Compiler construction 2012

Lecture 10

More on code optimization

- SSA form
- Constant propagation
- Common subexpression elimination
- Loop optimizations

Static Single Assignment form

Def-use chains

Dataflow analysis often needs to connect a definition with its uses and, conversely, find all definitions reaching a use.
This can be simplified if each variable has only one definition.

A new form of IR

Three-address code can be converted to **SSA form** by renaming variables so that each variable has just one definition.

A non-example

```
s := 0
x := 1
s := s + x
x := x + 1
```

Converted to SSA

```
s1 := 0
x1 := 1
s2 := s1 + x1
x2 := x1 + 1
```


Conversion to SSA

A harder example

```
    s := 0
    x := 1
L1: if x > n goto L2
    s := s + x
    x := x + 1
    goto L1
L2:
```

Note the def/use difficulty:
In `s + x`, which def of `s` does the use refer to?

Conversion started

```
    s1 := 0
    x1 := 1
L1: if x > n goto L2
    s2 := s? + x?
    x2 := x? + 1
    goto L1
L2:
```

What should replace the three '?' ?

An artificial device: ϕ-functions

Naive version

First pass: Add "definitions" of the form
$x := \phi(x, \ldots, x)$ in the beginning of each block with several predecessors and for each variable.
Second pass: Do renumbering.

After 1st pass

```
    s := 0
    x := 1
L1: s := φ(s,s)
    x := φ(x,x)
    if x > n goto L2
    s := s + x
    x := x + 1
    goto L1
L2:
```

After 2nd pass

```
    s1 := 0
    x1 := 1
L1: s3 := φ(s1,s2)
    x3 := φ(x1,x2)
    if x3 > n goto L2
    s2 := s3 + x3
    x2 := x3 + 1
    goto L1
L2:
```

What are ϕ-functions?

A device during optimization

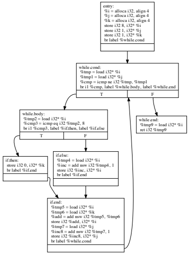
Think of ϕ-functions as function calls during optimization.
Later, some of them will be eliminated (e.g. by dead code elimination).
Others will after optimization be transformed to real code.
Idea: `x3 := ` ϕ`(x1,x2)` will be transformed to an instruction `x3 := x1` at the end of left predecessor and `x3 := x2` at end of right predecessor.

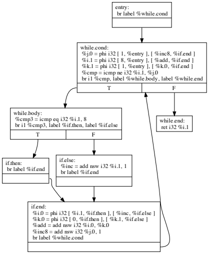
Advantages

Many analyses become much simpler when code is in SSA form.

Main reason: we see immediately for each use of a variable where it was defined.

Step 2 of example revisited: To SSA form





Computing SSA form; algorithm

We already did this

Yes, but the conversion inserts unnecessary ϕ-functions and is too inefficient – the gains in analysis with SSA form may be lost in conversion.

Better algorithms

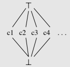
There are algorithms for finding the right number of ϕ-functions needed. These are based on the notion of **dominance**; if you intend to use SSA form, you need to learn about that – or use LLVM, which has tools to do it for you.

Simple constant propagation

A dataflow analysis based on SSA form

Uses values from a **lattice** *L* with elements
$\top$: Certainly not a constant.
$c_1, c_2, c_3, \ldots$: The value is constant, as indicated.
$\bot$: Yet unknown, may be constant.
Each variable `v` is assigned an initial value $\mathit{val}(\mathtt{v}) \in L$:
Variables with definitions `v := c` get $\mathit{val}(\mathtt{v}) = \mathtt{c}$,
input variables/parameters `v` get $\mathit{val}(\mathtt{v}) = \top$,
and the rest get $\mathit{val}(\mathtt{v}) = \bot$.

The lattice *L*



The lattice order

$\bot \le c \le \top$ for all c.
c_i and c_j not related.

Propagation phase, 1

Iteration

Initially, place all names n with $val(\mathtt{n}) \neq \top$ on a worklist.
Iterate by picking a name from the worklist, examining its uses and computing *val* of the RHS's, using rules as

$$\begin{aligned} 0 \cdot x &= 0 \quad (\text{for any } x) \\ x \cdot \bot &= \bot \\ x \cdot \top &= \top \quad (x \neq 0) \end{aligned}$$

plus ordinary multiplication for constant operands.

