IR Optimization
Announcements

- Programming Project 4 due **Wednesday, August 10** at 11:59PM.
- Written Assignment 2 graded.
  - Hard copies returned after lecture.
  - Electronic copies returned by email.
- Programming Assignment 2 grading almost done.
  - We'll email back feedback later today.
Where We Are

Source Code

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- IR Generation
- IR Optimization
- Code Generation
- Optimization

Machine Code
Where We Are


Achievement unlocked
Wond-IR-ful!
Where We Are

Source Code

Lexical Analysis

Syntax Analysis

Semantic Analysis

IR Generation

IR Optimization

Code Generation

Optimization

Machine Code
IR Optimization

- **Goal**: Improve the IR generated by the previous step to take better advantage of resources.
- One of the most important and complex parts of any modern compiler.
- A very active area of research.
- There is a whole class (CS243) dedicated to this material.
Sources of Optimization

• In order to optimize our IR, we need to understand why it can be improved in the first place.

• **Reason one:** IR generation introduces redundancy.
  • A naïve translation of high-level language features into IR often introduces subcomputations.
  • Those subcomputations can often be sped up, shared, or eliminated.

• **Reason two:** Programmers are lazy.
  • Code executed inside of a loop can often be factored out of the loop.
  • Language features with side effects often used for purposes other than those side effects.
Optimizations from IR Generation

```c
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```
Optimizations from IR Generation

```c
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```c
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;

_t2 = x + x;
_t3 = y;
b2 = _t2 == _t3;

_t4 = x + x;
_t5 = y;
b3 = _t5 < _t4;
```
Optimizations from IR Generation

```plaintext
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```plaintext
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;

_t2 = x + x;
_t3 = y;
b2 = _t2 == _t3;

_t4 = x + x;
_t5 = y;
b3 = _t5 < _t4;
```
Optimizations from IR Generation

```c
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```c
_t0 = x + x;
t1 = y;
b1 = _t0 < _t1;

b2 = _t0 == _t1;

b3 = _t0 < _t1;
```
while (x < y + z) {
    x = x - y;
}

Optimizations from Lazy Coders
while (x < y + z) {
    x = x - y;
}

_L0:
    _t0 = y + z;
    _t1 = x < _t0;
    IfZ _t1 Goto _L1;
    x = x - y;
    Goto _L0;

_L1:
while (x < y + z) {
    x = x - y;
}

_L0:
  _t0 = y + z;
  _t1 = x < _t0;
  IfZ _t1 Goto _L1;
  x = x - y;
  Goto _L0;

_L1:
while \( x < y + z \) {
    \( x = x - y; \)
}

\_t0 = y + z;
\_L0:
\_t1 = x < \_t0;
IfZ \_t1 Goto \_L1;
x = x - y;
Goto \_L0;
\_L1:
while (x < y + z) {
    x = x - y;
}

_t0 = y + z;
_L0:
_t1 = x < _t0;
IfZ _t1 Goto _L1;
    x = x - y;
    Goto _L0;
_L1:
A Note on Terminology

- The term “optimization” implies looking for an “optimal” piece of code for a program.
- This is, in general, undecidable.
  - e.g. create a program that can be simplified iff some other program halts.
- Our goal will be **IR improvement** rather than **IR optimization**.
The Challenge of Optimization

• A good optimizer
  • Should never change the observable behavior of a program.
  • Should produce IR that is as efficient as possible.
  • Should not take too long to process inputs.

• Unfortunately:
  • Even good optimizers sometimes introduce bugs into code.
  • Optimizers often miss “easy” optimizations due to limitations of their algorithms.
  • Almost all interesting optimizations are NP-hard.
What are we Optimizing?

- Optimizers can try to improve code usage with respect to many observable properties.
- What are some quantities we might want to optimize?

  - Runtime (make the program as fast as possible at the expense of time and power)
  - Memory usage (generate the smallest possible executable at the expense of time and power)
  - Power consumption (choose simple instructions at the expense of speed and memory usage)
  - Plus a lot more (minimize function calls, reduce use of floating-point hardware, etc.)
What are we Optimizing?

- Optimizers can try to improve code usage with respect to many observable properties.
- What are some quantities we might want to optimize?
  - **Runtime** (make the program as fast as possible at the expense of time and power)
  - **Memory usage** (generate the smallest possible executable at the expense of time and power)
  - **Power consumption** (choose simple instructions at the expense of speed and memory usage)
  - Plus a lot more (minimize function calls, reduce use of floating-point hardware, etc.)
IR Optimization vs Code Optimization

- There is not always a clear distinction between what belongs to “IR optimization” versus “code optimization.”

- Typically:
  - IR optimizations try to perform simplifications that are valid across all machines.
  - Code optimizations try to improve performance based on the specifics of the machine.

- Some optimizations are somewhere in-between:
  - Replacing \( x / 2 \) with \( x * 0.5 \)
Overview of IR Optimization

- **Formalisms and Terminology** *(Today)*
  - Control-flow graphs.
  - Basic blocks.

- **Local optimizations** *(Today)*
  - Speeding up small pieces of a function.

- **Global optimizations** *(Monday)*
  - Speeding up functions as a whole.

- **The dataflow framework** *(Monday/Wednesday)*
  - Defining and implementing a wide class of optimizations.

- **Lazy code motion** *(Wednesday)*
  - An extremely powerful IR optimization used in many compilers.
Formalisms and Terminology
Analyzing a Program

• In order to optimize a program, the compiler has to be able to reason about the properties of that program.

• An analysis is called **sound** if it never asserts an incorrect fact about a program.

• All the analyses we will discuss in this class are sound.
  • *(Why?)*
int x;
int y;
if (y < 5)
    x = 137;
else
    x = 42;
Print(x);
int x;
int y;

if (y < 5)
    x = 137;
else
    x = 42;

Print(x);
At this point in the program, \( x \) holds some integer value.
int x;
int y;

if (y < 5)
    x = 137;
else
    x = 42;

Print(x);

“At this point in the program, x is either 137 or 42”
Soundness

