SSA-based Optimizations

CSE 501
Lecture 4
April 10, 2013
Questions?

• Earlier lectures?
• Project?
• Reading?
Data-flow analysis

For many years, people approached optimization by writing and solving sets of data-flow equations, then using the results to guide transformations.

Entire optimizers were based on iterative bit-vector solvers.

Last lecture, I showed some examples where this approach didn't work as well as we might like.

This time, we'll look at some alternatives.
DU and UD chains

One scheme is to write optimizations based on Def-Use and Use-Def chains
• DU chains are links from definitions to uses
• UD chains are links from uses back to definitions

Typically, an IR statement will have a set of uses for every output variable and a set of definitions for every input.

Computed via DFA.
DU and UD chains

Consider the UD chains in this example:

\[
\begin{align*}
    UD(3, i) &= \{ 2 \} \\
    UD(3, n) &= \{ 1 \} \\
    UD(4, p) &= \{ 1, 4 \} \\
    UD(4, a) &= \{ 1 \} \\
    UD(5, x) &= \{ 1, 5 \} \\
    UD(5, a) &= \{ 1 \} \\
    UD(6, i) &= \{ 2, 6 \} \\
    UD(7, i) &= \{ 6 \} \\
    UD(7, n) &= \{ 1 \} \\
    UD(8, p) &= \{ 1, 4 \}
\end{align*}
\]

The DU chains will be similar.
Dead code elimination

Mark critical instructions, then chase the UD chains recursively.

\[
\begin{align*}
UD(3, i) &= \{ 2 \} \\
UD(3, n) &= \{ 1 \} \\
UD(4, p) &= \{ 1, 4 \} \\
UD(4, a) &= \{ 1 \} \\
UD(5, x) &= \{ 1, 5 \} \\
UD(5, a) &= \{ 1 \} \\
UD(6, i) &= \{ 2, 6 \} \\
UD(7, i) &= \{ 6 \} \\
UD(7, n) &= \{ 1 \} \\
UD(8, p) &= \{ 1, 4 \}
\end{align*}
\]

An easy worklist algorithm.

```
1 enter(p, x, n, a)
2 i ← 1
3 i < n
4 p ← p + a
5 x ← x × a
6 i ← i + 1
7 i < n
8 return p
```
Dead code elimination

UD(3, i) = { 2 }
UD(3, n) = { 1 }
UD(4, p) = { 1, 4 }
UD(4, a) = { 1 }
UD(5, x) = { 1, 5 }
UD(5, a) = { 1 }
UD(6, i) = { 2, 6 }
UD(7, i) = { 6 }
UD(7, n) = { 1 }
UD(8, p) = { 1, 4 }

worklist = { 3, 7, 8 }

1 enter(p, x, n, a)
2 i ← 1
3 i < n
4 p ← p + a
5 x ← x × a
6 i ← i + 1
7 i < n
8 return p
Dead code elimination

UD(3, i) = { 2 }
UD(3, n) = { 1 }
UD(4, p) = { 1, 4 }
UD(4, a) = { 1 }
UD(5, x) = { 1, 5 }
UD(5, a) = { 1 }
UD(6, i) = { 2, 6 }
UD(7, i) = { 6 }
UD(7, n) = { 1 }
UD(8, p) = { 1, 4 }

worklist = { 1, 2, 7, 8 }

1. enter(p, x, n, a)
2. \( i \leftarrow 1 \)
3. \( i < n \)
4. \( p \leftarrow p + a \)
5. \( x \leftarrow x \times a \)
6. \( i \leftarrow i + 1 \)
7. \( i < n \)
8. return \( p \)
Dead code elimination