For ϕ-functions, we take the join $\vee$ of the arguments, where $\bot \vee x = x$ for all x, $\top \vee x = \top$ for all x, and

$$c_i \vee c_j = \begin{cases} \top, & \text{if } c_i \neq c_j \\ c_i, & \text{otherwise.} \end{cases}$$

Propagation phase, 2

Iteration, continued

Update *val* for the defined variables, putting variables that get a new value back on the worklist.
Terminate when worklist is empty.

Termination

Values of variables on the worklist can only increase (in lattice order) during iteration. Each value can only have its value increased twice.

A disappointment

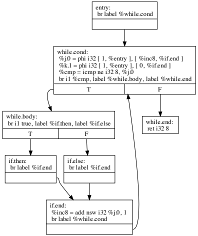
In our running example, this algorithm will terminate with all variables having value $\top$.

We need to take **reachability** into account.

Sparse Conditional Constant Propagation

Sketch of algorithm

- Uses also a worklist of `reachable` blocks.
- Initially, only the entry block is reachable.
- In evaluation of ϕ functions, only $\bot$ flows from unreachable blocks.
- New blocks added to worklist when elaborating terminating instructions.
- Result for running example as shown to the right (to be done in class).



Correctness of SCCP

A combination of two dataflow analyses

Sparse conditional constant propagation can be seen as the combination of simple constant propagation and reachability analysis/dead code analysis.

Both of these can be expressed as dataflow problems and a framework can be devised where the correctness of such combination can be proved.

Final steps

Control flow graph simplification

Fairly simple pass; SCCP does not change graph structure of CFG even when “obvious” simplifications can be done.

Dead Loop Elimination

Identifies an induction variable (namely `j`), which

- increases with 1 for each loop iteration,
- terminates the loop when reaching a known value,
- is initialised to a smaller value.

When such a variable is found, loop termination is guaranteed and the loop can be removed.

Common subexpression elimination

Problem

We want to avoid re-computing an expression; instead we want to use the previously computed value.

Code example

```
a := b + c
b := a - d
c := b + c
d := a - d
```

Notes

The second occurrence of `a - d` should not be computed; instead we should use `d := b`.

Both occurrences of `b + c` must be computed, since `b` is redefined in-between.

Value numbering, 1

A classic technique

Works on three-address code within a basic block.

Each expression is assigned a **value number** (VN), so that expressions that have the same VN must have the same value. (Note: The VN is **not** the value of the expression.)

Data structures

- A dictionary D_1 that associates
  - a variable or a literal with a VN.
  - a triple (VN,operator,VN) with a VN.

  Typically, D_1 is implemented as a hash table.
- A dictionary D_2, mapping VNs to sets of variables (implemented as an array).

Value numbering, 2

Algorithm

For each instruction `x := y # z`:

- Look up VN n_y for `y` in D_1.
  If not present, generate new unique VN n_y and put $D_1(\mathtt{y}) = n_y$, $D_2(n_y) = \mathtt{y}$.
- Do the same for `z`.
- Look up `x` in D_1; if n found, remove `x` from $D_2(n)$.
- Look up (n_y,#,n_z) in D_1.
  If VN m found,
  - insert $D_1(\mathtt{x}) = m$ (m has been computed before).
  - if $D_2(m)$ is non-empty, replace instruction by `x := v` for some `v` in that set.

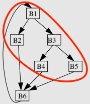
  Otherwise, generate new unique VN m and put $D_1(n_x, \#, n_y) = m$, $D_1(\mathtt{x}) = m$.
- Add `x` to $D_2(m)$.

Value numbering, 3

Extended basic blocks

A subtree of the CFG where each node has only one predecessor. Each path through the EBB is handled by value numbering.

To avoid starting from scratch, use stacks of dictionaries. (Needs SSA form.)



Algebraic identities

Value numbering can be combined with code improvement using identities such as

$$
\begin{aligned}
x \cdot 0 &= 0\\
0 \cdot x &= 0\\
x \cdot 1 &= x\\
1 \cdot x &= x\\
\ldots &= \ldots
\end{aligned}
$$

Avoid long sequences of tests!

