CS443: Compiler Construction

Lecture 17: Optimizations

Stefan Muller

Based on material by Stephen Chong

Reminders

- Midterm Tuesday
- Proj4 due tonight
  - If you turn it in tonight, I’ll try to grade it tomorrow
  - I’ll try to grade the rest Monday

Constant Propagation: If a var is known to be constant, replace var w/ constant

```
int N = 10000;
int a[] = malloc(N * sizeof(int));
for (int i = 0; i < N; i++) {
  a[i] = f(i, N);
}
```

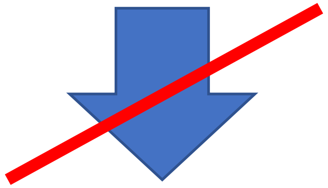
Now unnecessary!
(Dead code—later)

```
int N = 10000;
int a[] = malloc(10000 * sizeof(int));
for (int i = 0; i < 10000; i++) {
  a[i] = f(i, 10000);
}
```

Constant Propagation: Safety

- Need to make sure N isn't redefined

```
int N = 10000;
int a[] = malloc(10000 * sizeof(int));
for (int i = 0; i < N; i++) {
  a[i] = f(i, N);
  N = g(i);
}
```



Constant Propagation: In SSA, don't have to worry about other definitions

```
int N1 = 10000;
int a[] = malloc(N1 * sizeof(int));
N2 = Φ(N1, N3) for (int i = 0; i < N2; i++) {
  a[i] = f(i, N2);
  N3 = g(i);
}
```

Constant Propagation: Also have to worry about multiple branches

```
int a = 0;
if (p) {
  a = 1;
} else {
  a = 2;
}
```

- Normally: use reaching definitions analysis
- SSA: Fine

Generalize: **Copy Propagation**

```
b = a;
c = b + a;
```

```
b = a;
c = a + a;
```

Copy Propagation: Need to make sure neither var is redefined

```
int sqrt(int n) {
  int i = n;
  while (i * i > n)
    i--;
  return i;
}
```

```
int sqrt(int n) {
  int i = n;
  while (n * n > n)
    i = n - 1;
  return n;
}
```

```
temp = x;
x = y;
y = temp;
```

```
temp = x;
x = y;
y = x;
```

But we don't even have a move/set instruction! Why would you need this?

Other optimizations may generate move instructions (which you then need to get rid of to have valid LLVM!)

Constant Prop. Example #1

```
int n = 5;
int a = n * 2;
if (n < 6) n++
int b = n + 1;
```

Constant Prop. Example #1

```
int n = 5;
int a = n * 2;
if (n < 6) n++
int b = n + 1;
```

```
int n = 5;
int a = 5 * 2;
if (5 < 6) n = 5 + 1
int b = n + 1;
```

Constant Prop. Example #2

```
%n1 = bitcast i32 %x to i32
%x1 = add i32 %n1, 2
%n2 = mul i32 %n1, 2
%x2 = add i32 %n2, 5
```

Constant Prop. Example #2

```
%n1 = bitcast i32 %x to i32
%x1 = add i32 %n1, 2
%n2 = mul i32 %n1, 2
%x2 = add i32 %n2, 5
```

```
%n1 = bitcast i32 %x to i32
%x1 = add i32 %x, 2
%n2 = mul i32 %x, 2
%x2 = add i32 %n2, 5
```

Constant Prop. Example #3

```
int n = 5;
int a = f(&n);
int b = n + 1;
```

Constant Prop. Example #3

```
int n = 5;
int a = f(&n);
int b = n + 1;
```

```
int n = 5;
int a = f(&n);
int b = n + 1;
```

Might change n

Constant/Copy Propagation: Summary

- What does it optimize?
  - Time
  - Space (registers + maybe stack)
- When is it safe?
  - Non-SSA: if variable(s) haven't been redefined
  - SSA: always
- When is it an optimization?
  - Always (at least, shouldn't make things worse)

Implementing Constant/Copy Propagation on SSA code

- For each instruction “x = c”:
  - Add x -> c to a map
- For each instruction (e.g. x1 = add i32 v1, v2):
  - Look up variable operands in map, replace with value if there

LLVM specifics

- If %x -> c is in the map, can we propagate c to, e.g., `getelementptr i32, i32* %x, i32 5`?
  - Only if c is another variable (it should be if we're not manipulating bare memory addresses)

- What LLVM instructions have the form "%x = c"?
  - In LLVM.Ast: `ISet(%x, t, c)`
  - `%x = bitcast t1 c to t2` when t1 = t2
  - `%x = phi t [c1, %l1], …, [cn, %ln]` when c1 = … = cn