UD(3, i) = \{ 2 \}
UD(3, n) = \{ 1 \}
UD(4, p) = \{ 1, 4 \}
UD(4, a) = \{ 1 \}
UD(5, x) = \{ 1, 5 \}
UD(5, a) = \{ 1 \}
UD(6, i) = \{ 2, 6 \}
UD(7, i) = \{ 6 \}
UD(7, n) = \{ 1 \}
UD(8, p) = \{ 1, 4 \}

worklist = \{ 2, 7, 8 \}
Dead code elimination

UD(3, i) = { 2 }
UD(3, n) = { 1 }
UD(4, p) = { 1, 4 }
UD(4, a) = { 1 }
UD(5, x) = { 1, 5 }
UD(5, a) = { 1 }
UD(6, i) = { 2, 6 }
UD(7, i) = { 6 }
UD(7, n) = { 1 }
UD(8, p) = { 1, 4 }

worklist = { 7, 8 }

1 enter(p, x, n, a)
2 i ← 1
3 i < n
4 p ← p + a
5 x ← x × a
6 i ← i + 1
7 i < n
8 return p
Dead code elimination

\[
\begin{align*}
UD(3, i) &= \{ 2 \} \\
UD(3, n) &= \{ 1 \} \\
UD(4, p) &= \{ 1, 4 \} \\
UD(4, a) &= \{ 1 \} \\
UD(5, x) &= \{ 1, 5 \} \\
UD(5, a) &= \{ 1 \} \\
UD(6, i) &= \{ 2, 6 \} \\
UD(7, i) &= \{ 6 \} \\
UD(7, n) &= \{ 1 \} \\
UD(8, p) &= \{ 1, 4 \}
\end{align*}
\]

worklist = \{ 6, 8 \}
Dead code elimination

\[
\begin{align*}
\text{UD(3, } i \text{)} &= \{ 2 \} \\
\text{UD(3, } n \text{)} &= \{ 1 \} \\
\text{UD(4, } p \text{)} &= \{ 1, 4 \} \\
\text{UD(4, } a \text{)} &= \{ 1 \} \\
\text{UD(5, } x \text{)} &= \{ 1, 5 \} \\
\text{UD(5, } a \text{)} &= \{ 1 \} \\
\text{UD(6, } i \text{)} &= \{ 2, 6 \} \\
\text{UD(7, } i \text{)} &= \{ 6 \} \\
\text{UD(7, } n \text{)} &= \{ 1 \} \\
\text{UD(8, } p \text{)} &= \{ 1, 4 \}
\end{align*}
\]

worklist = \{ 8 \}
Dead code elimination

UD(3, i) = { 2 }
UD(3, n) = { 1 }
UD(4, p) = { 1, 4 }
UD(4, a) = { 1 }
UD(5, x) = { 1, 5 }
UD(5, a) = { 1 }
UD(6, i) = { 2, 6 }
UD(7, i) = { 6 }
UD(7, n) = { 1 }
UD(8, p) = { 1, 4 }

worklist = { 4 }

1. enter(p, x, n, a)
2. i ← 1
3. i < n
4. p ← p + a
5. x ← x × a
6. i ← i + 1
7. i < n
8. return p
Dead code elimination

\[
\begin{align*}
UD(3, i) &= \{ 2 \} \\
UD(3, n) &= \{ 1 \} \\
UD(4, p) &= \{ 1, 4 \} \\
UD(4, a) &= \{ 1 \} \\
UD(5, x) &= \{ 1, 5 \} \\
UD(5, a) &= \{ 1 \} \\
UD(6, i) &= \{ 2, 6 \} \\
UD(7, i) &= \{ 6 \} \\
UD(7, n) &= \{ 1 \} \\
UD(8, p) &= \{ 1, 4 \}
\end{align*}
\]

worklist = \{ \}

1. enter(p, x, n, a)
2. \( i \leftarrow 1 \)
3. \( i < n \)
4. \( p \leftarrow p + a \)
5. \( x \leftarrow x \times a \)
6. \( i \leftarrow i + 1 \)
7. \( i < n \)
8. \text{return } p
Dead code elimination

Now delete all unmarked, *useless*, instructions.

Note the analogy to mark-sweep garbage collection.

This approach is sometimes better and never worse than the liveness-based scheme from the last lecture.
Static single assignment form

In the mid '80s, researchers at IBM introduced the SSA form.

Every variable use is reached by exactly one definition.

Phi functions (phony functions) are introduced to make it work.

