IR Optimization

May 15th, 2013
But before we talk about IR optimization...

• Wrapping up IR generation
Three-Address Code

- Or "**TAC**"
- The IR that you will be using for the final programming project.
- High-level assembly where each operation has at most three operands.
- Uses explicit runtime stack for function calls.
- Uses vtables for dynamic dispatch.
Generating TAC
TAC Generation

• At this stage in compilation, we have
  • an AST,
  • annotated with scope information,
  • and annotated with type information.

• To generate TAC for the program, we do (yet another) recursive tree traversal!
  • Generate TAC for any subexpressions or substatements.
  • Using the result, generate TAC for the overall expression.
TAC Generation for Expressions

- Define a function \texttt{cgen(\textit{expr})} that generates TAC that computes an expression, stores it in a temporary variable, then hands back the name of that temporary.

- Define \texttt{cgen} directly for atomic expressions (constants, \texttt{this}, identifiers, etc.).

- Define \texttt{cgen} recursively for compound expressions (binary operators, function calls, etc.).
cgen for Basic Expressions
cgen for Basic Expressions

cgen(k) = { // k is a constant
    Choose a new temporary t
    Emit( t = k );
    Return t
}

cgen for Basic Expressions

\[
cgen(k) = \{ \quad // \; k \; \text{is a constant} \\
\text{Choose a new temporary } t \\
\text{Emit}(t = k); \\
\text{Return } t
\}
\]

\[
cgen(id) = \{ \quad // \; id \; \text{is an identifier} \\
\text{Choose a new temporary } t \\
\text{Emit}(t = id) \\
\text{Return } t
\}
\]
cgen for Binary Operators
cgen for Binary Operators

cgen(e_1 + e_2) = {
  Choose a new temporary t
  Let t_1 = cgen(e_1)
  Let t_2 = cgen(e_2)
  Emit( t = t_1 + t_2 )
  Return t
}

**An Example**

\[ cgen(5 + x) = \{
  \text{Choose a new temporary } t \\
  \text{Let } t_1 = cgen(5) \\
  \text{Let } t_2 = cgen(x) \\
  \text{Emit } (t = t_1 + t_2) \\
  \text{Return } t \\
\} \]
An Example

cgen(5 + x) = {
    Choose a new temporary t
    Let t_1 = {
        Choose a new temporary t
        Emit( t = 5 )
        return t
    }
    Let t_2 = cgen(x)
    Emit (t = t_1 + t_2)
    Return t
}
An Example

cgen(5 + x) = {
    Choose a new temporary $t$
    Let $t_1 =$ {
        Choose a new temporary $t$
        Emit( $t = 5$ )
        return $t$
    }
    Let $t_2 =$ {
        Choose a new temporary $t$
        Emit( $t = x$ )
        return $t$
    }
    Emit (t = $t_1 + t_2$)
    Return $t$
}
An Example

cgen(5 + x) = {
    Choose a new temporary t
    Let t₁ = {
        Choose a new temporary t
        Emit( t = 5 )
        return t
    }

    Let t₂ = {
        Choose a new temporary t
        Emit( t = x )
        return t
    }

    Emit (t = t₁ + t₂)
    Return t
}
cgen for Statements

• We can extend the cgen function to operate over statements as well.

• Unlike cgen for expressions, cgen for statements does not return the name of a temporary holding a value.
  • (Why?)
cgen for Simple Statements
cgen for Simple Statements

\[
cgen(expr ;) = \{ \\
    \quad \text{cgen}(expr) \\
\} 
\]
cgen for while loops
cgen for while loops

cgen(while (expr) stmt) = {

Let L before be a new label.
Let L after be a new label.
Emit( L before : )
Let t = cgen( expr )
Emit( IfZ t Goto L after )
cgen( stmt )
Emit( Goto L before )
Emit( L after : )

}
cgen for while loops

cgen(while (expr) stmt) = {
    Let \(L_{\text{before}}\) be a new label.
    Let \(L_{\text{after}}\) be a new label.
}

cgen for while loops

cgen(while (expr) stmt) = {
    Let L_{before} be a new label.
    Let L_{after} be a new label.
    Emit( L_{before} : )
    cgen(expr) stmt
    Emit( L_{after} : )
}
\textbf{cgen} for \textbf{while} loops

\texttt{cgen(while (expr) stmt) = \{  
  Let L_{before} be a new label.  
  Let L_{after} be a new label.  
  \texttt{Emit( L_{before} : )}  
  Let t = \texttt{cgen(expr)}  
  \texttt{Emit( IfZ t Goto L_{after} )}  
  \texttt{Emit( L_{after} : )}  
\}}
cgen for while loops

cgen(while (expr) stmt) = {
    Let $L_{before}$ be a new label.
    Let $L_{after}$ be a new label.
    Emit( $L_{before}$ : )
    Let $t = \text{cgen}(expr)$
    Emit( IfZ $t$ Goto $L_{after}$ )
    cgen(stmt)

    Emit( $L_{after}$ : )
}

cgen for while loops

cgen(while (expr) stmt) = {
    Let $L_{before}$ be a new label.
    Let $L_{after}$ be a new label.
    Emit( $L_{before}$ : )
    Let $t = \text{cgen}(expr)$
    Emit( IfZ $t$ Goto $L_{after}$ )
    cgen(stmt)
    Emit( Goto $L_{before}$ )
    Emit( $L_{after}$ : )
}

Where We Are

Where We Are

Source Code → Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation

Achievement unlocked
Wond-IR-ful!