Available expressions: a dataflow analysis

Purpose

An auxiliary concept in an **intraprocedural** analysis for finding common subexpressions.

Definition

An expression `x # y` is **available** at a point P in a CFG if the expression is evaluated on every path from the entry node to P **and** neither `x` nor `y` is redefined after the last such evaluation.

Locally defined sets

We consider sets of **expressions**.
gen(n) is the set of expressions `x # y` that are evaluated in n without subsequent definition of `x` or `y`.
kill(n) is the set of expressions `x # y` where n defines `x` or `y` without subsequent evaluation of `x # y`.

Available expressions: the flow equations

Sets to compute by flow analysis

avail-in(n) is the set of available exprs at the beginning of n.
avail-out(n) is the set of available exprs at the end of n.

$$
\begin{aligned}
\mathit{avail\text{-}out}(n) &= \mathit{gen}(n) \cup (\mathit{avail\text{-}in}(n) - \mathit{kill}(n))\\
\mathit{avail\text{-}in}(n_0) &= \{\} \quad \text{for the entry node } n_0\\
\mathit{avail\text{-}in}(n) &= \cap_{p \in \mathit{preds}(n)} \mathit{avail\text{-}out}(p) \quad (\text{other } n)
\end{aligned}
$$

Motivation

An expr is available on exit from n if it is either generated in n **or** it was already available on entry and not killed in n.

An expr is available on entry if it is available from **all** preds.

Available expressions: Comments

Solution method

- Iteration from the initial sets $\mathit{avail\text{-}in}(n) = \mathit{avail\text{-}out} = U$, where U is the set of **all** expressions occurring in the CFG (except for $\mathit{avail\text{-}in}(n_0) = \{\}$).
- Converges to the **greatest** fixpoint. All sets **shrink** monotonically during iterations.
- Fixpoint solution has the property that any expr declared available **is** really available.
  This does **not** hold for previous iterations.
- Sets can be represented as bit-vectors (U = all ones).
- This is a **forward** problem; information flows from predecessors to successors.
  Thus one should try to compute predecessors first.

Common subexpression elimination

Available expressions can be eliminated

If dataflow **analysis** finds that `y # z` in an instruction `x := y # z` is available we could eliminate it.
This a second, separate step (**code transformation**): replace instruction by `x := w`. But how to find `w`?

Basic idea

Generate a new name `w`. Follow the control backwards along all paths until a definition `v := y # z` is found (such a def must exist in all paths!).
Replace the def by

```
w := y # z
v := w
```

A more powerful idea

Find these definitions by dataflow analysis: **reaching definitions**.

Tail recursion

A different optimization

A recursive function is **tail-recursive** if it returns a value computed by (just) a recursive call. This can (and should) be optimized to a loop.

Recursive form

```
int sumTo(int lim) {
  return ack(1,lim,0);
}
int ack(int n,int k,int s){
  if n>k then
     return s;
  else
     return ack(n+1,k,s+n);
}
```

`ack` rewritten

```
int ack(int n,int k,int s){
L:  if n>k then
      return s;
    else
      k = k; // not needed
      s = s+n;
      n = n+1;
      // note reordering!
      goto L;
}
```


A motivating example

A simple Javalette function (in extension arrays1)

```
int sum (int [] a) {
  int res=0;
  for (int x : a)
    res = res + x;
  return res;
}
```

What code would you generate?

Possible naive LLVM code, part 1

```
%arr = type { i32, [ 0 x i32 ] }*
define i32 @sum(%arr %__p__a) {
entry:  %a = alloca %arr
        store %arr %__p__a , %arr* %a
        %_res_t0 = alloca i32
        store i32 0 , i32* %_res_t0
        %_x_t1 = alloca i32
        %t2 = load %arr* %a
        %t3 = getelementptr %arr %t2 , i32 0, i32 0
        %t4 = load i32* %t3
        %_indexx_t5 = alloca i32
        store i32 0 , i32* %_indexx_t5
        br label %lab0
lab0:   %t6 = load i32* %_indexx_t5
        %t7 = icmp slt i32 %t6 , %t4
        br i1 %t7 , label %lab1 , label %lab2
```