```c
int x;
int y;

if (y < 5)
    x = 137;
else
    x = 42;

Print(x);
```

“At this point in the program, x is 137”
At this point in the program, x is either 137, 42, or 271
Semantics-Preserving Optimizations

- An optimization is **semantics-preserving** if it does not alter the semantics of the original program.

- Examples:
  - Eliminating unnecessary temporary variables.
  - Computing values that are known statically at compile-time instead of runtime.
  - Evaluating constant expressions outside of a loop instead of inside.

- Non-examples:
  - Replacing bubble sort with quicksort.

- The optimizations we will consider in this class are all semantics-preserving.
A Formalism for IR Optimization

• Every phase of the compiler uses some new abstraction:
  • Scanning uses regular expressions.
  • Parsing uses CFGs.
  • Semantic analysis uses proof systems and symbol tables.
  • IR generation uses ASTs.

• In optimization, we need a formalism that captures the structure of a program in a way amenable to optimization.
main:
  BeginFunc 40;
  _tmp0 = LCall _ReadInteger;
  a = _tmp0;
  _tmp1 = LCall _ReadInteger;
  b = _tmp1;
_L0:
  _tmp2 = 0;
  _tmp3 = b == _tmp2;
  _tmp4 = 0;
  _tmp5 = _tmp3 == _tmp4;
  IfZ _tmp5 Goto _L1;
  c = a;
  a = b;
  _tmp6 = c % a;
  b = _tmp6;
  Goto _L0;
_L1:
  PushParam a;
  LCall _PrintInt;
  PopParams 4;
EndFunc;
main:
    BeginFunc 40;
    _tmp0 = LCall _ReadInteger;
    a = _tmp0;
    _tmp1 = LCall _ReadInteger;
    b = _tmp1;
_L0:
    _tmp2 = 0;
    _tmp3 = b == _tmp2;
    _tmp4 = 0;
    _tmp5 = _tmp3 == _tmp4;
    IfZ _tmp5 Goto _L1;
    c = a;
    a = b;
    _tmp6 = c % a;
    b = _tmp6;
    Goto _L0;
_L1:
    PushParam a;
    LCall _PrintInt;
    PopParams 4;
    EndFunc;
main:
  BeginFunc 40;
  _tmp0 = LCall _ReadInteger;
a = _tmp0;
  _tmp1 = LCall _ReadInteger;
b = _tmp1;
_L0:
  _tmp2 = 0;
  _tmp3 = b == _tmp2;
  _tmp4 = 0;
  _tmp5 = _tmp3 == _tmp4;
  IfZ _tmp5 Goto _L1;
c = a;
a = b;
  _tmp6 = c % a;
b = _tmp6;
Goto _L0;
_L1:
  PushParam a;
  LCall _PrintInt;
  PopParams 4;
  EndFunc;
Visualizing IR

main:
BeginFunc 40;
.tmp0 = LCall _ReadInteger;
a = .tmp0;
.tmp1 = LCall _ReadInteger;
b = .tmp1;
_L0:
.tmp2 = 0;
.tmp3 = b == .tmp2;
.tmp4 = 0;
.tmp5 = .tmp3 == .tmp4;
IfZ .tmp5 Goto _L1;
c = a;
a = b;
.tmp6 = c % a;
b = .tmp6;
Goto _L0;
_L1:
PushParam a;
LCall _PrintInt;
PopParams 4;
EndFunc;
Basic Blocks

- A **basic block** is a sequence of IR instructions where
  - There is exactly one spot where control enters the sequence, which must be at the start of the sequence.
  - There is exactly one spot where control leaves the sequence, which must be at the end of the sequence.
- Informally, a sequence of instructions that always execute as a group.
Control-Flow Graphs

• A control-flow graph (CFG) is a graph of the basic blocks in a function.
  • The term CFG is overloaded – from here on out, we'll mean “control-flow graph” and not “context-free grammar.”

• Each edge from one basic block to another indicates that control can flow from the end of the first block to the start of the second block.

• There is a dedicated node for the start and end of a function.
Types of Optimizations

• An optimization is **local** if it works on just a single basic block.

• An optimization is **global** if it works on an entire control-flow graph.

• An optimization is **interprocedural** if it works across the control-flow graphs of multiple functions.
  
  • We won't talk about this in this course.
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
Local Optimizations