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.
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

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

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```
int x;
int y;
bool b1;
bool b2;
bool b3;

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

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

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

_t4 = x + x;
_t5 = y;
b3 = _t5 < _t4;
int x;
int y;
bool b1;
bool b2;
bool b3;

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

_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;
}

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

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

\_t0 = y + z;
\_L0:
\_t1 = x < \_t0;
IfZ \_t1 Goto \_L1;
x = x - y;
Goto \_L0;
\_L1:
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 \text{NP}-hard or undecidable.
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?
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 $x * 0.5$ with $x / 2$
Optimization Overview

- Basic Concept
  - Control Flow Graphs
  - Basic blocks
- Local Optimization
- Global Optimization
- Advanced Topics
  - Data flow Analysis
  - Microarchitectural dependent optimizations
    - Register Allocation
    - Instruction selection
    - Instruction scheduling/pipelining
    - loop optimization
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;
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;
}
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;
}

Local Optimizations
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;

_t2 = y;
x = _t2;

end
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
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;
}

Global Optimizations

y = 137;
IfZ x Goto _L0;

z = y;

x = y;

end
Global Optimizations

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

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

```
start

y = 137;  
IfZ x Goto _L0;

z = y;  
x = y;

end
```
Global 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;
}

Global Optimizations

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);

_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 = 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;
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 ;
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;
ext x = new Object;
int a;
int b;
int c;
int 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

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

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

_tip0 = 4;
PushParam _tmp0;
-tip1 = LCall _Alloc;
PopParams 4;
-tip2 = Object;
*(tip1) = _tmp2;
x = _tmp1;
-tip3 = _tmp0;
a = _tmp3;
-tip4 = a + b;
c = _tmp4;
-tip5 = c;
-tip6 = *(x);
-tip7 = *(tip6);
PushParam _tmp5;
PushParam x;
ACall _tmp7;
PopParams 8;
Common Subexpression Elimination

- If we have two variable assignments
  \[ v_1 = a \text{ op } b \]
  
  \[ v_2 = a \text{ op } b \]
  
  and the values of \( v_1 \), \( a \), and \( b \) have not changed between the assignments, rewrite the code as
  
  \[ v_1 = a \text{ op } b \]
  
  \[ v_2 = v_1 \]

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

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

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

```plaintext
_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);
Copy Propagation

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

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

_handler = _tmp0 = 4;
PushParam _tmp0;
_handler = _tmp1 = LCall _Alloc;
PopParams 4;
_handler = _tmp2 = Object;
*( _tmp1) = _tmp2;
_handler = x = _tmp1;
_handler = _tmp3 = _tmp0;
a = _tmp3;
_handler = _tmp4 = a + b;
c = _tmp4;
_handler = _tmp5 = c;
_handler = _tmp6 = *( _tmp1);
_handler = _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;
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 _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 = _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);

_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;
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;
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 _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 = *( _temp6 ) ;
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 = _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);

_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);

_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;
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;
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;
*(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

• 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);

_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;
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 ;
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 ;
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 \( c \);

\( x = \text{new Object} ; \)
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;
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;
_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;
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);

_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;
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.
_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 ;
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

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

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

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

Common Subexpression Elimination
Applying Local Optimizations

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

Common Subexpression Elimination
Applying Local Optimizations

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

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

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

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

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

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

Common Subexpression Elimination (Again)
Applying Local Optimizations

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

Common Subexpression Elimination (Again)
Applying Local Optimizations

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

Intuition: determines for each point in the program the set of expressions that need not be recomputed. Those expressions are said to be available at such a point.

To be available on a program point, the operands of the expression should not be modified on any path from the occurrence of that expression to the program point.
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;
{ 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 = 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 = b, e = 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

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 }
Common Subexpression Elimination

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

{ }
{ 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

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

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

```plaintext
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

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

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 }
Liveness Analysis

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 }
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 }

{ 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

```c
{ 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

{ 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

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

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

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

\[
a = b;
\]

\[
d = a + b;
\]

\[
e = d;
\]

\[
d = a;
\]

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

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

\[
\begin{align*}
\{ \ b \ \} \\
a &= b;
\end{align*}
\]

\[
\begin{align*}
\{ \ a, \ b \ \}
\end{align*}
\]

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

\[
\begin{align*}
e &= d;
\end{align*}
\]

\[
\begin{align*}
\{ \ a, \ b \ \}
\end{align*}
\]

\[
\begin{align*}
d &= a;
\end{align*}
\]

\[
\begin{align*}
\{ \ b, \ d \ \}
\end{align*}
\]
Dead Code Elimination

{ b }
a = b;

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

{ a, b }
d = a;
{ 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

{b}
a = b;

{a, b}

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

\[ a = b; \]

\[ d = a; \]
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

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

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

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

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

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

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

\begin{align*}
  \textcolor{red}{\{a, b\}} & \\
  \textcolor{red}{d = a;} & \\
  \textcolor{red}{\{b, d\}} & \\
\end{align*}
A Combined Algorithm

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

\[ a = b; \]

\[ d = a; \]
Next Time

- **Formalisms for Local Optimizations**
  - Transfer functions and semantics
- **Global optimization**
  - Optimizing across basic blocks.
  - Meet operators and the dataflow framework.