Constant Folding: Do arithmetic with constants when you can

```
int a[] = malloc(10000 * sizeof(int));
for (int i = 0; i < 10000; i++) {
  a[i] = f(i, 10000);
}
```

=4

```
int a[] = malloc(40000);
for (int i = 0; i < 10000; i++) {
  a[i] = f(i, 10000);
}
```

Constant Folding: Who’s written code like this?

```
int seconds_to_days (int secs) {
  return secs / (60 * 60 * 24);
}
```

Constant Folding: Can extend to things that aren't just simple arithmetic

```
int x = 3;
int y = 4;
if (x < y) {
 s1;
} else {
 s2;
}
```

→

```
if (3 < 4) {
 s1;
} else {
 s2;
}
```

→

```
if (true) {
 s1;
} else {
 s2;
}
```

→

```
s1;
```

s2 is unreachable!
(More on that later)

This (previous slides) kind of constant folding is usually safe, but don't get carried away

`true && x && false`	→	`false && x`	✗
`1 + x + 2`	→	`3 + x`	✓
`42 / 0`	→	`INTERNAL COMPILER ERROR: Divide by z`	?
`1.0 + x + 2.0`	→	`3.0 + x`	

Aside on constant folding divisions by zero

```
let a = 5 / 0
```

ocamlopt

```
0000000000012cd0 <camlTest__entry>:
   12cd0:    sub    $0x8,%rsp
   12cd4:    lea 0x23bb0d(%rip),%rax      # 24e7e8 <caml_backtrace_pos>
   12cdb:    xor    %rbx,%rbx
   12cde:    mov    %ebx,(%rax)
   12ce0:    lea 0x229421(%rip),%rax      # 23c108 <caml_exn_Division_by_zero>
   12ce7:    callq  2c130 <caml_raise_exn>
```

Aside on constant folding divisions by zero

```
return 5 / 0;
```

gcc –O0

```
00000000000005fa <main>:
 5fa:    push   %rbp
 5fb:    mov    %rsp,%rbp
 5fe:    mov    $0x5,%eax
 603:    mov    $0x0,%ecx
 608:    cltd
 609:    idiv   %ecx
 60b:    pop    %rbp
 60c:    retq
```

gcc –O3

```
00000000000004f0 <main>:
 4f0:    ud2
```

UD2	0F 0B	Undefined Instruction	Generates an invalid opcode exception. This instruction is provided for software testing to explicitly generate an invalid opcode. The opcode for this instruction is reserved for this purpose.

Related: Even if we don't have all constants, we can do **algebraic simplifications**

a * 1 = a

1 * a = a

a + 0 = a

0 + a = a

a – 0 = a

a * 0 = 0

0 * a = 0

Again, careful of these, e.g.
(5 / z) * 0 if z = 0
or
(f()) * 0 if f has a side effect

a && true = a

true && a = a

a && false = false

false && a = false

a || true = true

true || a = true

a || false = a

false || a = a

Implementing Constant Folding

- Hope you like pattern matching
- Can do this at the same time as copy/constant propagation

Constant Folding: Summary

- What does it optimize?
  - Time
- When is it safe?
  - If it doesn’t change side effect behavior and doesn’t do weird FP stuff
- When is it an optimization?
  - Well, depends on what we do after

Optimizations can enable other optimizations

```
int secs_per_day = 60 * 60 * 24;
int days = secs / secs_per_day;
```

Constant Folding

```
int secs_per_day = 86400;
int days = secs / secs_per_day;
```

Constant Propagation

```
int secs_per_day = 86400;
int days = secs / 86400;
```

Optimizations can enable other optimizations: **Dead Code Elimination**

```
int secs_per_day = 60 * 60 * 24;
int days = secs / secs_per_day;
```

Constant Folding

```
int secs_per_day = 86400;
int days = secs / secs_per_day;
```

Constant Propagation

```
int secs_per_day = 86400;
int days = secs / 86400;
```

Unused assignment (“Dead code”)

Dead Code Elimination: defs with no uses can (sometimes) be safely removed

```
int secs_per_day = 60 * 60 * 24;
int days = secs / secs_per_day;
```

Constant Folding

```
int secs_per_day = 86400;
int days = secs / secs_per_day;
```

Constant Propagation

```
int secs_per_day = 86400;
int days = secs / 86400;
```

Dead Code Elim.

```
int days = secs / 86400;
```

Dead Code Elimination: Be careful of side effects!

```
int x = printf("Hello World\n");
// Never use x, because who does error checking
// on printf?
```

Could argue that, in this class, LLVM instructions have no side effects other than the one variable assignment

Can we eliminate dead code that raises an exception?

```c
int a = 5 / 0;
return 0;
```

```ocaml
let a = 5 / 0
exit 0
```

Changes program behavior… but only in a “good” way?