UD chains become trivial to represent and faster to compute.
Static single assignment form

Don't be distracted by the numbered variables; we could just as easily rename each variable uniquely.

```
enter(a, b, c, d)
e ← 1
e < c

f ← φ(a, i)
g ← φ(e, k)
h ← φ(b, j)
i ← f + d
j ← h × d
k ← g + 1
k < c

dl ← φ(a, i)
return l
```
Dead code elimination

DCE is easily adapted to SSA form. We begin by marking critical instructions and adding their operands to a worklist.

worklist = { c, e, k, l }
Dead code elimination

```
enter(a, b, c, d)
    e ← 1
    e < c
```

```
f ← \phi(a, i)
g ← \phi(e, k)
h ← \phi(b, j)
i ← f + d
j ← h \times d
k ← g + 1
k < c
```

```
l ← \phi(a, i)
return l
```

worklist = \{ e, k, l \}
Dead code elimination

worklist = \{ k, l \}
Dead code elimination

```
enter(a, b, c, d)
  e ← 1
  e < c

f ← φ(a, i)
g ← φ(e, k)
h ← φ(b, j)
i ← f + d
j ← h × d
k ← g + 1
k < c

l ← φ(a, i)
return l
```

worklist = \{ g, l \}
Dead code elimination

```latex
\begin{align*}
\text{enter}(a, b, c, d) & \\
   e & \leftarrow 1 \\
   e & < c \\
\end{align*}
```

```latex
\begin{align*}
   f & \leftarrow \phi(a, i) \\
   g & \leftarrow \phi(e, k) \\
   h & \leftarrow \phi(b, j) \\
   i & \leftarrow f + d \\
   j & \leftarrow h \times d \\
   k & \leftarrow g + 1 \\
   k & < c \\
\end{align*}
```

```latex
l & \leftarrow \phi(a, i) \\
\text{return } l
```

worklist = \{ l \}
Dead code elimination

worklist = \{ a, i \}
Dead code elimination

```plaintext
enter(a, b, c, d)
  e ← 1
  e < c

f ← φ(a, i)
g ← φ(e, k)
h ← φ(b, j)
i ← f + d
j ← h × d
k ← g + 1
k < c

worklist = { i }

l ← φ(a, i)
return l
```
Dead code elimination

worklist = {d, f}

```plaintext
enter(a, b, c, d)
e ← 1

e < c

f ← \phi(a, i)
g ← \phi(e, k)
h ← \phi(b, j)
i ← f + d
j ← h \times d
k ← g + 1
k < c

l ← \phi(a, i)
return l
```
Dead code elimination

```plaintext
worklist = \{ f \}
```

```plaintext
enter(a, b, c, d)
e ← 1
e < c
```

```plaintext
f ← φ(a, i)
g ← φ(e, k)
h ← φ(b, j)
i ← f + d
j ← h × d
k ← g + 1
k < c
```

```plaintext
l ← φ(a, i)
return l
```
Dead code elimination

\[
\text{enter}(a, b, c, d) \\
e \leftarrow 1 \\
e < c
\]

\[
f \leftarrow \phi(a, i) \\
g \leftarrow \phi(e, k) \\
h \leftarrow \phi(b, j) \\
i \leftarrow f + d \\
j \leftarrow h \times d \\
k \leftarrow g + 1 \\
k < c
\]

worklist = \{a, i\}
Dead code elimination

enter(a, b, c, d)
  e ← 1
  e < c

worklist = { i }

l ← φ(a, i)
  return l
Dead code elimination

When the worklist is empty, we delete the useless instructions.

\[
\text{worklist} = \{ \}
\]
Factored UD chains

SSA can be viewed as factored UD chains, saving space compared to ordinary UD chains. Compare
Factored UD chains

SSA can be viewed as factored UD chains, saving space compared to ordinary UD chains. Compare

\[
\begin{align*}
  a &\leftarrow \\
  b &\leftarrow \\
  c &\leftarrow \\
  d &\leftarrow \\
  e &\leftarrow \phi(a, b, c, d) \\
  \cdots &\leftarrow e
\end{align*}
\]
Constant propagation

Cprop is a useful optimization with some interesting effects:
- Can propagate constant values through copy instructions,
- Can fold expressions where all the operands are known constants,
- Can account for things like $0 \times x$, and
- Can account for conditional branches with constant operands.