```c
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0) {
        z = y;
    } else {
        x = y;
    }
}
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}

Local Optimizations

start

\[ y = 137; \]
\[ \text{IfZ x Goto } _L0; \]

\[ z = y; \]
\[ \_t2 = y; \]
\[ x = \_t2; \]

end
Local Optimizations

```c
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}

Local Optimizations

y = 137;
IfZ x Goto _L0;

z = y;

x = y;

end
Local Optimizations

```c
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```

```
y = 137;
IfZ x Goto _L0;
```

```
z = y;
x = y;
```

```
end
```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}

```
Global Optimizations
```

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    IfZ x Goto _L0;

    z = y;
    x = y;
}
```

```
start

y = 137;
IfZ x Goto _L0;

z = y;

x = y;

end
```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}

Global Optimizations

start

y = 137;
IfZ x Goto _L0;

z = y;

x = y;

end
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}

Global Optimizations

y = 137;
IfZ x Goto _L0;

z = 137;
x = 137;
end
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}

Global Optimizations

```
y = 137;
IfZ x Goto _L0;

start

y = 137;
IfZ x Goto _L0;

z = 137;
```

```
x = 137;
```

end
Local Optimizations
Common Subexpression Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
Common Subexpression Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;

x.fn(a + b);
Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;
x = _tmp1;
_tmp3 = 4;
a = _tmp3;
_tmp4 = a + b;
c = _tmp4;
_tmp5 = a + b;
_tmp6 = *(x);
_tmp7 = *(tmp6);
PushParam _tmp5;
PushParam x;
ACall _tmp7;
PopParams 8;
```
Common Subexpression Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*( _tmp1 ) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = *( x ) ;
_tmp7 = *( _tmp6 ) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
Common Subexpression Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;
x = _tmp1;
_tmp3 = 4;
a = _tmp3;
_tmp4 = a + b;
c = _tmp4;
_tmp5 = _tmp4;
_tmp6 = *(x);
_tmp7 = *(tmp6);
PushParam _tmp5;
PushParam x;
ACall _tmp7;
PopParams 8;
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
Common Subexpression Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

__tmp0 = 4;
PushParam __tmp0;
__tmp1 = LCall _Alloc;
PopParams 4;
__tmp2 = Object;
*(__tmp1) = __tmp2;
x = __tmp1;
__tmp3 = __tmp0;
a = __tmp3;
__tmp4 = a + b;
c = __tmp4;
__tmp5 = __tmp4;
__tmp6 = *(x);
__tmp7 = *(__tmp6);
PushParam __tmp5;
PushParam x;
ACall __tmp7;
PopParams 8;
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
Common Subexpression Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = a + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = *(x);
_tmp7 = *(tmp6);
PushParam _tmp5;
PushParam x;
ACall _tmp7;
PopParams 8;
Common Subexpression Elimination

- If we have two variable assignments
  \[ v_1 = a \text{ op } b \]
  
  \[ \ldots \]
  
  \[ v_2 = a \text{ op } b \]

  and the values of \( v_1 \), \( a \), and \( b \) have not changed between the assignments, rewrite the code as

  \[ v_1 = a \text{ op } b \]

  \[ \ldots \]

  \[ v_2 = v_1 \]

