Compiler construction 2011

Lecture 9

Code optimization

- General comments
- An example in LLVM
- SSA conversion
- Constant propagation
- Value numbering
- More algorithms

Desired properties

Improve the code

- Make execution faster.
- Make execution consume less power.
- Make program smaller.

These goals can be contradictory.

Don't change semantics

- Don't change values returned.
- Don't change side effects.
- Don't change runtime errors(!).
- Don't change termination properties.

Often subtle points.

Full optimization is impossible

Full employment theorem for compiler writers

We cannot build a compiler that optimizes all programs fully for program size.

Proof: The smallest non-terminating program without visible effects is `while (true) {}`
A fully optimizing compiler would translate any non-terminating program to this – and thus solve the halting problem.

Similar results for other optimization criteria.

Optimization at different stages

Where/when should we optimize?

We can optimize at different stages:

- Source code.
- Abstract syntax trees.
- Three-address code.
- Native code.

Except for source code, compilers do optimization at all these stages.

Code optimization

Improvement opportunities

- Naive syntax-directed translation often gives code that can be "obviously" improved.
- Compiler-generated code such as e.g. address calculations for array elements even more so.
- One improvement often opens for other improvements.

Consequences

- If you know that subsequent optimizations will be done, do not try to be clever in the first code generation step.
- Never rule out an optimization as useless by thinking that "the programmer would never write that" – the compiler itself might do so!

Inlining

Replace function call by body

Parameters need to be substituted by arguments.
Renaming of vars may be needed.

+ Function call overhead disappears.
+ Activation record disappears.
+ Memory traffic reduced.
+ New optimization opportunities.
- Code becomes bigger.

This is often done at AST levels.

For imperative code (w. statements and return),
rewrite to return a var and place the var at the call site.

In the rest of the lecture, we focus on three address code/native code optimization.

An example of optimization in LLVM

```
int f () {
  int i, j, k;
  i = 8;
  j = 1;
  k = 1;
  while (i != j) {
    if (i==8)
      k = 0;
    else
      i++;
    i = i+k;
    j++;
  }
  return i;
}
```

Comments

Human reader sees, with some effort, that the C/Javalette function `f` returns 8.

We follow how LLVM:s optimizations will discover this fact.

Step 1: Naive translation to LLVM

```
define i32 @f()  {
entry:
  %i = alloca i32
  %j = alloca i32
  %k = alloca i32
  store i32 8, i32* %i
  store i32 1, i32* %j
  store i32 1, i32* %k
  br label %while.cond
while.cond:
  %tmp = load i32* %i
  %tmp1 = load i32* %j
  %cmp = icmp ne i32 %tmp, %tmp1
  br i1 %cmp, label %while.body,
              label %while.end
while.body:
  %tmp2 = load i32* %i
  %cmp3 = icmp eq i32 %tmp2, 8
  br i1 %cmp3, label %if.then,
               label %if.else
if.then:
  store i32 0, i32* %k
  br label %if.end
if.else:
  %tmp4 = load i32* %i
  %inc = add i32 %tmp4, 1
  store i32 %inc, i32* %i
  br label %if.end
if.end:
  %tmp5 = load i32* %i
  %tmp6 = load i32* %k
  %add = add i32 %tmp5, %tmp6
  store i32 %add, i32* %i
  %tmp7 = load i32* %j
  %inc8 = add i32 %tmp7, 1
  store i32 %inc8, i32* %j
  br label %while.cond
while.end:
  %tmp9 = load i32* %i
  ret i32 %tmp9
}
```