Possible naive LLVM code, part 2

```
lab1: %t8 = getelementptr %arr %t2 , i32 0,  i32 1, i32 %t6
      %t9 = load i32* %t8
      store i32 %t9 , i32* %_x_t1
      %t10 = load i32* %_res_t0
      %t11 = load i32* %_x_t1
      %t12 = add i32 %t10 , %t11
      store i32 %t12 , i32* %_res_t0
      %t13 = add i32 %t6 , 1
      store i32 %t13 , i32* %_indexx_t5
      br label %lab0

lab2: %t14 = load i32* %_res_t0
      ret  i32 %t14
}
```


After `opt -mem2reg`

```
define i32 @sum(%arr %__p__a) {
entry: %t3 = getelementptr %arr %__p__a, i32 0, i32 0
       %t4 = load i32* %t3
       br label %lab0

lab0: %_res_t0.0 = phi i32 [ 0, %entry ], [ %t12, %lab1 ]
      %_indexx_t5.0 = phi i32 [ 0, %entry ], [ %t13, %lab1 ]
      %t7 = icmp slt i32 %_indexx_t5.0, %t4
      br i1 %t7, label %lab1, label %lab2

lab1: %t8 = getelementptr %arr %__p__a, i32 0, i32 1, i32 %_indexx_t5.0
      %t9 = load i32* %t8
      %t12 = add i32 %_res_t0.0, %t9
      %t13 = add i32 %_indexx_t5.0, 1
      br label %lab0

lab2: ret i32 %_res_t0.0
}
```


After `opt -std-compile-opts`

```
define i32 @sum(%arr nocapture %__p__a) nounwind readonly {
entry: %t3 = getelementptr %arr %__p__a, i32 0, i32 0
       %t4 = load i32* %t3
       %t71 = icmp sgt i32 %t4, 0
       br i1 %t71, label %bb.nph, label %lab2

bb.nph: %tmp = zext i32 %t4 to i64
        br label %lab1

lab1: %indvar = phi i64 [ 0, %bb.nph ], [ %indvar.next, %lab1 ]
      %_res_t0.02 = phi i32 [ 0, %bb.nph ], [ %t12, %lab1 ]
      %t8 = getelementptr %arr %__p__a, i64 0, i32 1, i64 %indvar
      %t9 = load i32* %t8
      %t12 = add i32 %t9, %_res_t0.02
      %indvar.next = add i64 %indvar, 1
      %exitcond = icmp eq i64 %indvar.next, %tmp
      br i1 %exitcond, label %lab2, label %lab1

lab2: %_res_t0.0.lcssa = phi i32 [ 0, %entry ], [ %t12, %lab1 ]
      ret i32 %_res_t0.0.lcssa
}
```


Generated x86 assembly (with `llc`)

```
_sum:    push   EDI
         push   ESI
         mov    ECX, DWORD PTR [ESP + 12]
         mov    EDX, DWORD PTR [ECX]
         test   EDX, EDX
         jg     LBB0_2
         xor    EAX, EAX
         jmp    LBB0_4
LBB0_2:  xor    ESI, ESI
         add    ECX, 4
         xor    EAX, EAX
LBB0_3:  add    EAX, DWORD PTR [ECX]
         add    EDX, -1
         adc    ESI, -1
         add    ECX, 4
         mov    EDI, EDX
         or     EDI, ESI
         jne    LBB0_3
LBB0_4:  pop    ESI
         pop    EDI
         ret
```

Comments

- No local vars; no stack frame handling.
- Uses callee save registers EDI and ESI; note save/restore.
- ECX holds address of current array elem; increased by 4 in each iteration.
- EDX counts nr of elems remaining.
- Use of ESI in loop termination test??

Optimizations of loops

In computationally demanding applications, most of the time is spent in executing (inner) loops.

Thus, an optimizing compiler should focus its efforts in improving loop code.

The first task is to identify loops in the code. In the source code, loops are easily identified, but how to recognize them in a low level IR code?