Particularly effective as a cleanup after inlining and certain loops transformations (blocking, vectorization, parallelization).

Can be extended to propagate the values of individual bits.
Examples

\[ i = 1 + 2*4; \]

\[ j = 1; \]
\[ k = 2; \]
\[ m = 4; \]
\[ n = k*m; \]
\[ j = n + j; \]

\[ i = 10; \]
\[ k = 20; \]
\[ n = 30; \]
\[ \text{return } i + j + k + m + n; \]

\[ p = \text{mumble}; \]
\[ i = 0; \]
\[ \text{do} \{ \]
\[ \quad p = p*i; \]
\[ \quad i++; \]
\[ \} \text{ while } (p > 0); \]
\[ \text{return } x/i; \]

\[ \text{if } (x) \{ \]
\[ \quad j = 4; \]
\[ \quad k = 6; \]
\[ \} \]
\[ \text{else } \{ \]
\[ \quad j = 7; \]
\[ \quad k = 3; \]
\[ \} \]
\[ \text{return } (j + k)*21; \]
A lattice

As an aid to reasoning, we define a lattice:

While the lattice is wide, there are only three levels:
1) Values that may yet prove be constant are marked $\top$ (top),
2) Values that are known to be constant are marked with their value, and
3) Values that we're unable to prove are constant are marked $\bot$ (bottom).

During cprop, we optimistically initialize each SSA value to $\top$, then lower them to their final value as the algorithm progresses.
Sparse constant propagation

\textbf{for} \( x \in \text{expressions} \)
\hspace{1em} value(x) \leftarrow \top, \ \text{const}(x), \ \text{or} \ \bot
worklist \leftarrow \emptyset
\textbf{for} \langle u, v \rangle \in \text{ssaEdges} \\
\hspace{1em} \textbf{if} \ \text{value}(u) \neq \top \\
\hspace{2em} \text{worklist} \leftarrow \text{worklist} \cup \{ \langle u, v \rangle \}
\textbf{while} \ \text{worklist} \neq \emptyset \\
\hspace{1em} \langle u, v \rangle \leftarrow \text{remove an edge from worklist} \\
\hspace{1em} \textbf{if} \ \text{value}(v) \neq \bot \\
\hspace{2em} t \leftarrow \text{eval}(v) \\
\hspace{2em} \textbf{if} \ t \neq \text{value}(v) \\
\hspace{3em} \text{value}(v) \leftarrow t \\
\hspace{3em} \text{worklist} \leftarrow \text{worklist} \cup \{ \langle v, w \rangle \mid \langle v, w \rangle \in \text{ssaEdges} \}
Speed

The limited depth of the lattice lets us talk about termination and complexity.

Each value \( x \) can only take on 3 values:
- \( \top, \text{const}, \bot \)

Each use can only be in worklist twice.

An optimistic algorithm: We initialize everything to \( \top \).
Branches

What happens when we propagate a value into a conditional branch?
- $\top \Rightarrow$ we gain no knowledge
- $\bot \Rightarrow$ either path can execute
- True of False $\Rightarrow$ only one path can execute

But the SC algorithm doesn't use this info.

Sparse conditional constant (SCC) adds the idea of refining feasible paths to improve our results.
- Until a block can execute, treat it as unreachable
- Optimistic initializations allow analysis to proceed, in spite of unevaluated blocks.

Result combines constant propagation with unreachable code elimination. Combination is stronger than either algorithm in isolation (even repeated).

An example of Click's notion of combining optimizations.
An important subtlety

Generally, we might say that

\[ x \oplus \top \Rightarrow \top \]

and

\[ x \oplus \bot \Rightarrow \bot \]

but since we also want

\[ 0 \times \bot \Rightarrow 0 \]

we'll need

\[ \top \times \bot \Rightarrow 0 \]

since the \( \top \) may later be lowered to 0.
Same for other operators with an annihilator.
And, ...