- Eliminates useless recalculation.
- Paves the way for later optimizations.
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = a + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = *(x);
_tmp7 = *(tmp6);
PushParam _tmp5;
PushParam x;
ACall _tmp7;
PopParams 8;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(__tmp1) = _tmp2;
x = __tmp1;
_tmp3 = _tmp0;
a = __tmp3;
_tmp4 = a + b;
c = __tmp4;
_tmp5 = c;
_tmp6 = *(x);
_tmp7 = *(__tmp6);
PushParam __tmp5;
PushParam x;
ACall __tmp7;
PopParams 8;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(__tmp1) = _tmp2;
x = __tmp1;
_tmp3 = _tmp0;
a = __tmp3;
_tmp4 = a + b;
c = __tmp4;
_tmp5 = c;
_tmp6 = *(__tmp1);
_tmp7 = *(__tmp6);
PushParam __tmp5;
PushParam __tmp1;
ACall __tmp7;
PopParams 8;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = a + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = *(_tmp1);
_tmp7 = *(_tmp6);
PushParam _tmp5;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*_tmp1 = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = _tmp3 + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = *( _tmp1);
_tmp7 = *( _tmp6);
PushParam _tmp5;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_temp0 = 4;
PushParam _temp0;
_temp1 = LCall _Alloc;
PopParams 4;
_temp2 = Object;
*(__temp1) = _temp2;
x = _temp1;
_temp3 = _temp0;
a = _temp3;
_temp4 = _temp3 + b;
c = _temp4;
_temp5 = c;
_temp6 = *(__temp1);
_temp7 = *(__temp6);
PushParam __temp5;
PushParam __temp1;
ACall __temp7;
PopParams 8;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(-tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = _tmp3 + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = *(tmp1);
_tmp7 = *(tmp6);
PushParam c;
PushParam tmp1;
ACall _tmp7;
PopParams 8;
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(__tmp1) = __tmp2;
x = __tmp1;
_tmp3 = __tmp0;
a = __tmp3;
_tmp4 = __tmp3 + b;
c = __tmp4;
_tmp5 = c;
_tmp6 = __tmp2;
_tmp7 = *(__tmp6);
PushParam c;
PushParam __tmp1;
ACall __tmp7;
PopParams 8;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*( _tmp1 ) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *( _tmp6 ) ;
PushParam c ;
PushParam _tmp4 ;
ACall _tmp7 ;
PopParams 8 ;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = _tmp3 + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = _tmp2;
_tmp7 = *(tmp2);
PushParam c;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_temp0 = 4;
PushParam _temp0;
_temp1 = LCall _Alloc;
PopParams 4;
_temp2 = Object;
*(temp1) = _temp2;
x = _temp1;
_temp3 = _temp0;
a = _temp3;
_temp4 = _temp3 + b;
c = _temp4;
_temp5 = c;
_temp6 = _temp2;
_temp7 = *(temp2);
PushParam c;
PushParam _temp1;
ACall _temp7;
PopParams 8;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_{tmp0} = 4;
PushParam _tmp0;
_{tmp1} = LCall _Alloc;
PopParams 4;
_{tmp2} = Object;
*(_tmp1) = _tmp2;
x = _tmp1;
_{tmp3} = _tmp0;
a = _tmp0;
_{tmp4} = _tmp0 + b;
c = _tmp4;
_{tmp5} = c;
_{tmp6} = _tmp2;
_{tmp7} = *(_tmp2);
PushParam c;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;
x = _tmp1;
_tmp3 = 4;
a = 4;
_tmp4 = _tmp0 + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = _tmp2;
_tmp7 = *(tmp2);
PushParam c;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;
x = _tmp1;
_tmp3 = 4;
a = 4;
_tmp4 = _tmp0 + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = _tmp2;
_tmp7 = *(tmp2);
PushParam c;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;
x = _tmp1;
_tmp3 = 4;
a = 4;
_tmp4 = _tmp0 + b;
c = _tmp4;
_tmp5 = _tmp4;
_tmp6 = _tmp2;
_tmp7 = *(tmp2);
PushParam _tmp4;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

\_tmp0 = 4 ;
PushParam \_tmp0 ;
\_tmp1 = LCall _Alloc ;
PopParams 4 ;
\_tmp2 = Object ;
\_tmp4 = \_tmp0 + b ;
c = \_tmp4 ;
\_tmp5 = \_tmp4 ;
\_tmp6 = \_tmp2 ;
\_tmp7 = *(_\_tmp2) ;
PushParam \_tmp4 ;
PushParam \_tmp1 ;
ACall \_tmp7 ;
PopParams 8 ;
Copy Propagation

• If we have a variable assignment
  \[ v_1 = v_2 \]
  then as long as \( v_1 \) and \( v_2 \) are not reassigned, we can rewrite expressions of the form
  \[ a = \ldots v_1 \ldots \]
  as
  \[ a = \ldots v_2 \ldots \]
  provided that such a rewrite is legal.

• This will help immensely later on, as you'll see.
Dead Code Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
Dead Code Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
Dead Code Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
Dead Code Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*( _tmp1 ) = _tmp2 ;

_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *( _tmp2 ) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
Dead Code Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(_tmp1) = _tmp2;

a = 4;
_tmp4 = _tmp0 + b;
c = _tmp4;
_tmp5 = _tmp4;
_tmp6 = _tmp2;
_tmp7 = *(__tmp2);
PushParam _tmp4;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Dead Code Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

Tmp0 = 4;
PushParam_Tmp0;
Tmp1 = LCall_Alloc;
PopParams 4;
Tmp2 = Object;
*(Tmp1) = Tmp2;
a = 4;
Tmp4 = Tmp0 + b;
c = Tmp4;
Tmp5 = Tmp4;
Tmp6 = Tmp2;
Tmp7 = *(Tmp2);
PushParam_Tmp4;
PushParam_Tmp1;
ACall_Tmp7;
PopParams 8;
Dead Code Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

\[
	\_\text{tmp0} = 4; \\
\text{PushParam } \_\text{tmp0}; \\
\_\text{tmp1} = \text{LCall } \_\text{Alloc}; \\
\text{PopParams} 4; \\
\_\text{tmp2} = \text{Object}; \\
*(_\text{tmp1}) = \_\text{tmp2}; \\
\]

\[
\_\text{tmp4} = \_\text{tmp0} + b; \\
c = \_\text{tmp4}; \\
\_\text{tmp5} = \_\text{tmp4}; \\
\_\text{tmp6} = \_\text{tmp2}; \\
\_\text{tmp7} = *(\_\text{tmp2}); \\
\text{PushParam } \_\text{tmp4}; \\
\text{PushParam } \_\text{tmp1}; \\
\text{ACall } \_\text{tmp7}; \\
\text{PopParams} 8; \\
\]
Dead Code Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;

