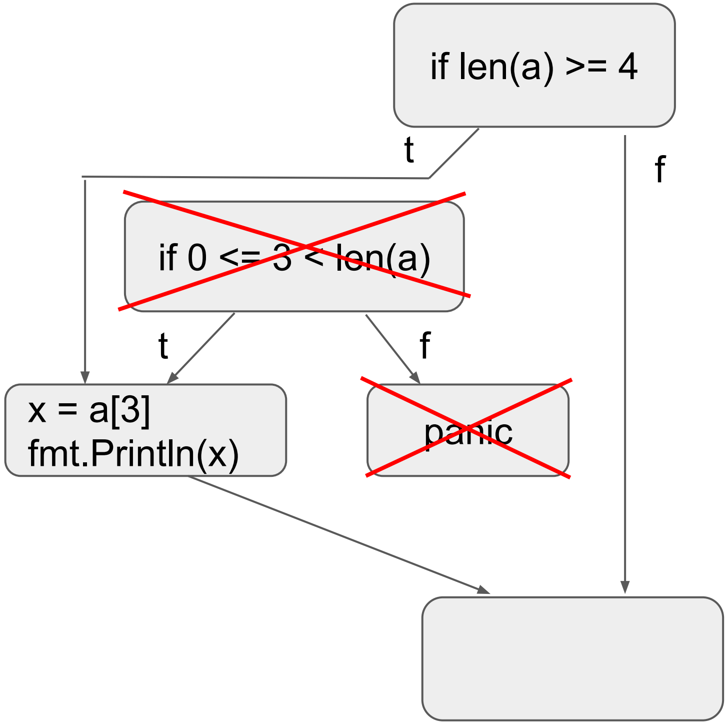
Bounds Check Elimination

```
if len(a) >= 4 {
    fmt.Println(a[3])
}
```



Bounds Check Elimination

```
if len(a) >= 4 {
    fmt.Println(a[3])
}
```



Rewrite Rules

Many optimizations can be specified using rewrite rules on the SSA form.

```
y = x - x        →        y = 0
```

(Sub64 x x) -> (Const64 [0])

```
y = 5 * 8      →      y = 40
```

(Mul64 (Const64 [c]) (Const64 [d])) -> (Const64 [c*d])

```
y = x * 16     →      y = x << 4
```

(Mul64 x (Const64 [c])) && isPowerOfTwo(c) -> (Lsh64x64 x (Const64 [log2(c)]))

```
z = x == y     →      w = x != y
w = !z
```

(Not (Eq64 x y)) -> (Neq64 x y)

```
z = x + y      →      w = y
w = z - x
```

(Sub64 (Add64 x y) x) -> y

Rewrite rules are also used to lower machine-independent operations to machine-dependent operations.

```
(Add64 x y) -> (ADDQ x y)
(Eq64 x y) -> (SETEQ (CMPQ x y))
```

Rewrite rules can get pretty complicated

```
(ORQ
    s1:(SHLQconst [j1] x1:(MOVBload [i1] {s} p mem))
    or:(ORQ
        s0:(SHLQconst [j0] x0:(MOVBload [i0] {s} p mem))
          y))
  && i1 == i0+1
  && j1 == j0+8
  && j0 % 16 == 0
  && x0.Uses == 1
  && x1.Uses == 1
  && s0.Uses == 1
  && s1.Uses == 1
  && or.Uses == 1
  && mergePoint(b,x0,x1) != nil
  && clobber(x0)
  && clobber(x1)
  && clobber(s0)
  && clobber(s1)
  && clobber(or)
  -> @mergePoint(b,x0,x1) (ORQ <v.Type> (SHLQconst <v.Type> [j0] (MOVWload [i0] {s} p mem)) y)
```

Rewrite rules can get pretty complicated

```
(ORQ
    s1:(SHLQconst [j1] x1:(MOVBload [i1] {s} p mem))
    or:(ORQ
        s0:(SHLQconst [j0] x0:(MOVBload [i0] {s} p mem))
          y))
  && i1 == i0+1
  && j1 == j0+8
  && j0 % 16 == 0
  && x0.Uses == 1
  && x1.Uses == 1
  && s0.Uses == 1
  && s1.Uses == 1
  && or.Uses == 1
  && mergePoint(b,x0,x1) != nil
  && clobber(x0)
  && clobber(x1)
  && clobber(s0)
  && clobber(s1)
  && clobber(or)
  -> @mergePoint(b,x0,x1) (ORQ <v.Type> (SHLQconst <v.Type> [j0] (MOVWload [i0] {s} p mem)) y)
```

Rewrite rules make new ports easy!

- It took a year to write the SSA backend for amd64.
- It took only a few months to write the SSA backends for all other architectures.

```
$ wc -l *.rules
  1253 386.rules
  2417 AMD64.rules
  1343 ARM64.rules
  1224 ARM.rules
   443 dec64.rules
    92 dec.rules
  1383 generic.rules
   700 MIPS64.rules
   731 MIPS.rules
   877 PPC64.rules
  1892 S390X.rules
```

Converting the compiler to use an SSA IR led to substantial improvements in the generated code.

	performance improvement	code size improvement
amd64	12%	13%
arm	20%	18%

Lots still to do:

- Alias analysis
  - Store-load forwarding
  - Better dead store removal
  - Devirtualization
- Better register allocation
- Better code layout
- Better instruction scheduling
- Lifting loop invariant code out of loops

… but only if it can be done efficiently and is demonstrably effective.

Thanks!

Alan Donovan
Alberto Donizetti
Alexandru Moșoi
Austin Clements
Brad Fitzpatrick
Cherry Zhang
Daniel Morsing
Dave Cheney
David Chase
David du Colombier
Ian Lance Taylor
Ilya Tocar

Josh Bleecher Snyder
Kevin Burke
Lynn Boger
Martin Möhrmann
Matthew Dempsky
Michael Matloob
Michael Munday
Michael Pratt
Minux Ma
Robert Griesemer
Russ Cox
Todd Neal