If we end up with expressions like

\[ \bot \times 1 \]

we can replace them with copies. Same for other operators with an identity.

This might also be a good time to rewrite

\[ \bot \times \text{const} \]

as a combination of shifts and adds.
An example

int zap(int x, int p) {
    int i = 0;
    do {
        p = p*i;
        i++;
    } while (p > 0);
    return x/i;
}

```
enter(x, p)
i ← 0

q ← ϕ(p, r)
j ← ϕ(i, k)
r ← q × j
k ← j + 1
r > 0

r

```
t ← x ÷ k
return t
An example

initialize

\[ \begin{align*}
  x & \rightarrow T \\
  p & \rightarrow T \\
  q & \rightarrow T \\
  r & \rightarrow T \\
  i & \rightarrow T \\
  j & \rightarrow T \\
  k & \rightarrow T \\
  t & \rightarrow T \\
\end{align*} \]

FlowWorkList = \{ 0 \}

SSAWorkList = \{ \}

\[
\begin{align*}
  & \text{enter}(x, p) \\
  & i \leftarrow 0 \\
  & q \leftarrow \varphi(p, r) \\
  & j \leftarrow \varphi(i, k) \\
  & r \leftarrow q \times j \\
  & k \leftarrow j + 1 \\
  & r > 0 \\
  & t \leftarrow x \div k \\
  & \text{return } t
\end{align*}
\]
An example

make edge executable

\[x \uparrow\]
\[p \uparrow\]
\[q \uparrow\]
\[r \uparrow\]
\[i \uparrow\]
\[j \uparrow\]
\[k \uparrow\]
\[t \uparrow\]

FlowWorkList = { }

SSAWorkList = { }

enter(x, p)
\[i \leftarrow 0\]

\[q \leftarrow \varphi(p, r)\]
\[j \leftarrow \varphi(i, k)\]
\[r \leftarrow q \times j\]
\[k \leftarrow j + 1\]
\[r > 0\]

\[t \leftarrow x \div k\]
return \[t\]
An example

visit expressions in block

\[
\begin{align*}
x &\perp \iff \bot \\
p &\perp \iff \bot \\
q &\top \\
r &\top \\
i &\top \\
j &\top \\
k &\top \\
t &\top
\end{align*}
\]

FlowWorkList = {}

SSAWorkList = \{ p, x \}
An example

visit expressions in block

\[
x \perp
\]
\[
p \perp
\]
\[
q \top
\]
\[
r \top
\]
\[
i \leftarrow 0 \Leftarrow 0
\]
\[
j \top
\]
\[
k \top
\]
\[
t \top
\]

FlowWorkList = \{ \}

SSAWorkList = \{ i, p, x \}
An example
add outgoing edge to worklist

\[
\begin{aligned}
x & \perp \\
p & \perp \\
q & \top \\
r & \top \\
i & 0 \\
j & \top \\
k & \top \\
t & \top
\end{aligned}
\]

FlowWorkList = \{ 1 \}

SSAWorkList = \{ i, p, x \}

```
enter(x, p)
  i ← 0
```

```
q ← φ(p, r)
j ← φ(i, k)
r ← q \times j
k ← j + 1
r > 0
```

```
t ← x ÷ k
return t
```
An example

make edge executable

\[
x \perp
\]
\[
p \perp
\]
\[
q \top
\]
\[
r \top
\]
\[
i \leftarrow 0
\]
\[
j \top
\]
\[
k \top
\]
\[
t \top
\]

FlowWorkList = \{
\}

SSAWorkList = \{ i, p, x \}
An example

visit phi nodes in block

\[
x \perp
\]
\[
p \perp
\]
\[
q \perp \quad \Leftarrow \perp \text{ (r edge is not executable)}
\]
\[
r \top
\]
\[
i 0
\]
\[
j \top
\]
\[
k \top
\]
\[
t \top
\]