_tmp4 = _tmp0 + b;
c = _tmp4;
_tmp5 = _tmp4;
_tmp6 = _tmp2;
_tmp7 = *(tmp2);
PushParam _tmp4;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Dead Code Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;

_tmp4 = _tmp0 + b;
_tmp5 = _tmp4;
_tmp6 = _tmp2;
_tmp7 = *(tmp2);
PushParam _tmp4;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Dead Code Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*( _tmp1 ) = _tmp2 ;

_tmp4 = _tmp0 + b ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *( _tmp2 ) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
Dead Code Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
Dead Code Elimination

Object x;
int a;
int b;
int c;
x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;

_tmp4 = _tmp0 + b;

_tmp6 = _tmp2;
_tmp7 = *(tmp2);
PushParam _tmp4;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Dead Code Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_	mp0 = 4;
PushParam _tmp0;
_	mp1 = LCall _Alloc;
PopParams 4;
_	mp2 = Object;
*( _tmp1 ) = _tmp2;
_	mp4 = _tmp0 + b;

_	mp7 = *( _tmp2 );
PushParam _tmp4;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Dead Code Elimination

• An assignment to a variable $v$ is called dead if the value of that assignment is never read anywhere.

• **Dead code elimination** removes dead assignments from IR.

• Determining whether an assignment is dead depends on what variable is being assigned to and when it's being assigned.
For Comparison

```plaintext
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*( _tmp1 ) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = a + b ;
_tmp6 = *( x ) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

```plaintext
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*( _tmp1 ) = _tmp2 ;
_tmp4 = _tmp0 + b ;
_tmp7 = *( _tmp2 ) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```
Applying Local Optimizations

- The different optimizations we've seen so far all take care of just a small piece of the optimization.
  - Common subexpression elimination eliminates unnecessary statements.
  - Copy propagation helps identify dead code.
  - Dead code elimination removes statements that are no longer needed.
- To get maximum effect, we may have to apply these optimizations numerous times.
Applying Local Optimizations

\[
\begin{align*}
  b &= a \times a; \\
  c &= a \times a; \\
  d &= b + c; \\
  e &= b + b;
\end{align*}
\]
Applying Local Optimizations

\[
\begin{align*}
    b &= a \times a; \\
    c &= a \times a; \\
    d &= b + c; \\
    e &= b + b;
\end{align*}
\]
Applying Local Optimizations

\[
b = a \times a;
c = a \times a;
d = b + c;
e = b + b;
\]

Common Subexpression Elimination
Applying Local Optimizations

```
b = a * a;
c = b;
d = b + c;
e = b + b;
```

Common Subexpression Elimination
Applying Local Optimizations

\[
\begin{align*}
  b &= a \times a; \\
  c &= b; \\
  d &= b + c; \\
  e &= b + b;
\end{align*}
\]
Applying Local Optimizations

```plaintext
b = a * a;
c = b;
d = b + c;
e = b + b;
```
Applying Local Optimizations

\[
\begin{align*}
\text{b} &= \text{a} \times \text{a}; \\
\text{c} &= \text{b} ; \\
\text{d} &= \text{b} + \text{c}; \\
\text{e} &= \text{b} + \text{b};
\end{align*}
\]

Copy Propagation
Applying Local Optimizations

\[
b = a \times a; \\
c = b; \\
d = b + b; \\
e = b + b;
\]

Copy Propagation
Applying Local Optimizations

\[ b = a \times a; \]
\[ c = b; \]
\[ d = b + b; \]
\[ e = b + b; \]
Applying Local Optimizations

\[
b = a \times a; \\
c = b; \\
d = b + b; \\
e = b + b;
\]
Applying Local Optimizations

\[
b = a \times a; \\
c = b; \\
d = b + b; \\
e = b + b;
\]

Common Subexpression Elimination (Again)
Applying Local Optimizations

\[
\begin{align*}
\ b &= a \times a; \\
\ c &= b; \\
\ d &= b + b; \\
\ e &= d;
\end{align*}
\]

Common Subexpression Elimination (Again)
Applying Local Optimizations

\[ b = a \times a; \]
\[ c = b; \]
\[ d = b + b; \]
\[ e = d; \]
Other Types of Local Optimization

• **Arithmetic Simplification**
  • Replace “hard” operations with easier ones.
  • e.g. rewrite \( x = 4 \times a; \) as \( x = a \ll 2; \)

• **Constant Folding**
  • Evaluate expressions at compile-time if they have a constant value.
  • e.g. rewrite \( x = 4 \times 5; \) as \( x = 20; \).
Implementing Local Optimization
Optimizations and Analyses

• Most optimizations are only possible given some analysis of the program's behavior.