Step 2: Translating to SSA form (`opt -mem2reg`)

```
define i32 @f() {
entry:
  br label %while.cond

while.cond:
  %k.1 = phi i32 [ 1, %entry ],
                 [ %k.0, %if.end ]
  %j.0 = phi i32 [ 1, %entry ],
                 [ %inc8, %if.end ]
  %i.1 = phi i32 [ 8, %entry ],
                 [ %add, %if.end ]
  %cmp = icmp ne i32 %i.1, %j.0
  br i1 %cmp, label %while.body,
              label %while.end

while.body:
  %cmp3 = icmp eq i32 %i.1, 8
  br i1 %cmp3, label %if.then,
               label %if.else
```

```
if.then:
  br label %if.end

if.else:
  %inc = add i32 %i.1, 1
  br label %if.end

if.end:
  %k.0 = phi i32 [ 0, %if.then ],
                 [ %k.1, %if.else ]
  %i.0 = phi i32 [ %i.1, %if.then ],
                 [ %inc, %if.else ]
  %add = add i32 %i.0, %k.0
  %inc8 = add i32 %j.0, 1
  br label %while.cond

while.end:
  ret i32 %i.1
}
```


Step 3: Sparse Conditional Constant Propagation (`opt -sccp`)

```
define i32 @f() {
entry:
  br label %while.cond

while.cond:
  %j.0 = phi i32 [ 1, %entry ],
                 [ %inc8, %if.end ]
  %k.1 = phi i32 [ 1, %entry ],
                 [ 0, %if.end ]
  %cmp = icmp ne i32 8, %j.0
  br i1 %cmp, label %while.body,
              label %while.end

while.body:
  br i1 true, label %if.then,
              label %if.else
```

```
if.then:
  br label %if.end

if.else:
  br label %if.end

if.end:
  %inc8 = add i32 %j.0, 1
  br label %while.cond

while.end:
  ret i32 8
}
```


Step 4: CFG Simplification (`opt -simplifycfg`)

```
define i32 @f() {
entry:
  br label %while.cond

while.cond:
  %j.0 = phi i32 [ 1, %entry ],
                 [ %inc8, %if.end ]
  %k.1 = phi i32 [ 1, %entry ],
                 [ 0, %if.end ]
  %cmp = icmp ne i32 8, %j.0
  br i1 %cmp, label %if.end,
              label %while.end

if.end:
  %inc8 = add i32 %j.0, 1
  br label %while.cond

while.end:
  ret i32 8
}
```

Comments

If the function terminates, the return value is 8.

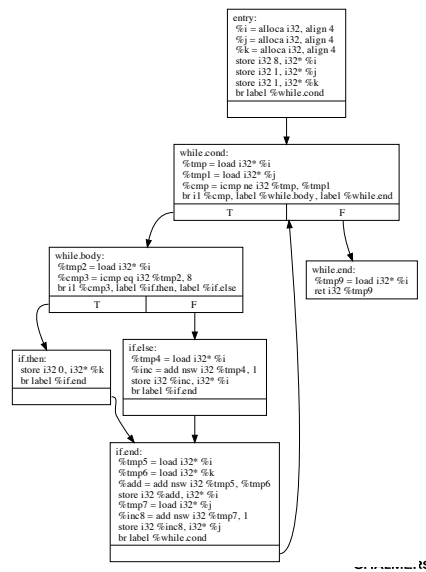
`opt` has not yet detected that the loop is certain to terminate.

Step 5: Dead Loop Deletion `opt -loop-deletion`)

```
define i32 @f() {
entry:
  br label %while.end

while.end:
  ret i32 8
}
```

When discussing this sequence of improvements we will make use of a further capability of `opt`:
To generate graph visualizations of the CFG. To the right is the graph after Step 1.