A loop in a CFG is a subset of the nodes that

- has a **header** node, which dominates all nodes in the loop.
- has a **back edge** from some node in the loop back to the header. A back edge is an edge where the head dominates the tail.

Moving loop-invariant code out of the loop

A simple example

```
for (i=0; i<n; i++)
  a[i] = b[i] + 3*x;
```

should be replaced by

```
t = 3*x;
for (i=0; i<n; i++)
  a[i] = b[i] + t;
```

We need to insert an extra node (a **pre-header**) before the header.

Not quite as simple

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    a[i][j] = b[i][j]+10*i+3*x;
```

should be replaced by

```
t = 3*x;
for (i=0; i<n; i++) {
  u = 10*i + t;
  for (j=0; j<n; j++)
    a[i][j] = b[i][j] + u;
}
```


Induction variables

A **basic** induction variable is an (integer) variable which has a single definition in the loop body, which increases its value with a fixed (loop-invariant) amount.

Example: `n = n + 3`

A basic IV will assume values in arithmetic progression when the loop executes.

Given a basic IV we can find a collection of **derived** IV's, each of which has a single def of the form
`m` = a*n+b;
where `a` and `b` are loop-invariant.
The def can be extended to allow RHS of the form `a*k+b` where also `k` is an already established derived IV.

Strength reduction for IV's

`n` is a basic IV (only def is to increase by 1).
`k` is derived IV.

Replace multiplication involved in def of `k` by addition.

```
while (n<100) {
   k = 7*n + 3;
   a[k]++;
   n++;
}
```

Replace multiplication involved in def of derived IV by addition.

```
k = 7*n + 3;
while (n<100) {
   a[k]++;
   n++;
   k+=7;
}
```

Could there be some problem with this transformation?

Strength reduction for IV's, continued

The loop might not execute at all, in which case k would not be evaluated.
Better to perform loop inversion first.

```
if (n<100) {
   k = 7*n + 3;
   do {
     a[k]++;
     n++;
     k+=7;
   } while (n<100);
}
```

If n is not used after the loop, it can be eliminated from the loop

```
if (n<100) {
   k = 7*n + 3;
   do {
     a[k]++;
     k+=7;
   } while (k<703);
}
```


One more example

Sample loop

```
int sum = 0;
for(i=0; i<1000; i++)
  sum += a[i];
```

What can these techniques do for this loop?

Naive assembler code

```
    %sum = 0
    %i = 0
L1: %off = mul %i, 4
    %addr = add %addr.a,%off
    %a.i = load %addr
    %sum = add %sum,%a.i
    %i = add %i, 1
    %stop = cmp lt %i,1000
    br %stop, L1, L2
L2:
```

Strength reduction/IV techniques

```
    %sum = 0
    %off = 0
    %addr = %addr.a
    %end = add %addr.a,4000
L1: %a.i = load %addr
    %sum = add %sum,%a.i
    %addr = add %addr, 4
    %stop = cmp lt %addr,%end
    br %stop, L1, L2
L2:
```


Loop unrolling

```
for (i=0; i<100; i++)
   a[i] = a[i] + x[i]
```

```
for (i=0; i<100; i=i+4) {
   a[i] = a[i] + x[i]
   a[i+1] = a[i+1] + x[i+1]
   a[i+2] = a[i+2] + x[i+2]
   a[i+3] = a[i+3] + x[i+3]
}
```

- In which ways is this an improvement?
- What to do if upper bound is n?
- Is unrolling four steps the best choice?
- What could be the disadvantages?

Optimizations in gcc

On ASTs

Inlining, constant folding, arithm. simplification.

On RTL code ($\approx$ three-address code)

- Tail (and sibling) call optimization.
- Jump optimization.
- SSA pass: constant propagation, dead code elimination.
- Common subexpression elimination, more constant propagation.
- Loop optimization.
- ...

Difficult decisions: optimization order, repetitions.

Summing up

On optimization

We have only looked at a few of many, many techniques.

Modern optimization techniques use sophisticated algorithms and clever data structures.

Frameworks such as LLVM make it possible to get the benefits of state-of-the-art techniques in your own compiler project.

Rest of course

- No more lectures.
- Submit project next Thursday.
- Oral exam in exam week.