- gcc (with –O1 and above) eliminates assignment
- ocamlopt doesn’t

DCE: Summary

- What does it optimize?
  - Time
  - Space
  - Instruction cache
- When is it safe?
  - If it doesn’t change side effect behavior
- When is it an optimization?
  - Probably always, but processors are very weird

Implementing DCE – intuitive algorithm

- Initialize uses(i) to 0 for all instructions
- For each instruction (e.g., x = add i32, v1, v2):
  - Increment uses(vi) for all variable operands vi (x is not used)
- Then:
- For each definition x = e:
  - If uses(x) = 0 *and e has no side effects*, eliminate

Intuitive algo might lead to more dead code

```
int a = 2;
int b = a * 2;
int c = b + 4;
return 8;
```

```
int a = 2;
int b = a * 2;
return 8;
```

```
int a = 2;
return 8;
```

```
return 8;
```

More efficient worklist algorithm

- Let V = all vars in the program
- while V is not empty:
  - remove v from V
  - if uses_of(v) is empty and def_of(v) has no other side effects:
    - remove def_of(v) from the program
    - for x in use(def_of(v)) / {v}:
      - delete def_of(v) from uses_of(x)
      - V = V U {x}

For best results: Fold, propagate, eliminate, and repeat

```
int a = 2;
int b = a * 2;
int c = 3;
int d = 0;
if (b > c) {
  d = 2;
} else {
  d = 3;
}
return d;
```

```
int b = 2 * 2;
int d;
if (b > 3) {
  d = 2;
} else {
  d = 3;
}
return d;
```

```
int d;
if (4 > 3) {
  d = 2;
} else {
  d = 3;
}
return d;
```

```
int d = 2;
return d;
```

```
return 2;
```

(Stop when code doesn't get any shorter)

Example #4

```
  %temp1$1 = bitcast i32 %n to i32
  %temp2$2 = bitcast i32 1 to i32
  %a$3 = add i32 %temp1$1, %temp2$2
  %temp7$4 = bitcast i32 1 to i32
  %temp8$5 = bitcast i32 2 to i32
  %temp5$6 = add i32 %temp7$4, %temp8$5
  %temp6$7 = bitcast i32 4 to i32
  %temp4$8 = icmp sgt i32 %temp5$6, %temp6$7
  br i1 %temp4$8, label %label1, label %label2
label1:
  %temp3$10 = bitcast i32 %a$3 to i32
  ret i32 %temp3$10
label2:
  br label %label3
label3:
  %temp9$9 = bitcast i32 1 to i32
  ret i32 %temp9$9
```

Example #4

```
  %a$3 = add i32 %n, 1
  %temp5$6 = add i32 1, 2
  %temp4$8 = icmp sgt i32 %temp5$6, 4
  br i1 %temp4$8, label %label1, label %label2
label1:
  ret i32 %a$3
label2:
  br label %label3
label3:
  ret i32 1
```

Example #4

```
  %a$3 = add i32 %n, 1
  %temp5$6 = bitcast i32 3 to i32
  %temp4$8 = icmp sgt i32 %temp5$6, 4
  br i1 %temp4$8, label %label1, label %label2
label1:
  ret i32 %a$3
label2:
  br label %label3
label3:
  ret i32 1
```

Example #4

```
  %a$3 = add i32 %n, 1
  %temp4$8 = icmp sgt i32 3, 4
  br i1 %temp4$8, label %label1, label %label2
label1:
  ret i32 %a$3
label2:
  br label %label3
label3:
  ret i32 1
```

Example #4

```
  %a$3 = add i32 %n, 1
  %temp4$8 = bitcast i1 0 to i1
  br i1 %temp4$8, label %label1, label %label2
label1:
  ret i32 %a$3
label2:
  br label %label3
label3:
  ret i32 1
```

Example #4

```
  %a$3 = add i32 %n, 1
  br i1 0, label %label1, label %label2
label1:
  ret i32 %a$3
label2:
  br label %label3
label3:
  ret i32 1
```

Example #4

```
  %a$3 = add i32 %n, 1
  br label %label2
label1:
  ret i32 %a$3
label2:
  br label %label3
label3:
  ret i32 1
```

Example #4

```
  %a$3 = add i32 %n, 1
  br label %label2
label2:
  br label %label3
label3:
  ret i32 1
```

Next time:
Delete unreachable code!