• In order to implement an optimization, we will talk about the corresponding program analyses.
Available Expressions

- Both common subexpression elimination and copy propagation depend on an analysis of the available expressions in a program.
- An expression is called available if some variable in the program holds the value of that expression.
- In common subexpression elimination, we replace an available expression by the variable holding its value.
- In copy propagation, we replace the use of a variable by the available expression it holds.
Finding Available Expressions

- Initially, no expressions are available.
- Whenever we execute a statement $a = b + c$:
  - Any expression holding $a$ is invalidated.
  - The expression $a = b + c$ becomes available.
- **Idea**: Iterate across the basic block, beginning with the empty set of expressions and updating available expressions at each variable.
Available Expressions

a = b;

c = b;

d = a + b;

e = a + b;

d = b;

f = a + b;
Available Expressions

\[
\begin{align*}
\{ & \\
\{ & \\
a &= b; \\
c &= b; \\
d &= a + b; \\
e &= a + b; \\
d &= b; \\
f &= a + b;
\end{align*}
\]
Available Expressions

{   }
a = b;
{   a = b   }
c = b;
d = a + b;
e = a + b;
d = b;
f = a + b;
Available Expressions

{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
e = a + b;
d = b;
f = a + b;
Available Expressions

```
{ }  
    a = b;
   { a = b }  
    c = b;
  { a = b, c = b }  
    d = a + b;
{ a = b, c = b, d = a + b }  
    e = a + b;

    d = b;

    f = a + b;
```
Available Expressions

```
{ }
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;

f = a + b;
```
Available Expressions

{ }

a = b;

{ a = b }

c = b;

{ a = b, c = b }

d = a + b;

{ a = b, c = b, d = a + b }

e = a + b;

{ a = b, c = b, d = a + b, e = a + b }

d = b;

{ a = b, c = b, d = b, e = a + b }

f = a + b;
Available Expressions

```
{ }
{ a = b }
{ a = b, c = b }
{ a = b, c = b, d = a + b }
{ a = b, c = b, d = a + b, e = a + b }
{ a = b, c = b, d = b, e = a + b }
{ a = b, c = b, d = b, e = a + b, f = a + b }
```
Common Subexpression Elimination

{ }
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
Common Subexpression Elimination

{ }
a = b;
{ a = b } c = b;
{ a = b, c = b } d = a + b;
{ a = b, c = b, d = a + b } e = a + b;
{ a = b, c = b, d = a + b, e = a + b } d = b;
{ a = b, c = b, d = b, e = a + b } f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
Common Subexpression Elimination

```plaintext
{ }
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```
Common Subexpression Elimination

{ }
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }

{ a = b, c = b, d = b, e = a + b, f = a + b }
Common Subexpression Elimination

```
{   }
 a = b;
{   a = b   }
c = a;
{   a = b, c = b   }
d = a + b;
{   a = b, c = b, d = a + b   }
e = d;
{   a = b, c = b, d = a + b, e = a + b   }
d = b;
{   a = b, c = b, d = b, e = a + b   }
f = a + b;
{   a = b, c = b, d = b, e = a + b, f = a + b   }
```
Common Subexpression Elimination

```
{ }
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = d;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```
Common Subexpression Elimination

```plaintext
{   }
  a = b;
{   a = b   }
  c = a;
{   a = b, c = b   }
  d = a + b;
{   a = b, c = b, d = a + b   }
  e = d;
{   a = b, c = b, d = a + b, e = a + b   }
  d = a;
{   a = b, c = b, d = b, e = a + b   }
  f = a + b;
{   a = b, c = b, d = b, e = a + b, f = a + b   }
```
Common Subexpression Elimination