FlowWorkList = \{ \}

SSAWorkList = \{ i, p, q, x \}
An example

visit phi nodes in block

\[ \begin{align*}
x \perp & \\
p \perp & \\
q \perp & \\
r & \top & \\
i & 0 & \\
j & 0 \leftarrow 0 \text{ (k edge is not executable)} & \\
k & \top & \\
t & \top & \\
\end{align*} \]

FlowWorkList = \{ \}

SSAWorkList = \{ i, j, p, q, x \}

```
enter(x, p)

i \leftarrow 0

q \leftarrow \varphi(p, r)

j \leftarrow \varphi(i, k)

r \leftarrow q \times j

k \leftarrow j + 1

r > 0

t \leftarrow x \div k

\text{return } t
```
An example

visit expressions in block

\[
\begin{align*}
x & \perp \\
p & \perp \\
q & \perp \\
r & 0 \iff \perp \times 0 \\
i & 0 \\
j & 0 \\
k & \top \\
t & \top
\end{align*}
\]

FlowWorkList = \{ \}

SSAWorkList = \{ i, j, p, q, r, x \}
An example

visit expressions in block

\[
x \perp \\
p \perp \\
q \perp \\
r \ 0 \\
i \ 0 \\
j \ 0 \\
k \ 1 \ \Leftarrow 0 + 1 \\
t \ \top
\]

FlowWorkList = \{\}

SSAWorkList = \{i, j, k, p, q, r, x\}
An example

visit expressions in block

\[
x \perp \\
p \perp \\
q \perp \\
r \ 0 \\
i \ 0 \\
j \ 0 \\
k \ 1 \\
t \top
\]

FlowWorkList = \{ 3 \}

SSAWorkList = \{ i, j, k, p, q, r, x \}
An example
make edge executable

\[ x \perp \]
\[ p \perp \]
\[ q \perp \]
\[ r \ 0 \]
\[ i \ 0 \]
\[ j \ 0 \]
\[ k \ 1 \]
\[ t \ \top \]

FlowWorkList = {}
An example

visit expressions in block

\[
x \perp \\
p \perp \\
q \perp \\
r \ 0 \\
i \ 0 \\
j \ 0 \\
k \ 1 \\
t \perp \leftarrow \perp \div 1
\]

FlowWorkList = { }

SSAWorkList = { i, j, k, p, q, r, t, x }
An example

visit expressions in block

\[
x \perp
p \perp
q \perp
r \ 0
i \ 0
j \ 0
k \ 1
t \perp
\]

\[
\text{FlowWorkList} = \{ \}
\]

\[
\text{SSAWorkList} = \{ i, j, k, p, q, r, t, x \}
\]
An example

start on SSAWorkList

\[
x \perp \\
p \perp \\
q \perp \\
r \ 0 \\
i \ 0 \\
j \ 0 \\
k \ 1 \\
t \ \perp
\]

FlowWorkList = \{ \}

SSAWorkList = \{ i, j, k, p, q, r, t, x \}
An example

start on SSAWorkList (but nothing interesting happens)

\[ x \perp \]
\[ p \perp \]
\[ q \perp \]
\[ r \ 0 \]
\[ i \ 0 \]
\[ j \ 0 \]
\[ k \ 1 \]
\[ t \perp \]

FlowWorkList = {}
An example

clean up

\[
x \perp \\
p \perp \\
q \perp \\
r \ 0 \\
i \ 0 \\
j \ 0 \\
k \ 1 \\
t \perp
\]
An example

delete dead code
An example

clean up CFG

\[
\text{enter}(x, p) \\
\quad t \leftarrow x \\
\text{return } t
\]
An example

Coalescing

```
enter(x, p)
\[ t \leftarrow x \]
return x
```
Value numbering

An expression $x + y$ is redundant iff, along every path from the entry, it has been evaluated and its subexpressions ($x$ and $y$) have not been redefined.

If the compiler can prove an expression is redundant
- It can arrange to preserve the earlier results, and
- It can replace the current evaluation with a reference.