Example #4

```
  br label %label2
label2:
  br label %label3
label3:
  ret i32 1
```

Reusing available expressions

```
int c = a + b;
int d = a + b;
```

→

```
int c = a + b;
int d = c;
```

Recall: Available expressions dataflow analysis

```
int c = a + b;
int d = a + b;
```

```
int c = a + b;
int d = c;
```

Available: c = a + b

Available expressions analysis takes care of safety for reusing expressions

```
int c = a + b;
c = c + 1;
int d = a + b;
```

c = a + b no longer available

```
int c = a + b;
a = a + 1;
int d = a + b;
```

(and this isn't a concern in SSA, anyway, but there is still one concern)

Algorithm for reusing expressions

- Perform available expressions dataflow analysis
- For each node n where stmt(n) is “v = e1”:
  - If an assignment “a = e2” is available where e1 = e2
    - Replace n with “v = a”

Not totally straightforward.
e.g., does x + y = y + x?

Reusing expressions example

- Can we optimize this?

```
int d = a[i] + b[i];
c[j] = d;
int e = a[i] + b[i];
```

- A: Not necessarily. c[j] might alias a[i] or b[i].
- Available expressions should handle this (recall: c[j] = d will kill any uses that might alias c[j]).
- Conservative approximation: stores kill all available expressions

Common Subexpression Elimination (CSE)

```
int c = (a + b) * 2;
int d = (a + b) * 3;
```

→

```
int temp = a + b;
int c = temp * 2;
int d = temp * 3
```

In LLVM, CSE = reusing expressions!

```
int c = (a + b) * 2;
int d = (a + b) * 3;
```

```
int temp = a + b;
int c = temp * 2;
int d = temp * 3
```

```
%temp1 = add i32 %a %b
%c = mul i32 %temp1 2
%temp2 = add i32 %a %b
%d = mul i32 %temp2 c
```

RE

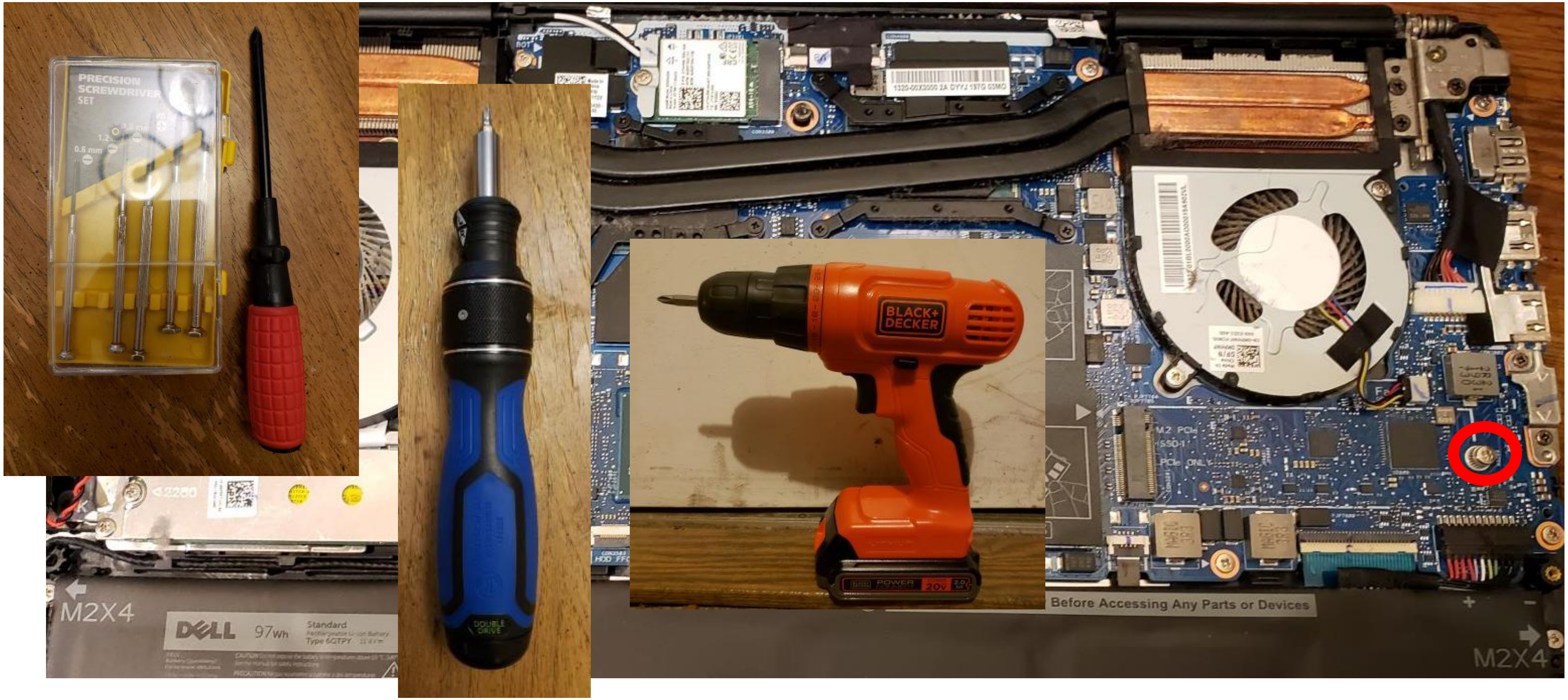
```
%temp1 = add i32 %a %b
%c = mul i32 %temp1 2
%temp2 = %temp1
%d = mul i32 %temp2 c
```