{ }
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = d;
{ a = b, c = b, d = a + b, e = a + b }
d = a;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
Common Subexpression Elimination

```
{ }
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = d;
{ a = b, c = b, d = a + b, e = a + b }
d = a;
{ a = b, c = b, d = b, e = a + b }
f = e;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```
Common Subexpression Elimination

\[ a = b; \]
\[ c = a; \]
\[ d = a + b; \]
\[ e = d; \]
\[ d = a; \]
\[ f = e; \]
Live Variables

• The analysis corresponding to dead code elimination is called **liveness analysis**.

• A variable is **live** at a point in a program if later in the program its value will be read before it is written to again.

• Dead code elimination works by computing liveness for each variable, then eliminating assignments to dead variables.
Computing Live Variables

• To know if a variable will be used at some point, we iterate across the statements in a basic block in reverse order.

• Initially, some small set of values are known to be live (which ones depends on the particular program).

• When we see the statement $a = b + c$:
  
  • Just before the statement, $a$ is not alive, since its value is about to be overwritten.
  
  • Just before the statement, both $b$ and $c$ are alive, since we're about to read their values.
  
  • *(what if we have $a = a + b$?)*
Liveness Analysis

\[ a = b; \]
\[ c = a; \]
\[ d = a + b; \]
\[ e = d; \]
\[ d = a; \]
\[ f = e; \]
Liveness Analysis

\[
\begin{align*}
  a &= b; \\
  c &= a; \\
  d &= a + b; \\
  e &= d; \\
  d &= a; \\
  f &= e; \\
  \{ b, d \}
\end{align*}
\]
Liveness Analysis

a = b;
c = a;
d = a + b;
e = d;
d = a;
{ b, d, e }
f = e;
{ b, d }
Liveness Analysis

\[
a = b; \\
c = a; \\
d = a + b; \\
e = d; \\
\{ \ a, b, e \ \} \\
d = a; \\
\{ \ b, d, e \ \} \\
f = e; \\
\{ \ b, d \ \}
\]
Liveness Analysis

\[
\begin{align*}
  a &= b; \\
  c &= a; \\
  d &= a + b; \\
  \{ a, b, d \} \\
  e &= d; \\
  \{ a, b, e \} \\
  d &= a; \\
  \{ b, d, e \} \\
  f &= e; \\
  \{ b, d \}
\end{align*}
\]
Liveness Analysis

\begin{align*}
a &= b; \\
c &= a; \\
\{ a, b \} \\
d &= a + b; \\
\{ a, b, d \} \\
e &= d; \\
\{ a, b, e \} \\
d &= a; \\
\{ b, d, e \} \\
f &= e; \\
\{ b, d \}
\end{align*}
Liveness Analysis

\[
\begin{align*}
a &= b; \\
\{ a, b \} \\
c &= a; \\
\{ a, b \} \\
d &= a + b; \\
\{ a, b, d \} \\
e &= d; \\
\{ a, b, e \} \\
d &= a; \\
\{ b, d, e \} \\
f &= e; \\
\{ b, d \}
\end{align*}
\]
Liveness Analysis

{ b }
\[
a = b;
{ a, b }
\]
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
Dead Code Elimination

```
{ b }  
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```
Dead Code Elimination

{ b }
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
Dead Code Elimination

```plaintext
{ b }
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
{ b, d }
```
Dead Code Elimination

```plaintext
{ b }
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }

{ b, d }
```
Dead Code Elimination

{ b }  
a = b;  
{ a, b }  

{ a, b }  
d = a + b;  
{ a, b, d }  
e = d;  
{ a, b, e }  
d = a;  
{ b, d, e }  

{ b, d }
Dead Code Elimination

\[
\begin{align*}
    a &= b; \\
    d &= a + b; \\
    e &= d; \\
    d &= a; 
\end{align*}
\]
Liveness Analysis II

\[
\begin{align*}
a &= b; \\
d &= a + b; \\
e &= d; \\
d &= a;
\end{align*}
\]
Liveness Analysis II

\[ a = b; \]

\[ d = a + b; \]

\[ e = d; \]

\[ d = a; \]

\{ b, d \}
Liveness Analysis II

\[
a = b;
\]

\[
d = a + b;
\]

\[
e = d;
\]

\[
\{ a, b \}
\]

\[
d = a;
\]

\[
\{ b, d \}
\]
Liveness Analysis II

\[
a = b;
\]

\[
d = a + b;
\]

\[
\{ a, b, d \}
\]

\[
e = d;
\]

\[
\{ a, b \}
\]

\[
d = a;
\]

\[
\{ b, d \}
\]
Liveness Analysis II

```
a = b;

{ a, b }

d = a + b;
{ a, b, d }
e = d;
{ a, b }
d = a;
{ b, d }
```
Liveness Analysis II

\{ b \}
a = b;

\{ a, b \}

d = a + b;
\{ a, b, d \}

e = d;
\{ a, b \}

d = a;
\{ b, d \}
Dead Code Elimination

{ b }
a = b;

{ a, b }
d = a + b;

{ a, b, d }
e = d;

{ a, b }
d = a;

{ b, d }
Dead Code Elimination

{ b }
\[ a = b; \]

{ a, b }
\[ d = a + b; \]
\[ \{ a, b, d \} \]
\[ e = d; \]
\[ \{ a, b \} \]
\[ d = a; \]
\[ \{ b, d \} \]
Dead Code Elimination

\{ \texttt{b} \}
\texttt{a} = \texttt{b};