Two parts to the problem
Proving that $x + y$ is redundant
Rewriting the IR to eliminate the redundant evaluation

A technique to accomplish both in a single pass is called value numbering.
Local value numbering

for each block $b$ (in any order)

for each operation $x ← y \otimes z$ in $b$ (working forward through $b$)

1) Get value numbers for each operand
2) Hash $\langle \otimes, \text{VN}(y), \text{VN}(z) \rangle$ to get a vn for the operation
3) If it already existed in the hash table, replace operation with a reference
4) If $y$ and $z$ are both constant, evaluate the operation and replace it
5) Store vn for $x$

If hashing behaves, the algorithm runs in linear time.

Using SSA means no names are overwritten!

Minor issues

- Sort operands for commutative operators
- Look for algebraic simplifications
- A value number is just a pointer
Local value numbering

- \( m \leftarrow a + b \)
- \( n \leftarrow a + b \)
- \( p \leftarrow c + d \)
- \( r \leftarrow d + c \)
- \( q \leftarrow a + b \)
- \( r \leftarrow c + d \)
- \( e \leftarrow b + 18 \)
- \( s \leftarrow a + b \)
- \( u \leftarrow e + f \)
- \( e \leftarrow a + 17 \)
- \( t \leftarrow c + d \)
- \( u \leftarrow e + f \)
- \( y \leftarrow a + b \)
- \( z \leftarrow c + d \)
- \( v \leftarrow a + b \)
- \( w \leftarrow c + d \)
- \( x \leftarrow e + f \)
Local value numbering

\begin{align*}
  m & \leftarrow a + b \\
  n & \leftarrow a + b \\
  p & \leftarrow c + d \\
  r & \leftarrow d + e \\
  q & \leftarrow a + b \\
  r & \leftarrow c + d \\
  e & \leftarrow b + 18 \\
  s & \leftarrow a + b \\
  u & \leftarrow e + f \\
  e & \leftarrow a + 17 \\
  t & \leftarrow c + d \\
  u & \leftarrow e + f \\
  v & \leftarrow a + b \\
  w & \leftarrow c + d \\
  x & \leftarrow e + f \\
  y & \leftarrow a + b \\
  z & \leftarrow c + d
\end{align*}

- 1 block at a time
- strong local results
- no cross-block effects

Need stronger methods
Extended basic blocks

m ← a + b
n ← a + b

p ← c + d
r ← d + c

q ← a + b
r ← c + d

e ← b + 18
s ← a + b
u ← e + f

e ← a + 17
t ← c + d
u ← e + f

v ← a + b
w ← c + d
x ← e + f

y ← a + b
z ← c + d

a tree of basic blocks
Extended basic blocks

\[
\begin{align*}
  m & \leftarrow a + b \\
  n & \leftarrow a + b
\end{align*}
\]

\[
\begin{align*}
  p & \leftarrow c + d \\
  r & \leftarrow d + e
\end{align*}
\]

\[
\begin{align*}
  e & \leftarrow b + 18 \\
  s & \leftarrow a + b \\
  u & \leftarrow e + f
\end{align*}
\]

\[
\begin{align*}
  q & \leftarrow a + b \\
  r & \leftarrow c + d
\end{align*}
\]

\[
\begin{align*}
  e & \leftarrow a + 17 \\
  t & \leftarrow c + d \\
  u & \leftarrow e + f
\end{align*}
\]

\[
\begin{align*}
  v & \leftarrow a + b \\
  w & \leftarrow c + d \\
  x & \leftarrow e + f
\end{align*}
\]

\[
\begin{align*}
  y & \leftarrow a + b \\
  z & \leftarrow c + d
\end{align*}
\]

a tree of basic blocks let's us benefit from our parent
Dominators

\[ m \leftarrow a + b \]
\[ n \leftarrow a + b \]
\[ p \leftarrow c + d \]
\[ r \leftarrow d + e \]
\[ y \leftarrow a + b \]
\[ z \leftarrow c + d \]
\[ q \leftarrow a + b \]
\[ r \leftarrow c + d \]
\[ e \leftarrow b + 18 \]
\[ s \leftarrow a + b \]
\[ u \leftarrow e + f \]
\[ v \leftarrow a + b \]
\[ w \leftarrow c + d \]
\[ x \leftarrow e + f \]
\[ e \leftarrow a + 17 \]
\[ t \leftarrow c + d \]
\[ u \leftarrow e + f \]
Effectiveness

Dominator-based value numbering is my favorite optimization. Easy to implement, very fast, very effective.