CP, DCE can further optimize

Common Subexp. Elim.: Summary

- What does it optimize?
  - Time, space
- When is it safe?
  - As long as expressions are available
- When is it an optimization?
  - Probably always

Strength Reduction



Strength Reduction – Replace expensive operations w/ cheaper ones


```
x * 2
x * 3
```

```
x + x
x + x + x
```


```
x << 1
```

Strength Reduction: Summary

- What does it optimize?
  - Time (at proc. cycle level)
- When is it safe?
  - Always
- When is it an optimization?
  - When assumptions about architecture hold

Function Inlining: function calls are *expensive*

```
int double(int a) {
  return a * 2;
}

int main() {
  int a = 2;
  int b = double(a);
  int c = double(b);
  return c;
}
```

```
__double:
  addi sp,sp,-8
  sw fp,4(sp)
  sw ra,0(sp)
  addi fp,sp,4
  addi sp,sp,0
…
double__exit:
  addi sp,fp,-4
  lw fp,4(sp)
  lw ra,0(sp)
  addi sp,sp,8
  jalr zero,ra,0
```

```
  lw t0,double
  jalr ra,t0,0
  addi sp,sp,-32
  addi s1,a0,0
```

Function Inlining: function calls are *expensive*

```
int double(int a) {
  return a * 2;
}

int main() {
  int a = 2;
  int b = double(a);
  int c = double(b);
  return c;
}
```

→

```
int main() {
  int a = 2;
  int b = a * 2;
  int c = b * 2;
  return c;
}
```

Function Inlining Exercise

```
define i32 @f(i32 %a, i1 %b) {
f__entry:
  br i1 %b, label %ltrue, label %lfalse
ltrue:
  %temp1 = add i32 %a 1
  ret %temp1
lfalse:
  %temp2 = sub i32 %a 1
  ret %temp2
}

define @g(i32 %c) {
  %r = call i32 @f(i32 %c, i1 1)
  return i32 %r
}
```

Function Inlining Exercise

```
define i32 @f(i32 %a, i1 %b) {
f__entry:
  br i1 %b, label %ltrue, label %lfalse
ltrue:
  %temp1 = add i32 %a 1
  ret %temp1
lfalse:
  %temp2 = sub i32 %a 1
  ret %temp2
}
```

1. Substitute arguments
2. Replace returns with assignments to the dest. of the call
3. Add branches back to main code (may not be necessary if only one ret)

```
define @g(i32 %c) {
 br i1 1, label %ltrue, label %lfalse
ltrue:
  %temp1 = add i32 %c 1
  %r = %temp1
  br label %ldone
lfalse:
  %temp2 = sub i32 %c 1
  %r = %temp2
  br label %ldone
ldone:
  return i32 %r
}
```

Inlining: Tradeoffs

- Saves instructions for function call overhead (optimizes *time*)
- But makes code bigger—makes instruction cache usage worse
- Tradeoff: usually only inline smaller functions
  - Some compilers give the inlining threshold as an option

Eliminating unreachable code: if there are no branches to a label, can get rid of the block

```
  br label %ltrue
ltrue:
  %a = add %b 1
  br label %l1
lfalse:
  %a = add %c 2
  br label %l2
l1:
  %d = mul %a 2
  br label %ldone
l2:
  %d = mul %a 3
  br label %ldone
ldone:
  ret %a
```

```
  br label %ltrue
ltrue:
  %a = add %b 1
  br label %l1

l1:
  %d = mul %a 2
  br label %ldone
l2:
  %d = mul %a 3
  br label %ldone
ldone:
  ret %a
```

```
  br label %ltrue
ltrue:
  %a = add %b 1
  br label %l1

l1:
  %d = mul %a 2
  br label %ldone

ldone:
  ret %a
```

Already done in LLVM.SSA (necessary for SSA conv.)