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

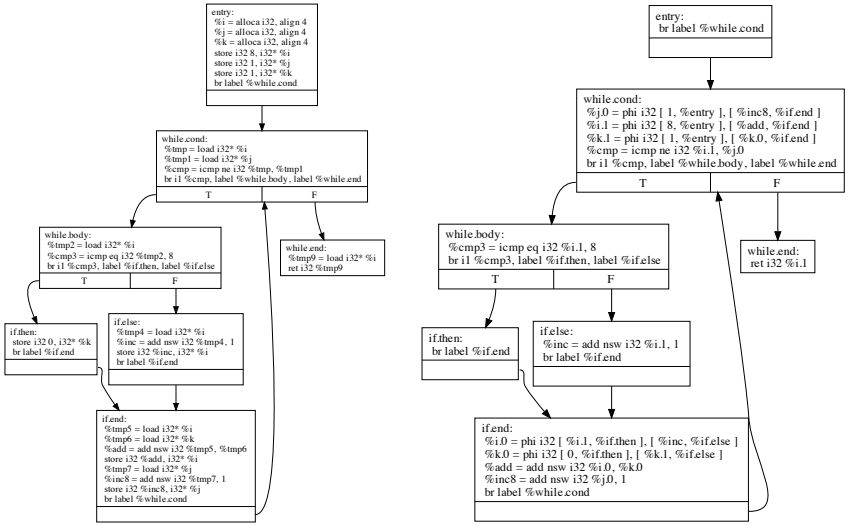
Think of ϕ-functions as function calls during optimization.
Later, some of them will be eliminated (e.g. by dead code elimination).
Others will after optimization be transformed to real code.
Idea: `x3 := φ(x1,x2)` will be transformed to an instruction `x3 := x1` at the end of left predecessor and `x3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

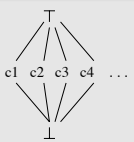
There are algorithms for finding the right number of ϕ-functions needed. These are based on the notion of **dominance**; if you intend to use SSA form, you need to learn about that – or use LLVM, which has tools to do it for you.

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 $\mathtt{v}$ is assigned an initial value $val(\mathtt{v}) \in L$:
Variables with definitions $\mathtt{v}$:= $\mathtt{c}$ get $val(\mathtt{v}) = \mathtt{c}$,
input variables/parameters $\mathtt{v}$ get $val(\mathtt{v}) = \top$,
and the rest get $val(\mathtt{v}) = \bot$.

The lattice L



The lattice order

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

Propagation phase, 1

Iteration

Initially, place all names $\mathtt{n}$ 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

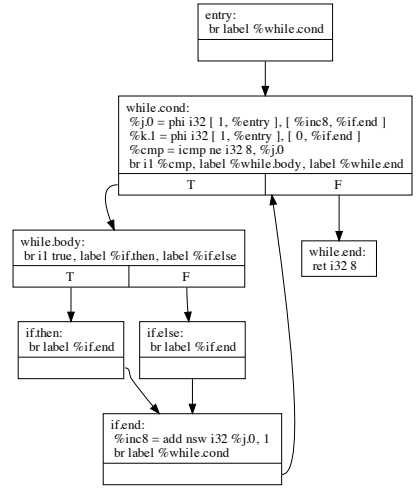
In our running example, this algorithm will terminate with all variables having value $\top$.

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 $D_1(\texttt{y}) = n_y$, $D_2(n_y) = \texttt{y}$.
- 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 $D_1(\texttt{x}) = m$ (m has been computed before).
  - if $D_2(m)$ is non-empty, replace instruction by `x := v` for some `v` in that set.

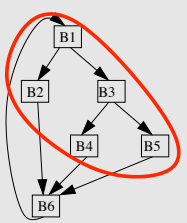
  Otherwise, generate new unique VN m and put $D_1(n_x, \#, n_y) = m$, $D_1(\texttt{x}) = m$.
- Add `x` to $D_2(m)$.

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.

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

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 $\textit{avail-in}(n) = \textit{avail-out} = U$, where U is the set of **all** expressions occurring in the CFG (except for $\textit{avail-in}(n_0) = \{\}$).
- 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
     n = n+1;
     k = k
     s = s+n;
     goto L;
}
```


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.

What next?

- Last lecture on Thursday: More on optimization.
- Submission deadline next Thursday.
- Oral exams in the exam week. Book a time slot!