In some compilers, it slows the compiler down if you turn off VN (because later passes have so much more code to deal with).

Many opportunities for cleverness during implementation, but also for error. E.g., x - y might overflow whereas x == y won't.
Torturing optimizers

We can write test cases to expose what sort of algorithms an optimizer is using. Imagine a sequence of successively harder test case for value numbering, or cprop, or dead code elimination.

To test value numbering, we might start with something like

```c
void vn_test0(int *data) {
  int j = data[1]*data[3];
  int i = data[3];
  int m = data[1];
  int k = data[2];
  data[k] = 2;
  int n = m*i;
  data[0] = n - j;
}
```
Torturing optimizers

We can write test cases to expose what sort of algorithms an optimizer is using. Imagine a sequence of successively harder test case for value numbering, or cprop, or dead code elimination.

To test value numbering, we might start with something like

```c
void vn_test0(int *data) {
  int j = data[1]*data[3];
  int i = data[3];
  int m = data[1];
  int k = data[2];
  data[k] = 2;
  int n = m*i;
  data[0] = n - j;
}
```

```c
void vn_result0(int *data) {
  int k = data[2];
  data[k] = 2;
  data[0] = 0;
}
```
Do they work over EBBs?

```c
void vn_test1(int *data) {
    int j = data[1]*data[3];
    if (data[3] == 3) {
        int i = data[3];
        int m = data[1];
        int k = data[2];
        data[k] = 2;
        int n = m*i;
        data[0] = n - j;
    }
    else if (data[0] & 1) {
        j = 3 + data[2] - j;
        data[j] = 2;
    }
}
```
Do they work over EBBs?

void vn_test1(int *data) {
    int j = data[1]*data[3];
    if (data[3] == 3) {
        int i = data[3];
        int m = data[1];
        int k = data[2];
        data[k] = 2;
        int n = m*i;
        data[0] = n - j;
    } else if (data[0] & 1) {
        j = 3 + data[2] - j;
        data[j] = 2;
    }
}

void vn_result1(int *data) {
    int j = data[1]*data[3];
    if (data[3] == 3) {
        int k = data[2];
        data[k] = 2;
        data[0] = 0;
    } else if (data[0] & 1) {
        j = 3 + data[2] - j;
        data[j] = 2;
    }
}
How about dominators?

```c
void vn_test2(int *data) {
  int j = data[1]*data[3];
  int m = data[1];
  int k = j;
  if (data[0])
    j = j + 3;
  else
    j = j - 3;
  int n = data[3];
  j = data[2] + j;
  data[j] = 2;
  data[4] = k - m*n;
}
```
How about dominators?

void vn_test2(int *data) {
    int j = data[1]*data[3];
    int m = data[1];
    int k = j;
    if (data[0])
        j = j + 3;
    else
        j = j - 3;
    int n = data[3];
    j = data[2] + j;
    data[j] = 2;
    data[4] = k - m*n;
}

void vn_result2(int *data) {
    int j = data[1]*data[3];
    if (data[0])
        j = j + 3;
    else
        j = j - 3;
    j = data[2] + j;
    data[j] = 0;
    data[4] = 0;
}
How about derived facts?

```c
void vn_test3(int *data) {
    int i = data[0];
    int m = i + 1;
    int j = data[1];
    int n = j + 1;
    data[2] = m + n;
    if (i == j)
        data[3] = (m - n)*21;
}
```
How about derived facts?

```c
void vn_test3(int *data) {
    int i = data[0];
    int m = i + 1;
    int j = data[1];
    int n = j + 1;
    data[2] = m + n;
    if (i == j)
        data[3] = (m - n)*21;
}
```

```c
void vn_result3(int *data) {
    int i = data[0];
    int m = i + 1;
    int j = data[1];
    int n = j + 1;
    data[2] = m + n;
    if (i == j)
        data[3] = 0;
}
```