\{ \texttt{a, b} \}
\texttt{d} = \texttt{a} + \texttt{b};
\{ \texttt{a, b, d} \}

\{ \texttt{a, b} \}
\texttt{d} = \texttt{a};
\{ \texttt{b, d} \}
Dead Code Elimination

\[
a = b;
\]

\[
d = a + b;
\]

\[
d = a;
\]
Liveness Analysis III

\[ a = b; \]

\[ d = a + b; \]

\[ d = a; \]
Liveness Analysis III

\[ a = b; \]
\[ d = a + b; \]
\[ d = a; \]
\[ \{b, d\} \]
Liveness Analysis III

\[ a = b; \]

\[ d = a + b; \]

\[ \{a, \ b\} \]

\[ d = a; \]

\[ \{b, \ d\} \]
Liveness Analysis III

\[ a = b; \]

\{a, b\}

\[ d = a + b; \]

\{a, b\}

\[ d = a; \]

\{b, d\}
Liveness Analysis III

\{b\}
a = b;

\{a, b\}

d = a + b;

\{a, b\}

d = a;

\{b, d\}
Dead Code Elimination

\[
\begin{align*}
\{b\} \\
a &= b; \\
\{a, b\} \\
d &= a + b; \\
\{a, b\} \\
d &= a; \\
\{b, d\}
\end{align*}
\]
Dead Code Elimination

\[
\begin{align*}
\{b\} \\
a &= b; \\
\{a, b\} \\
d &= a + b; \\
\{a, b\} \\
d &= a; \\
\{b, d\}
\end{align*}
\]
Dead Code Elimination

```plaintext
{\text{b}}
a = b;

{\text{a, b}}

d = a;
{\text{b, d}}
```
Dead Code Elimination

```plaintext
a = b;
d = a;
```
A Combined Algorithm
A Combined Algorithm

\[
\begin{align*}
a &= b; \\
c &= a; \\
d &= a + b; \\
e &= d; \\
d &= a; \\
f &= e;
\end{align*}
\]
A Combined Algorithm

\[
\begin{align*}
a & = b; \\
c & = a; \\
d & = a + b; \\
e & = d; \\
d & = a; \\
f & = e; \\
\{b, d\}
\end{align*}
\]
A Combined Algorithm

\[a = b;\]
\[c = a;\]
\[d = a + b;\]
\[e = d;\]
\[d = a;\]
\[f = e;\]
\[\{b, d\}\]
A Combined Algorithm

\[
\begin{align*}
  a &= b; \\
  c &= a; \\
  d &= a + b; \\
  e &= d; \\
  d &= a; \\
  \{b, d\}
\end{align*}
\]
A Combined Algorithm

\[
\begin{align*}
  a &= b; \\
  c &= a; \\
  d &= a + b; \\
  e &= d; \\
  \{a, b\} &\quad \{a, b\} \\
  d &= a; \\
  \{b, d\} &\quad \{b, d\}
\end{align*}
\]
A Combined Algorithm

\[
a = b;
\]
\[
c = a;
\]
\[
d = a + b;
\]
\[
e = d;
\]
\[
\{a, b\}
\]
\[
d = a;
\]
\[
\{b, d\}
\]
A Combined Algorithm

\[
a = b;
\]
\[
c = a;
\]
\[
d = a + b;
\]
\[
\{a, b\}
\]
\[
d = a;
\]
\[
\{b, d\}
\]
A Combined Algorithm

\[ a = b; \]
\[ c = a; \]
\[ d = a + b; \]
\[ \{a, b\} \]
\[ d = a; \]
\[ \{b, d\} \]
A Combined Algorithm

\[
\begin{align*}
a &= b; \\
c &= a; \\
\{a, b\} \\
d &= a; \\
\{b, d\}
\end{align*}
\]
A Combined Algorithm

a = b;

c = a;

\{a, b\}

d = a;

\{b, d\}
A Combined Algorithm

\[ a = b; \]
\[ \{a, b\} \]
\[ d = a; \]
\[ \{b, d\} \]
A Combined Algorithm

{b}
\[ a = b; \]

{a, b}
\[ d = a; \]

{b, d}
A Combined Algorithm

\[ a = b; \]

\[ d = a; \]
Properties of Local Analysis

- The only way to find out what a program will actually do is to run it.

- Problems:
  - The program might not terminate.
  - The program might have some behavior we didn't see when we ran it on a particular input.

- However, this is not a problem inside a basic block.
  - Basic blocks contain no loops.
  - There is only one path through the basic block.
Another View of Local Optimization

• In local optimization, we want to reason about some property of the runtime behavior of the program.

• We know that we can run the code in a basic block and guarantee termination.

• Could we run the program and just watch what happens?

• **Idea**: Redefine the semantics of our programming language to give us information about our analysis.
Assigning New Semantics

• Example: Available Expressions
• Redefine the statement \( a = b + c \) to mean “\( a \) now holds the value of \( b + c \), and any variable holding the value \( a \) is now invalid.”

• Run the program assuming these new semantics.
• Treat the compiler as an interpreter for these new semantics.
Next Time

• **Global optimization**
  • Optimizing across basic blocks.
  • Meet operators and the dataflow framework.
Written Assignment 2 Scores

Mean: 86 / 120
Median: 93.5 / 120
Stdev: 23